**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_weqflbmuxjys)

[Purpose 2](#_dc3slpin2upo)

[Scope 2](#_mm2mx0whdu3)

[Module Three Milestone 2](#_glnjic43fimp)

[Ten Core Security Principles 2](#_3vbv5qu4xd4o)

[C/C++ Ten Coding Standards 3](#_g1i3h71s0xx9)

[Coding Standard 1 4](#_t7nsgfoookzi)

[Coding Standard 2 5](#_ahcugr1fs39e)

[Coding Standard 3 6](#_u2en2sq618br)

[Coding Standard 4 7](#_8d50t2m9li5r)

[Coding Standard 5 8](#_i3mt5uf149ci)

[Coding Standard 6 9](#_e4k1o9gd1iv9)

[Coding Standard 7 10](#_g2i4gks9yh82)

[Coding Standard 8 11](#_uh21009svw4d)

[Coding Standard 9 13](#_9ns14vs4n8w)

[Coding Standard 10 14](#_ugo8lmq7eqov)

[Defense-in-Depth Illustration 15](#_acf7im40ask0)

[Project One 15](#_bgzzv77v99do)

[1.](#_ucvrpehstmlh) Revise the C/C++ Standards 15

[2.](#_6w7ftof4w48n) Risk Assessment 15

[3.](#_hc4squwckvo6) Automated Detection 15

[4.](#_ggu71w69ou40) Automation 15

[5.](#_q2ghwbvmjqq1) Summary of Risk Assessments 16

[6.](#_m6ze0hw7736u) Create Policies for Encryption and Triple A 16

[7.](#_udqs3hld1cnt) Map the Principles 17

[Audit Controls and Management 18](#_flgfg1b3c0o3)

[Enforcement 18](#_vgd9h316e9y3)

[Exceptions Process 18](#_jdl5fuq2o3er)

[Distribution 19](#_omhprrv9xcy8)

[Policy Change Control 19](#_g99qek7l93v)

[Policy Version History 19](#_tlogggx596ub)

[Appendix A Lookups 19](#_wjlw4zzer4vm)

[Approved C/C++ Language Acronyms 19](#_xzglecdemfvj)

## 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 all input obtained from a user is removed of any contents that could potentially cause crashes, buffer overflows, SQL injections, or other types of attacks. Validating the input involves assuming the input is an attack, and using code to either reject the input, or remove portions of the input to make it safe to run. |
| 1. Heed Compiler Warnings | You should always handle warnings that the compiler gives you, as it can catch potential vulnerabilities that you would have otherwise missed. The compiler is automatically checking for certain types of problems, and you should handle any warnings it gives to you. |
| 1. Architect and Design for Security Policies | Design your code with security in mind from the start, such as checking permission levels and other core principles. Designing like this early makes it easier to work with the code in the future. |
| 1. Keep It Simple | Avoid redundant parts of code, and make sure your code is readable and understandable. Remove all ‘dead code’ such as unused calculations, to reduce potential attacks. |
| 1. Default Deny | Deny all requests by default, and only allow them if specifically needed. The default case for a permission check would be to deny the request. |
| 1. Adhere to the Principle of Least Privilege | Only grant the minimum privilege level required to do the task, and no more. The less things people have access to, the less potential attacks there are and less work figuring out who did something. |
| 1. Sanitize Data Sent to Other Systems | Ensure all input and output coming from an API or other system is sanitized, similarly to principle #1. Assume the data is malicious or malformed, and perform checks to make sure it is valid. |
| 1. Practice Defense in Depth | Use multiple overlapping methods of security to make it more difficult for attackers to breach your system. Adding input validation, a RBAC system, and other validation methods can all help make it harder to breach. |
| 1. Use Effective Quality Assurance Techniques | Go through code reviews, and run code testing tools like static and non static testing to check for vulnerabilities, and to stay on top of potential threats. Maven has a static testing tool, and there is also Clang or Cppcheck. |
| 1. Adopt a Secure Coding Standard | Find a method that works and stick to it. Follow protocols and procedures when writing code, and especially for code that handles sensitive information or operations. |

### 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 declare or define a reserved identifier <https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL51-CPP.+Do+not+declare+or+define+a+reserved+identifier> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a user-defined literal operator"" x is declared. However, literal suffix identifiers are required to start with an underscore; literal suffixes without the underscore prefix are reserved for future library implementations. |
| #include <cstddef>    unsigned int operator"" x(const char \*, std::size\_t); |

| **Compliant Code** |
| --- |
| In this compliant solution, the user-defined literal is named operator"" \_x, which is not a reserved identifier. |
| #include <cstddef>    unsigned int operator"" \_x(const char \*, std::size\_t); |

| **Principles(s):** 2. Heed Compiler Warnings, the compiler is likely able to determine when a collision like this would happen, and will warn you in the console that these errors are happening. Listening to compiler warnings and fixing them will solve this issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GDB online Debugger | web-2025 | [-Wliteral-suffix] | warning: literal operator suffixes not preceded by ‘\_’ are reserved for future standardization |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not depend on the order of evaluation for side effects  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP50-CPP.+Do+not+depend+on+the+order+of+evaluation+for+side+effects> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, i is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined. |
| void f(int i, const int \*b) {  int a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| These examples are independent of the order of evaluation of the operands and can each be interpreted in only one way. |
| void f(int i, const int \*b) {  ++i;  int a = i + b[i];  // ...  } |

| **Principles(s):** 4. Keep It Simple, if the code cannot be easily understood, let alone by the computer, then you are making some mistakes. The code should flow like english, even math operations, avoid nuanced operations like ++i, hard to understand and can cause problems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.16.0 | unknownEvaluationOrder | Expression ‘i+b[++i]’ depends on order of evaluation of side effects |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

| **Principles(s):** 1. Validate Input Data, even if the string is null, any operations that take in strings should expect null values, and find ways to work with them or return an error if they do. 2. Heed Compiler Warnings, oftentimes the compiler can warn you about null strings or uninstantiated values, so be sure to look for these. 10. Adopt a Secure Coding Standard, look out for uninstantiated strings and know how to avoid them all together with coding standards. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL injection <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection> |

| **Noncompliant Code** |
| --- |
| The JDBC library provides an API for building SQL commands that sanitize untrusted data. The java.sql.PreparedStatement class properly escapes input strings, preventing SQL injection when used correctly. This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement. |
| String pwd = hashPassword(password);  String sqlString = "select \* from db\_user where username=" +  username + " and password =" + pwd;  PreparedStatement stmt = connection.prepareStatement(sqlString);    ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name. |
| String pwd = hashPassword(password);    // Validate username length  if (username.length() > 8) {  // Handle error  }    String sqlString =  "select \* from db\_user where username=? and password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  } |

| **Principles(s):** 7. Sanitize Data Sent to Other Systems, database requests are sent to the server, so be sure the request isn’t malicious, which can be through prepared statements for MySQL. 10. Adopt a Secure Coding Standard, knowing ahead of time that SQL injection attacks exist, plan ahead and learn how to avoid them and best common practices used in applications that also avoid them. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two memory allocations are performed within the same expression. Because the memory allocations are passed as arguments to a function call, an exception thrown as a result of one of the calls to new could result in a memory leak. |
| struct A { /\* ... \*/ };  struct B { /\* ... \*/ };    void g(A \*, B \*);  void f() {  g(new A, new B);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a std::unique\_ptr is used to manage the resources for the A and B objects with RAII. In the situation described by the noncompliant code example, B throwing an exception would still result in the destruction and deallocation of the A object when then std::unique\_ptr<A> was destroyed. |
| #include <memory>    struct A { /\* ... \*/ };  struct B { /\* ... \*/ };    void g(std::unique\_ptr<A> a, std::unique\_ptr<B> b);  void f() {  g(std::make\_unique<A>(), std::make\_unique<B>());  } |

| **Principles(s):** 2. Heed Compiler Warnings, some of these issues can be resolved by reading compiler warnings, they can sometimes determine potential memory allocation errors or leaks, and should not be ignored. 10. Adopt a Secure Coding Standard, learn when potential memory leaks could occur, or adopt a secure way of handling the creation and destruction of data objects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression <https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression> |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

| **Principles(s):** 10. Adopt a Secure Coding Standard, adopting a standard for assertions of this type would help remove this situation. Protocols and procedures for writing assertion statements can be written, and should be followed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

| **Principles(s):** 1. Validate Input Data, input data that could cause unknown crashes should be handled and verified before being used in calculations. 2. Heed Compiler Warnings, sometimes the compiler will tell you when an exception type is unhandled, and this should not be ignored. 10. Adopt a Secure Coding Standard, following procedures for handling errors in try and catch blocks would help make this more consistent. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Miscellaneous** | [STD-008-CPP] | Detect and remove unused values <https://wiki.sei.cmu.edu/confluence/display/c/MSC13-C.+Detect+and+remove+unused+values> |

| **Noncompliant Code** |
| --- |
| In this example, p2 is assigned the value returned by bar(), but that value is never used. Note this example assumes that foo() and bar() return valid pointers (see DCL30-C. Declare objects with appropriate storage durations). |
| int \*p1;  int \*p2;  p1 = foo();  p2 = bar();    if (baz()) {  return p1;  }  else {  p2 = p1;  }  return p2; |

| **Compliant Code** |
| --- |
| This example can be corrected in many different ways, depending on the intent of the programmer. In this compliant solution, p2 is found to be extraneous. The calls to bar() and baz() can be removed if they do not produce any side effects. |
| int \*p1 = foo();    /\* Removable if bar() does not produce any side effects \*/  (void)bar();    /\* Removable if baz() does not produce any side effects \*/  (void)baz();  return p1; |

| **Principles(s):** 2. Heed Compiler Warnings, the compiler will tell you if any value is unused, and they should be removed when possible to not waste any resources. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.16.0 | unusedFunction | The function ‘f’ is never used. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Output** | [STD-009-CPP] | Close files when they are no longer needed <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not 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;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::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();  } |

| **Principles(s):** 10. Adopt a Secure Coding Standard, a procedure should be followed when dealing with opening and closing files, to ensure that memory leaks do not occur, and no other security vulnerabilities can happen due to have the file open or loaded in memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| none | - | No tool I tried was able to automatically discover the noncompliant code | Tried Clang 6.0.0, CppCheck 2.16.0, Visual Studio Code, GDB Online Debugger, and others |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | [STD-010-CPP] | Do not read uninitialized memory <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory> |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the object is initialized prior to printing its value. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

| **Principles(s):** 1. Validate Input Data, the data should be validated to make sure this cannot happen or cause an unexpected crash. 2. Heed Compiler Warnings, sometimes the compiler will tell you when you may be reading uninitialized memory, and this should not be ignored. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang C++ | 6.0.0 online | [-Wuninitialized] | warning: variable 'i' is uninitialized when used here |

### 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.

To modify the current DevOps process to turn it into DevSecOps, various security measures need to be integrated along each step of the current process. This diagram shows how in the “Asses and plan” and “Design” stage, there is responding to new threats and security driven design. In the “Verify and Test” stage, there is vulnerability scanning, and security testing. On the production side, it transitions to health checks and monitoring, which involve penetration testing and log collection. Lastly, in the “Maintain and Stabilize” stage, a security baseline is established, and returned to after an attack or compromise. Implementing these security changes into the DevOps process will move it towards DevSecOps. The cycle of development is a continuous loop, and with DevSecOps in the middle it makes it easier to maintain system integrity and overall security.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Unlikely | Medium | High | 3 |
| STD-003-CPP | High | Common | Medium | High | 3 |
| STD-004-CPP | Extreme | Very Likely | High | Highest | 5 |
| STD-005-CPP | High | Unlikely | High | Medium | 3 |
| STD-006-CPP | Medium | Unlikely | Medium | Low | 2 |
| STD-007-CPP | High | Likely | High | High | 4 |
| STD-008-CPP | Low | Very High | Medium | Medium | 1 |
| STD-009-CPP | Medium | Unlikely | Medium | Medium | 2 |
| STD-010-CPP | Very High | Unlikely | High | High | 4 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to encrypting data that is in storage. This would include things like hard-drives, cloud storage, photos, and any other sort of data that isn’t actively being used. The policy applies because the system eventually stores user information outside of RAM, and this data should be protected from attackers. |
| Encryption in flight | Encryption in flight refers to encrypting data that is crossing between devices or connections. This would include things such as sending messages, emails, or other sorts of requests or packets between devices. The policy applies because the system handles sending and receiving ‘data in flight’, and this data should be safe from attackers with the security policy. |
| Encryption in use | Encryption in use refers to data that is actively loaded into a computer's RAM, or things that are actively being edited or worked on. This would include the variables a system uses to operate, or a document that is being edited which was loaded into memory. The policy applies because the system handles user data and other important information, which could potentially be exposed while un-encrypted if an error or leak occurs. |

| 1. **Triple-A Framework** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to verifying a user’s identity when they log-in with either a token, password, face-id, 2fa, etc, or other authentication method. The policy applies because it is crucial to confirm a request is from the correct user. |
| Authorization | Authorization refers to the privileges a user has or level of access, which determines what actions and operations they can make. This includes which files they can access, what functionalities they can run, and what their priority is over other users. The policy applies because it is crucial to only allow a user access to the minimum systems necessary to accomplish their goal, the principle of least privilege applies here. |
| Accounting | Accounting refers to keeping a log of user activity or changes to parts of the system, so that a ‘history’ can be made and traced back to a time or user. The policy applies because it is important to be able to track down changes to a person and time frame, so that things can be ‘rolled back’ or people can be held accountable for past actions. |

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 04/04/2025 | Added Coding Standards | Michael Turco |  |
| 1.2 | 04/10/2025 | Finished all sections in document | Michael Turco |  |
| 1.3 | 04/17/2025 | Updated Triple-A policies, minor touch ups | Michael Turco |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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