**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 | We are able to stop all software vulnerabilities by checking the inputs of all untrusted data. We must be aware of all types of data sources, particularly external data, when validating inputs. |
| 1. Heed Compiler Warnings | When writing code, it's crucial to pay attention to the warnings. By altering the code, we must always erase all warnings. Security issues remain when warnings are ignored. |
| 1. Architect and Design for Security Policies | It is crucial that you consider and adhere to security regulations when creating your software architecture. |
| 1. Keep It Simple | It's crucial to keep your code as straightforward as you can when creating it. Making things more complicated than necessary creates potential for mistakes that could lead to vulnerabilities. |
| 1. Default Deny | Standardize how things are carried out. If things are not carried out in that precise manner, always deny it. By maintaining a regular procedure, if something is done differently, we are aware that something is amiss. |
| 1. Adhere to the Principle of Least Privilege | Give processes access for just the amount of time necessary to complete their tasks. The same issue happens when processes that need more access are running. Access restrictions lower the likelihood of vulnerability. |
| 1. Sanitize Data Sent to Other Systems | Sanitization will be used to stop injection attacks. Data sent will be able to go via a filter before being transferred to the other system thanks to data sanitization. |
| 1. Practice Defense in Depth | Always use numerous levels of protection so that, if one fails, there are still others. |
| 1. Use Effective Quality Assurance Techniques | Be certain that the security is functioning properly by incorporating a process to adhere to. Testing, evaluating, and outside viewpoints are all included. |
| 1. Adopt a Secure Coding Standard | Always have an appropriate secure coding standard for the language you are using that you can adhere to. |

### 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] | Do not cast to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| Checking if the value is within the enumeration value range. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| Checking the value before conversion. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** The values are not specified making an overflow of the buffer possible. This creates a vulnerability that could be exploited by an attacker. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Cast-integer-to-enum | Partially checked |
| CodeSonar | 7.3p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| RuleChecker | 22.10 | Cast-to-integer-enum | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use valid references, pointers, and iterators to reference elements of a container. |

| **Noncompliant Code** |
| --- |
| Invalidated after the first call, and loop iterations have undefined behavior. |
| void f(const double \*items, std::size\_t count) {  std::deque<double> d;  auto pos = d.begin();  for (std::size\_t i = 0; i < count; ++i, ++pos) {  d.insert(pos, items[i] + 41.0);  }  } |

| **Compliant Code** |
| --- |
| Assigned a valid iterator on each insertion, preventing undefined behavior. |
| void f(const double \*items, std::size\_t count) {  std::deque<double> d;  auto pos = d.begin();  for (std::size\_t i = 0; i < count; ++i, ++pos) {  pos = d.insert(pos, items[i] + 41.0);  }  } |

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

| **Principles(s):** Causes risks that the underlying container could be modified to the point where the iterator, pointer, or reference becomes invalid. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use After Free |
| Klockwork | 2023.1 | ITER.CONTAINER.MODIFIED | -- |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-CTR51-a | Do no modify container while iterating over it |
| Polyspace Bug Finder | R2023a | CERT C++: CTR51-CPP | Checks for use of invalid iterator (rule partially covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| The char pointer str is initialized to the address of a string literal, and attempting to modify the string literal is undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely |
| char str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** Modifying a string literal often results in an access violation because string literals are usually stored in read-only memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to “char \*” |
| LDRA tool suite | 9.7.1 | 157 S | Partially implemented |
| Polyspace Bug Finder | R2023a | CERT C: Rule STR30-C | Check for writing to const qualified object (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Provide a valid ordering predicate. |

| **Noncompliant Code** |
| --- |
| Object is created with a comparator that does not adhere to the strict weak ordering requirement. |
| #include <functional>  #include <iostream>  #include <set>    void f() {  std::set<int, std::less\_equal<int>> s{5, 10, 20};  for (auto r = s.equal\_range(10); r.first != r.second; ++r.first) {  std::cout << \*r.first << std::endl;  }  } |

| **Compliant Code** |
| --- |
| Uses the default comparator with std::set instead of providing an invalid one. |
| #include <iostream>  #include <set>    void f() {  std::set<int> s{5, 10, 20};  for (auto r = s.equal\_range(10); r.first != r.second; ++r.first) {  std::cout << \*r.first << std::endl;  }  } |

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

| **Principles(s):** Using an invalid ordering rule can lead to erratic behavior or infinite loops. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.1 | C++ 3293 | -- |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-CTR57-a | For associative containers never use comparison function returning true for equal values |
| Polyspace Bug Finder | R2023a | CERT C++: CTR57-CPP | Check for predicate lacking strict weak ordering (rule partially covered) |

#### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| 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** |
| --- |
| 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):** Pointers to memory that have been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Clang-analyzer-cplusplus.NewDelete | Checked by clang-tidy. But does not catch all violations of this rule. |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use after free |
| Coverity | V7.5.0 | 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 |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Avoid cycles during initialization of static objects. |

| **Noncompliant Code** |
| --- |
| The initialization of the static local array cache involves recursion, the behavior of the function is undefined. |
| #include <stdexcept>    int fact(int i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    static const int cache[] = {  fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),  fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),  fact(12), fact(13), fact(14), fact(15), fact(16)  };    if (i < (sizeof(cache) / sizeof(int))) {  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

| **Compliant Code** |
| --- |
| Avoids initializing the static local array cache and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet. |
| #include <stdexcept>    int fact(int i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    // Use the lazy-initialized cache.  static int cache[17];  if (i < (sizeof(cache) / sizeof(int))) {  if (0 == cache[i]) {  cache[i] = i > 0 ? i \* fact(i - 1) : 1;  }  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

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

| **Principles(s):** Reentering a function during the initialization of one of its static objects can result in an attacker being able to cause a crash or denial of service. Indeterminately ordered dynamic initialization can lead to undefined behavior due to accessing an uninitialized object. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.INIT.CYCLE  LANG.STRUCT.INIT.UNORDERED | Initialization Cycle  Unordered Initialization |
| LDRA tool suite | 9.7.1 | 6 D | Enhanced Enforcement |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL56-a | Avoid initialization order problems across translation units by replacing non-local static objects with local static objects |
| Polyspace Bug Finder | R2023a | CERT C++: DCL56-CPP | Checks for:  Recursive initialization of static variables  Undetermined initialization order of global variables  Rule fully covered. |

#### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions thrown by throwing\_func(). |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all exceptions. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  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 | R2023a | CERT C++: ERR51-CPP | Checks for unhadled exceptions (rule partially covered) |

#### Coding Standard 8

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

| **Noncompliant Code** |
| --- |
| If the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected. |
| #include <iostream>    void f() {  int i, j;  std::cin >> i >> j;  // ...  } |

| **Compliant Code** |
| --- |
| Exceptions are enabled so that any conversion failure results in an exception being thrown. |
| #include <iostream>    void f() {  int i, j;    std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try {  std::cin >> i >> j;  // ...  } catch (std::istream::failure &E) {  // Handle error  }  } |

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

| **Principles(s):** It is rare for a violation of this rule to result in a security vulnerability unless it occurs in security-sensitive code. However, violations of this rule can easily result in lost or misinterpreted data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ -ERR62 | -- |
| Clang | 3.9 | Cert-err34-c | Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C |
| CodeSonar | 7.3p0 | BADFUNC.ATOF  BADFUNC.ATOI  BADFUNC.ATOL  BADFUNC.ATOLL | Use of atof  Use of atoi  Use of atol  Use of atoll |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-ERR62-a | The library functions atof, atoi, and atol from library stdlib.h shall not be used |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-009-CPP] | Never hard code sensitive information. |

| **Noncompliant Code** |
| --- |
| It passes the authentication code to this function as a string literal. The authentication code exists in the program's binary executable and can be easily discovered. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  if (!authenticate("correct code")) {  printf("Authentication error\n");  return -1;  }    printf("Authentication successful\n");  // ...Work with system...  return 0;  } |

| **Compliant Code** |
| --- |
| Requires the user to supply the authentication code, and securely erases it when done. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  #define CODE\_LEN 50  char code[CODE\_LEN];  printf("Please enter your authentication code:\n");  fgets(code, sizeof(code), stdin);  int flag = authenticate(code);  memset\_s(code, sizeof(code), 0, sizeof(code));  if (!flag) {  printf("Access denied\n");  return -1;  }  printf("Access granted\n");  // ...Work with system...  return 0;  } |

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

| **Principles(s):** Hard coding sensitive information exposes that information to attackers. The severity of this rule can vary depending on the kind of information that is disclosed. Frequently, the information disclosed is password or key information, which can lead to remote exploitation. Consequently, a high severity rating is given but may be adjusted downwards according to the nature of the sensitive data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Hardcoded Authentication  Hardcoded DNS Name  Hardcoded Crypto Key  Hardcoded Crypto Salt  Hardcoded Seed in PRNG |
| Parasoft C/C++ test | 2022.2 | CERT\_C-MSC41-a | Do not hard code string literals |
| PC-lint Plus | 1.4 | 2460 | Assistance provided: reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate |
| Polyspace Bug Finder | R2023a | CERT C: Rule MSC41-C | Checks for hard coded sensitive data (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-010-CPP] | Do not use floating-point variables as loop counters. |

| **Noncompliant Code** |
| --- |
| A floating-point variable is used as a loop counter. The decimal number 0.1 is a repeating fraction in binary and cannot be exactly represented as a binary floating-point number. |
| void func(void) {  for (float x = 0.1f; x <= 1.0f; x += 0.1f) {  /\* Loop may iterate 9 or 10 times \*/  }  } |

| **Compliant Code** |
| --- |
| The loop counter is an integer from which the floating-point value is derived |
| #include <stddef.h>    void func(void) {  for (size\_t count = 1; count <= 10; ++count) {  float x = count / 10.0f;  /\* Loop iterates exactly 10 times \*/  }  } |

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

| **Principles(s):** The use of floating-point variables as loop counters can result in unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-flp30-c | Checked by clang-tidy |
| CodeSonar | 7.3p0 | LANG.STRUCT.LOOP.FPC | Float-typed loop counter |
| PC- lint Plus | 1.4 | 9009 | Fully supported |
| RuleChecker | 22.04 | For-loop-float | Fully checked |

### 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.

The best thing to do is to verify and test during the pre-production phase. Before starting production, this company needs to ensure that there are as few vulnerabilities as possible and as secure as possible. This makes the production phase much easier to manage if the company is starting out with less vulnerabilities. During the production phase, the top priority is the Transition and health check part of the phase. We want to have to deal with the least number of responses possible by testing the security of the system ourselves. Both of these phases work together to make a more secure system.

### 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 | Unlikely | Medium | Medium | 3 |
| STD-002-CPP | High | Probable | High | Medium | 2 |
| STD-003-CPP | Low | Likely | Low | Medium | 2 |
| STD-004-CPP | Low | Probable | High | Low | 3 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | Medium | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-009-CPP | High | Probable | Medium | High | 1 |
| STD-010-CPP | Low | Probable | Low | 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 | This is used for data that is never going to be moved. Think of data that needs to be secured like credit card information or passwords. |
| Encryption at flight | This is used for data that is going to be moving. Think of data that needs to be secured while on the move, such as online bank transactions or sensitive emails. |
| Encryption in use | This is used for data that is currently accessed and used. A good example is the use of a database. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Usernames and passwords to access certain information. |
| Authorization | The level of access a user is authorized to read/write, what files they are allowed to access, and any changes to the database with higher levels of authorization. |
| Accounting | Accounts that people use to login with. |

**\***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 | 03/19/2023 | Added coding standards | Tyler Pease |  |
| 3.0 | 04/09/2023 | Added risk assessments, automations, and policies | Tyler Pease |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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