**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 creates, deploys, or supports 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 | Data input by users is a major security threat vector, and should be validated prior to being accepted to insure against accidental or purposeful injection or overflow attempts. This may include limiting length, limiting value, or scanning input for characters that might indicate an attempted attack. |
| 1. Heed Compiler Warnings | The fact that a program compiles and runs doesn’t mean that it is fit for user interaction from a security standpoint. Many potential security flaws go unnoticed by compilers, so being flagged by one makes fixing compiler warnings all the more important. |
| 1. Architect and Design for Security Policies | Software should be constructed around security policies rather than vice versa. By first determining the security needs of a program and building it around them, programmers can make following security standards flow with the design of the application rather than against it. |
| 1. Keep It Simple | By keeping software simple, programmers help both the user, themselves, and future programmers who might work on the project. The more complex a program is, the more difficult it is to operate and repair, and the more likely a security flaw is to be written in unnoticed. |
| 1. Default Deny | Any privileged action in a program should start from a state of no-trust rather than default-trust. Allowed users and actions should be white-listed, rather than disallowed users and actions being black-listed. |
| 1. Adhere to the Principle of Least Privilege | Users should be given the least amount of privilege necessary to complete any task. This privilege should extend only to the parts of the program the user needs to interact with, and should only be given for as long as the user needs it. |
| 1. Sanitize Data Sent to Other Systems | In the same vein as input validation, information passed from an application to another system or application may be at risk of interception and alteration by attackers. Similar checks should be in place to insure against code injection. |
| 1. Practice Defense in Depth | Multiple layers of defense should be in place in the event one layer is compromised. Protecting possible attack vectors on multiple fronts creates a failsafe in the event accidental flaws are written in, or an attacker manages to breach a single layer |
| 1. Use Effective Quality Assurance Techniques | A codified testing and code review procedure should be a natural part of software development rather than an afterthought once production ends. A built-in, iterative QA structure ensures secure coding standards are in place throughout a project. |
| 1. Adopt a Secure Coding Standard | Having a set of rules in place regarding the programming language being used and the security needs of a project before any code is written ensures that all developers are on the same page and are writing mutually understandable uniformly secure code. |

## 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-DT] | Data should be explicitly-typed according to the purpose it is meant to serve. Input outside of a stated type should either be converted before processing, or throw an exception. This prevents accidental storage of inappropriate values and lessens the risk of code injection |

| **Noncompliant Code** |
| --- |
| A variable is defined as an unsigned char when negative numbers are necessary, user prompt doesn’t specify the need for a whole number. |
| unsigned char temp;  temp = 0;  std::cout << "Enter temperature: ";  std::cin >> temp; |

| **Compliant Code** |
| --- |
| Proper data type is initialized & requested from the user. |
| signed char temp;  temp = 0;  std::cout << "Enter temperature rounded to nearest degree: ";  std::cin >> temp; |

|  |
| --- |
| **Principles(s):** ValidateInput Data – Data typing is important when considering parameters for a function that a user might input. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-DV] | Variables with defined lengths should use only as many characters as necessary to preserve memory, and insure against unauthorized input from users. If a data limit is surpassed, the program should truncate it before acceptance to avoid overflow. |

| **Noncompliant Code** |
| --- |
| A defined-length variable is given a length too long for its purpose within the program, and doesn’t protect against buffer overflow in the event that limit is surpassed. |
| const std::string account\_number = "ACCT12345";  char user\_input[50];  std::cout << "Enter account number: ";  std::cin >> user\_input; |

| **Compliant Code** |
| --- |
| A variable is defined with a reasonable limit, and protects against buffer overflow. |
| const std::string account\_number = "ACCT12345";  char user\_input[20];  std::cout << "Enter a value: ";  std::cin.getline(user\_input, 20); |

|  |
| --- |
| **Principles(s):** Architect and Design for Security Policies – Like the previous standard regarding data types, this standard involves ensuring that a program is designed around security principles rather than the other way around. Having a set standard involving capping data at a reasonable value is a good preemptive standard to help ensure against buffer overflow. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-SC] | String input from users should be checked for the presence of attack indicators. Like numerical variables, strings are susceptible to buffer overflow and –notably – code injection attacks, which can be executed within string input fields if input isn’t sanitized first. |

| **Noncompliant Code** |
| --- |
| A string variable is defined with a given length, but without safeguards against buffer overflow. |
| const std::string user\_string = "correct string";  char user\_input[20];  std::cout << "Enter a value: ";  std::cin << user\_input; |

| **Compliant Code** |
| --- |
| A string variable is defined, and a method is put in place to truncate text exceeding the limit to protect against buffer overflow. |
| const std::string user\_string = "correct string";  char user\_input[20];  std::cout << "Enter a value: ";  std::cin.getline(user\_input, 20); |

|  |
| --- |
| **Principles(s):** Validate User Input – I think the previous “design for security policies” could apply here, but I see strings as being most user-provided, so it’s always important to check carefully when a string is input into a program. Failure to validate this could lead to an opening for SQL Injection. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-SQL] | All user input should be sanitized against the inclusion of keywords or phrases indicating a SQL Injection attack. A positive result should be sanitized of the potential attack or, preferably, thrown an exception with an explanation to the user. |

| **Noncompliant Code** |
| --- |
| A string input field will accept anything the user enters without regard for potential attack keywords. |
| Std::string user\_string = “ “;  Cout >> “please enter your string”;  Cin << user\_string;  Return user\_string; |

| **Compliant Code** |
| --- |
| After user input, the input variable is searched for a potential SQL injection attack, rejects the input, and throws an exception to the user |
| . . .  std::string threatStr = " or ";  Cout >> “please enter your string”;  Cin << user\_string;  try {  if (user\_string.find(threatStr) != std::string::npos) {  throw 99; } |

|  |
| --- |
| **Principles(s):** Validate User Input – Like string correctness, SQL injection requires that a user be able to sneak database queries into a program via their inputs. The most important method of weeding this out is to check user input for flags that might suggest injection attempts. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-MP] | Only initialized memory should be read, uninitialized memory should not be read, and the same position in memory should only be de-allocated once. Dynamic memory management in C++ makes it a fast and efficient language, but also leaves open possible attack vectors if memory isn’t allocated, de-allocated, and referenced properly. |

| **Noncompliant Code** |
| --- |
| A variable p is freed by the programmer and subsequently freed a second time |
| x = malloc(n \* sizeof(int));  free(x);  y = malloc(n \* sizeof(int));  free(x); |

| **Compliant Code** |
| --- |
| A variable p is freed a single time and no longer referenced unless it is re-allocated |
| x = malloc(n \* sizeof(int));  free(x);  y = malloc(n \* sizeof(int)); |

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard – Leaving hanging pointers or attempting to reference de-allocated memory is a flaw that is entirely in the hands of programmers. Developing an enforceable security standard that delineates how memory management should be handled stops these defects before they’re coded into a program. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-AST] | Assertions should be used throughout a project to provide runtime QA, but should not be used as input-validation devices. There may be call to remove assert statements once a project is ready for production which would inadvertently remove these security features. |

| **Noncompliant Code** |
| --- |
| An assert statement checks that a user’s input matches a set phrase. |
| Std::string user\_input = “ “;  Cout >> “enter a string: “;  Cin << user\_input;  Assert(user\_input) == “correct string”; |

| **Compliant Code** |
| --- |
| The same validation is provided by a try-catch block instead. Leaving other assert statements to perform QA functions. |
| Std::string user\_input = “ “;  Cout >> “enter a string: “;  Cin << user\_input;  Try {  If (user\_input != “correct string”) {  Throw stringexception; }  } catch(…) |

|  |
| --- |
| **Principles(s):** Use Effective Quality Assurance Techniques – Assertions are part of an effective C++ quality assurance framework. Letting programmers know how assertions should be used will result in more efficient unit testing. While not necessarily a prime security target, a solid testing program will allow developers to more quickly weed out possible exploits. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-EXC] | Take every opportunity to throw exceptions, but do not use shortcut functions to prematurely abort a program. This may leave temporary objects in an indeterminate state. In addition, each exception thrown should be caught and handled rather than simply thrown and ignored. |

| **Noncompliant Code** |
| --- |
| A function that causes an exception is ignored. |
| Int value1 = 0;  If (value1 != 1) {  Throw exeption;  } |

| **Compliant Code** |
| --- |
| A function creates an exception within a try-catch block, allowing the program to process it. |
| Int value1 = 0;  Try {  If (value1 != 1) {  Throw exception;  } catch(…) {  Cout >> “exception occurred”; } |

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings – Exceptions and compiler warnings aren’t the same thing, but building an effective framework for fixing broken or potentially broken code is essential to a security-forward workflow. Compiler warnings are the advance warnings about possibly defective code, and exceptions are notifications that a facet of a program has failed during runtime. They are two sides of the same coin in preventing flaws from making it into production code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input & Output Streams | [STD-008-IOS] | Any file opened to be written to or read from should be closed before the program terminates. |

| **Noncompliant Code** |
| --- |
| A file is opened, it is either written to or read from, and the program is terminated before the file is closed |
| std::fstream file(fileName);  if (!file.is\_open()) {  return;  }  std::terminate(); } |

| **Compliant Code** |
| --- |
| A file is opened, written to or read from, and is closed before the program is closed. |
| if (!file.is\_open()) {      return;    }    file.close();    if (file.fail()) //handle error then…    }    std::terminate(); |

|  |
| --- |
| **Principles(s):** Sanitize Data Sent to Other Systems – While not every input or output stream implies a network connection to another system, I feel like treating opened files on a system as if they’re connected to the internet is a good practice, as most systems will be. A program that opens and edits files should also close those files to prevent additional alterations or unauthorized reading on the same system or across networks. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Use of Smart Pointers | [STD-009-SPT] | Using smart pointers in the correct manner gets rid of many of the possible memory vulnerabilities present when using raw pointers. They should be used whenever it makes sense with |

| **Noncompliant Code** |
| --- |
| A raw pointer is used to point to a memory location and would need to be deleted manually, leaving open the possibility of error on the part of the programmer. |
| void UseRawPointer()  {  Bird\* rBird = new Bird("Robin");  // use object  delete rBird;  } |

| **Compliant Code** |
| --- |
| Here, a smart pointer is used for the same task, and will be deleted automatically. |
| void UseSmartPointer()  {  unique\_ptr<Bird> bird2(new Bird("Robin"));  // use object  } // object deleted automatically |

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard – much as with the memory management standard, leaving hanging pointers or attempting to reference de-allocated memory is a flaw that the proper use of smart pointers can mitigate to some extent. Developing an enforceable security standard that delineates how memory management should be handled stops these defects before they’re coded into a program. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Random Number Generation | [STD-010-RNG] | Random number generation is actually only pseudo-random, and depends on how a generator is seeded. By failing to seed, a RNG will produce the same numbers in sequence over and over, rendering the sequence discoverable by a user. |

| **Noncompliant Code** |
| --- |
| A random number generator is started without a seed value. This will produce the same sequence of numbers every time. |
| std::time\_t t;    std::mt19937 engine(std::time(&t));      for (int i = 0; i < 10; ++i) {      std::cout << engine() << ", ";    } |

| **Compliant Code** |
| --- |
| A random number generator is stated, and seeded with yet another random number generator, ensuring that a new sequence will be produced with each call. |
| std::random\_device dev;    std::mt19937 engine(dev());      for (int i = 0; i < 10; ++i) {      std::cout << engine() << ", ";    } |

|  |
| --- |
| **Principles(s):** Practice Defense in Depth – Many principles could apply to this standard, just as this principle applies to nearly all of the standards before it. Exploitation of an RNG system is a somewhat unlikely vector of attack and needs to be prioritized accordingly. At the same time, however, it cannot be ignored. It plays an important part as another layer in the overlapping framework of security measures that compose a defense in depth system. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.3 | CppCheck | Comprehensive standalone Static C++ testing tool |
| Visual Studio | 16.9.0 | Static Code Analysis | Part of the Visual Studio IDE, checks static C++ code |
| Visual Studio | 16.9.0 | Google Test | A unit testing plug-in for Visual Studio |
| CppUnit | 1.15.1 | CppUnit | A standalone unit tester based on the Java language’s JUnit |

## Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Automation



An effective security policy should be nearly automatic to the developers following it. Putting measures in place to ensure that the code being written follows security standards takes an additional load off of the mind of developers. The fewer conscious decisions they have to make, the more they can concentrate on building the correct functionality into their code. The policy and security standards themselves are a tool in their own right, but not necessarily a self-enforcing one. It is only when we approach quality control and coding tools that we can institute controls on production that will automatically test whether or not security standards are being met.

In the “verify and test” stage of the diagram above, programmers or dedicated testers put their code through pre-made unit tests that both test the functionality of the code and also help verify that it’s following security standards. Using a test structure for this rather than solely static code reviews both frees other developers up to devote more work to the project and cuts down on the amount of potential flaws missed due to human error. Of course, testing won’t always detect every case, but automation can solve the most tedious parts of QA and policy enforcement.

## Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-DT | Medium | Likely | Low | Medium | 1 |
| STD-002-DV | Medium | Likely | Low | High | 1 |
| STD-003-SC | High | Likely | Low | High | 1 |
| STD-004-SQL | High | Likely | Medium | High | 1 |
| STD-005-MP | Medium | Moderate | Medium | Medium | 2 |
| STD-006-AST | Low | Moderate | Low | Low | 3 |
| STD-007-EXC | Low | Moderate | Low | Low | 3 |
| STD-008-IOS | Medium | Likely | Low | Medium | 2 |
| STD-009-SPT | Low | Unlikely | Low | Low | 2 |
| STD-010-RNG | Low | Unlikely | Low | Low | 3 |

## Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Applies to the encryption of data during storage. Data isn’t only at risk when actively moving from one system or another or when being used. Attackers can gain access to stores of data, leading to the opportunity for mass leaks. Keeping data encrypted on disk ensures that even if databases are accessed, attackers won’t have access to easy-to-distribute plaintext data. |
| Encryption at flight | Applies to the encryption of data when moving between systems on a network. Data is particularly vulnerable when travelling from one system to another, and encrypting it before it is sent renders it either useless or much more cumbersome for attackers to use in the event it is intercepted. |
| Encryption in use | Applies to the encryption of data as it is being accessed as part of a program’s normal operation, but not during transmission as is handled in “encryption in flight”. Data is accessed from databases and input-output streams are opened from files regularly during a program’s use. Just as in network communication, the transmission of data from one part of an application to another represents a weak point at which data is more easily intercepted. Encrypting data as it undergoes these short hops mitigates this risk. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process by which an application’s users prove who they are to the system and gain access to it. This is most often done with a login system specific to individual users. This is highly important to any secure system. Without authentication, the system assumes that all users are identical and should be given blanket access to data. Once even a single bit of sensitive data is present in a program, controlling outside access is an absolute necessity. When new users need to be added to a system’s whitelist, it is through authentication that they gain access. A secure login system should be implemented, and credentials should be both regularly updated for active users and quickly revoked for terminated users |
| Authorization | Authorization is the process of determining what level of access is required by a user. This goes hand in hand with authentication, as a user must have a discrete identity in order for the system to differentiate between access permissions. The security principle “Adhere to the policy of least permission” is the gold-standard of authorization. By limiting each user to the least amount of access to a system’s files as possible, the opportunity for a user to abuse their permissions to use information in an unethical way isn’t even presented as a possibility. Incorporating least permission into the company’s authorization policy ensures that only those with legitimate cause to access data will have the ability to. |
| Accounting | Accounting involves a system tracking and auditing the actions of users and the resources they use while they interact with a system. It is a more active measure than authentication and authorization and may have direct effects on both. Accounting might consist of a series of checks monitoring what a user does when using an application, or checking their inputs for possible signs out unethical use. These accounting measures might determine how administrators choose to alter the permissions of one user or class of users in the future. Implementing a thorough accounting procedure will allow developers and administrators to decide on what proper access looks like and will warn them ahead of time if any potential red flags appear in the user-base. |

# 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 |
| --- | --- | --- | --- | --- |
| 1.0.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.0.5 | 03/20/2021 | First Draft (Mod 3) | Evan Baber |  |
| 1.1.0 | 04/07/2021 | Complete Sec. Policy | Evan Baber |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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