**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 | Depending on the type of database used, a malicious user may try to input data in order to gain access to data, information, and/or permissions they shouldn’t have. Validating the input of a user is essential in preventing this type of attack. Validating can be done by ensuring the user has entered the correct variable type, syntax, characters, and/or is within a set range, otherwise the exception should be caught. |
| 1. Heed Compiler Warnings | Warnings should not go ignored, no matter how mundane they may initially seem. An attacker will find any way they can to gain unauthorized access to a program, and these warnings show a potential way how. Users should use both dynamic and static analysis tools to catch any unwanted security concerns. |
| 1. Architect and Design for Security Policies | Instead of simply finding security flaws in a program and fixing them, security should receive a top level overview before and during the creation of the program. By designing an architecture of security, higher level security principles can be easily incorporated. |
| 1. Keep It Simple | The more complex the system, the more prone to errors the system will be. Keeping the design simple makes security flaws easier to stay on top of. |
| 1. Default Deny | By default, access should be denied within a program. Access should only be granted to users with set permissions. |
| 1. Adhere to the Principle of Least Privilege | Permissions should only be given to users who need them in order to complete their tasks. A user with permissions they don’t need could be a liability. |
| 1. Sanitize Data Sent to Other Systems | Before data is sent to another system, the data should be looked over by the calling function to ensure the data is sanitized and won’t cause problems. This principle is necessary when input validation doesn’t catch an error and works as another line of defense from malicious input. |
| 1. Practice Defense in Depth | Defense in Depth uses multiple layers of security to ensure the entire system isn’t compromised. These layers include the physical, administrative and technical, each with their own sub-layers. Physical includes the physical location of where the system is (such as the databases and servers), and should be protected by physical means (security guards and cameras, for example). Administrative includes the rules and standards for the users, such as having employees change their passwords every week. Technical refers to the software itself. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance ensures that if a program is deemed finished, that finished program is secured from all types of threats. These techniques include fuzz testing, penetration testing, source code audits, and independent security reviews. |
| 1. Adopt a Secure Coding Standard | Depending on the language(s) used for the program, the secure coding standards of said language should always be followed to ensure there are no vulnerabilities based on the language itself. |

### 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-CLG | Implement abstract data types using opaque types  Rationalization: If an abstract data type is not made private, the contents of the unique data type become fully visible to the user. A malicious user could then potentially manipulate the data type. |

| **Noncompliant Code** |
| --- |
| The variable type of new\_type is fully visible to the user, which would allow them to access and directly manipulate the fields within the structure. |
| **struct** new\_type {  unsigned int type\_int;  **char** \*type\_char;  };  **typedef** **struct** new\_type unique\_var; |

| **Compliant Code** |
| --- |
| While not shown in the code, new\_type has been defined in a header file, meaning the data type is not visible to the user. |
| **struct new\_type;**  **typedef** **struct** new\_type unique\_var; |

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

| **Principles(s):**  6. Adhere to the Principle of Least Privilege: Users don’t need to view the new\_type structure, meaning they shouldn’t be able to.  8. Practice Defense in Depth: By putting the new\_type structure in another file that the user doesn’t have access to, another layer was created for better security.  3. Architect and Design for Security Policies: Putting the structure in a separate header file sets up the design to better enforce security. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-001-CLG | Low | Unlikely | High | Low |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL12 |  |
| LDRA tool suite | 9.7.1 | 104 D | Partially implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Value-returning functions must return a value from all exit paths  Rationalization: when calling a function with an expected return, some type of data value should always be returned, otherwise undefined behavior will occur. |

| **Noncompliant Code** |
| --- |
| If val is false, there is nothing to return, resulting in undefined behavior. |
| int return\_functino(bool val) {  if (val) {  return 1  }  } |

| **Compliant Code** |
| --- |
| Regardless of whether val is true or false, a value will be returned. |
| int return\_function(bool val) {  if (val) {  return 1  }  return 0  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: C++ coding standards state that a value-returning function should return something.  2. Heed Compiler Warnings: Most C++ compilers will present a warning that a function must return a value from all points. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-002-CPP | Medium | Probable | Medium | High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| CodeSonar | 7.3p0 | LANG.STRUCT.MRS | Missing return statement |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CLG | Do not attempt to modify string literals  Rationalization: modifying string literals will result in undefined behavior, as string literals are stored as read-only memory and aren’t meant to be modified. |

| **Noncompliant Code** |
| --- |
| The string literal is not meant to be changed. |
| char\* str = “string”;  str[0] = ‘b’; |

| **Compliant Code** |
| --- |
| A copy of the string literal, held in the space allocated by str, is used to modify the character arrays string. |
| char str[] = “string”;  str[0] = ‘b’; |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: C doesn’t allow string literals to be modified.  2. Heed Compiler Warnings: If a string literal has a value and that variable is changed later in the code, a warning should present itself. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-003-CLG | Low | Likely | Low | High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to “char \*” |
| RuleChecker | 23.04 | String-literal-modification | Partially checked |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-JAV | Prevent SQL injection  Rationalization: When users are given the ability to input data that will be read and/or written by an SQL database, the database may be at risk of SQL injection. A malicious user is able to write specific inputs that manipulate a query sent to a database, and this must be prevented. |

| **Noncompliant Code** |
| --- |
| The sql statement records the user input as is. This is a problem, as users can manipulate the string and change how the statement functions, giving them unauthorized access to the database, such as: random’ OR ‘1’=’1’. |
| Connection connection = getConnection();  Scanner input = new Scanner(System.in);  String username = input.nextLine();  String password = input.nextLine();  String query = "SELECT \* FROM users WHERE username = " + username + “AND password = “ + password;  Statement stmt = connection.createStatement();  ResultSet result = stmt.executeQuery(query); |

| **Compliant Code** |
| --- |
| By using a prepared statement, the statement knows that whatever is entered for the username, the statement will always be treated as a string, regardless of what the user inputs. |
| Connection connection = getConnection();  Scanner input = new Scanner(System.in);  String username = input.nextLine();  String password = input.nextLine();  String query = "SELECT \* FROM users WHERE username =? AND password =?;  PreparedStatement stmt = connection.prepareStatement(query);  stmt.setString(1, username);  stmt.setString(2, password);  ResultSet result = stmt.executeQuery(); |

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

| **Principles(s):**  1. ValidateInput Data: Prepared statements ensures that any input data will be treated as the intended value type.  7. Sanitize Data Sent to Other Systems: Prepared statements means SQL will treat all input as their intended types.  8. Practice Defense in Depth: Prepared statements offer a layer of defense between the user input and accessing the SQL database.  3. Architect and Design for Security Policies: Using prepared statements as a policy is considered good practice, meaning SQL programs should be designed with prepared statements in mind. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-004-JAV | High | Likely | Medium | Very High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | JAVA.IO.INJ.SQL | SQL injection (Java) |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory  Rationalization: If memory that has been deallocated is referenced, undefined behavior will occur in the form of dangling pointers, resulting in potential vulnerabilities. |

| **Noncompliant Code** |
| --- |
| test\_struct is referenced after it has been deleted. |
| **struct** TestStruct {  **void** test\_func();  };    **void** main(){    TestStruct test\_struct = new TestStruct();  delete test\_struct;  test\_struct->test\_func();  } |

| **Compliant Code** |
| --- |
| test\_struct is not referenced after it has been deleted. |
| **struct** TestStruct {  **void** test\_func();  };    **void** main(){    TestStruct test\_struct = new TestStruct();  test\_struct->test\_func();  delete test\_struct;  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: C++ coding standards recommend against accessing freed memory.  2. Heed Compiler Warnings: Most C++ compilers will present a warning if memory is accessed after it has been deleted in the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-005-CPP | High | Likely | Medium | Very High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.UAF | Use after free |
| Helix QAC | 2023.1 | C++4303, C++4304 |  |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CLG | Use a static assertion to test the value of a constant expression  Rationalization: Because constant expressions don’t change, they only need to be asserted at compile time. If the assert is done at run-time, it might not even be executed, and will likely be executed much later than when the assertion was needed. |

| **Noncompliant Code** |
| --- |
| The assertion is done within a function and uses a normal assert declaration, meaning it will only occur at runtime. |
| **struct** TestStruct {    unsigned int DATA;  };    **void** test\_func(){    assert(sizeof(struct TestStruct) == sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| By removing the assertion from a function and using static\_assert, the assertion will be done at compile time. |
| **struct** TestStruct {    unsigned int DATA;  };  static\_assert(sizeof(struct TestStruct) == sizeof(unsigned int), “Done at compile time”); |

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

| **Principles(s):**  9. Use Effective Quality Assurance Techniques: Assertions allow developers to verify a piece of code works as intended, ensuring that (in the case of static assertions) the program compiles as expected. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-006-CLG | Low | Unlikely | High | Low |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions  Rationalization: When an exception is not caught, termination of the program cannot be done in a controlled fashion, resulting in undefined behavior. |

| **Noncompliant Code** |
| --- |
| If test\_function throws an exception, the main() function has no way of catching it, resulting in the program being terminated and undefined behavior occuring. |
| void test\_function() {  throw\_function();  }  int main() {  test\_function();  } |

| **Compliant Code** |
| --- |
| The main() function tries to run the test\_function() function and catches all errors thrown by it. |
| void test\_function() {  throw\_function();  }  int main() {  try {  test\_function();  }  catch (…) {  std::cout << “Caught Exception” << std::endl;  }  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: C++ coding standards recommend that all exceptions are handled to prevent against memory errors.  8. Practice Defense in Depth: Exception handlers add a layer of security between errors occurring and the rest of the program, where errors will not crash the whole program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-007-CPP | Low | Probable | Medium | Low |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2023.1 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Correctness | STD-008-CPP | Range check element access  Rationalization: Strings vary in length, and std::string variables are able to change, meaning a character at index 5 may only temporarily exist. To prevent undefined behavior from an out\_of\_range error, the range of the string should be checked before an element is accessed. |

| **Noncompliant Code** |
| --- |
| No element exists at the 10th index of variable str, resulting in an out\_of\_range error. |
| int main() {  std::string str = “string”;  char str\_char = s[9]  } |

| **Compliant Code** |
| --- |
| The length of str is checked before it’s used to ensure that str[9] exists. |
| int main() {  std::string str = “string”;  char str\_char;  if (str.length() >= 10) {  str\_char = str[9];  }  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: C++ coding standards recommend that elements in an array, vector, etc. shouldn’t be accessed if they’re out of bounds. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-008-CPP | High | Unlikely | Medium | High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | assert\_failure |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Correctness | STD-009-CLG | Do not pass a non-null-terminated character sequence to a library function that expects a string  Rationalization: Many library functions expect that the strings they’re given are null terminated, so when they aren’t, memory can be accessed that is out of bounds. |

| **Noncompliant Code** |
| --- |
| The str variable can only hold the 5 elements given, meaning no null-terminated character is included which printf expects, resulting in an error. |
| int main() {  char str[5] = “12345”;  std::printf(“%s”, str);  } |

| **Compliant Code** |
| --- |
| The length of str is not specified, allowing it to include a null-terminated character. |
| int main() {  char str[] = “12345”;  std::printf(“%s”, str);  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: Most library functions have their own standards for how their functions should be used, and these standards should be follow to ensure a library won’t cause vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-009-CLG | High | Probable | Medium | Very High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | MISC.MEM.NTERM.CSTRING | Unterminated C String |
| Coverity | 2017.07 | STRING\_NULL | Fully implemented |
|  |  |  |  |
|  |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | STD-010-CLG | Do not read uninitialized memory  Rationalization: If memory is read without initializing it, it will try to read indeterminate values and undefined behavior will occur. |

| **Noncompliant Code** |
| --- |
| The variable num2 is never initialized, meaning adding num1 to num2 will result in undefined behavior. |
| int main() {  int num1 = 1;  int num2;  int sum = num1 + num2;  } |

| **Compliant Code** |
| --- |
| Both num1 and num2 are initialized and are read to make the sum variable. |
| int main() {  int num1 = 1;  int num2 = 2;  int sum = num1 + num2;  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: In order to use a C variable, the value must be initialized to not cause errors in the language.  2. Heed Compiler Warnings: Trying to utilize an uninitialized variable will usually cause a compiler warning. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| STD-010-CLG | High | Probable | Medium | Very High |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC | 4.3.5 |  | Can detect some violations of this rule when the -Wuninitialized flag is used |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Fully implemented |
|  |  |  |  |
|  |  |  |  |

### 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.



Through automation, coding standards can be complied with and enforced at a more accurate and efficient rate. Automation can be implemented in multiple areas of the DevSecOps process. It can be used in the build phase, primarily to test libraries and find their vulnerabilities. It can be used in the verify and test phase, where static and dynamic tests can be used to ensure the program runs as intended. It can be used in the transition and health check, ensuring the predefined configurations aren’t causing problems. It can be used in the monitor and detect phase, where it can send reports to developers about successes and failures of the system. In basically every part of the DevSecOps process, there are ways to automate the process and ensure coding standards are complied with and enforced.

### 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 | High | Low | 3 |
| STD-002-CPP | Medium | Probable | Medium | High | 2 |
| STD-003-CLG | Low | Likely | Low | High | 2 |
| STD-004-JAV | High | Likely | Medium | Very High | 1 |
| STD-005-CPP | High | Likely | Medium | Very High | 1 |
| STD-006-CLG | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | High | Unlikely | Medium | High | 2 |
| STD-009-CLG | High | Probable | Medium | Very High | 1 |
| STD-010-CLG | High | Probable | Medium | Very High | 1 |

### 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 | Encryption at rest encrypts data while it’s being statically held within a database. This policy ensures that if an unauthorized user gained access to a database, the data held within wouldn’t be readable. For example, confidential data such as passwords and credit card numbers are almost never held as plain text, meaning if any attackers gained access to the system, they would have no use for the encrypted data. |
| Encryption at flight | As data is being moved across systems, attackers are able to intercept these connections. If this happens, encryption at flight ensures that the attacker will have no use for the data, as the data will be encrypted. For example, if a private message is sent from one individual to another, the private message will be fully encrypted until it reaches the authorized individual who can access the message through a private key. |
| Encryption in use | Encryption in use encrypts data while it’s being used, ensuring that only authorized individuals can edit and view the data in plain text. This allows data to be protected throughout its entire life cycle. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is used to identify users and sure they are who they claim to be. For example, user logins are the most common form of this policy, where a user must enter the correct username and password which only they should know. This policy is useful when certain users are authorized for actions that not all users should have access too. |
| Authorization | Authorization ensures that a user can only accomplish their tasks based on the enforcement of policies. For example, a standard user may only be able to change information specific to them within a database. An administrator, however, can do the same while also being able to add new users to the database. Users having varying levels of access allows each user to have the ability to accomplish their needed tasks without giving them any further permission they shouldn’t have. |
| Accounting | Accounting ensures that activity within a system is behaving as expected, and any anomalies or security concerns are recorded. For example, a system keeping track of the actions of users, the files they’ve accessed, the time of the actions, and where the action came from can uncover security flaws if those actions ended up harming the system. If both authentication and authorization fail, accounting can show how those policies failed and what can be done in the future to ensure that they don’t. |

**\***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 | 07/12/2023 | Milestone 3 | Samuel Hanson |  |
| 1.1 | 07/26/2023 | Project 1 | Samuel Hanson |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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