

Green Pace Secure Development Policy

Mitchell Ibarra

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 | All input from unknown and untrusted data sources should be properly validated, which can eliminate many software vulnerabilities. Validate suspicious external data sources such as command line arguments, network interfaces, environmental variables, and user controller files. |
| 1. Heed Compiler Warnings | Compile code using the compiler’s highest warning level available and eliminate warnings through modifying the code. |
| 1. Architect and Design for Security Policies | Keep security as one of the forefront considerations when creating and designing software architecture. Security best practices may require systems to be split up to ensure secure policies are applied in layers. |
| 1. Keep It Simple | Keeping the design simple and clean will help eliminate complexity and reduce the change of introducing potential security concerns. |
| 1. Default Deny | Deny permission by default and only escalate permissions only when required and conditions under which access should be elevated have been met. |
| 1. Adhere to the Principle of Least Privilege | Each process should only be executed with the least number of privileges required to complete the job. These elevated privileges should only be allowed for the least amount of time required to complete the job. |
| 1. Sanitize Data Sent to Other Systems | All data that is passed through complex subsystems shall be sanitized by the system sending the information. This is to prevent unintended function calls via inputs made through the command line, relational databases, and commercial off-the-shelf components. |
| 1. Practice Defense in Depth | Design systems through layered defense strategies, so that if one layer of defense should fail, another layer can potentially prevent a security flaw. This will limit security risk and help prevent exploitable vulnerability and/or the limit the consequences of a successful exploit. |
| 1. Use Effective Quality Assurance Techniques | High quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Independent security reviews can lead to more secure systems and external requires can help provide an unbiased perspective to bring to light ways to make more secure systems. |
| 1. Adopt a Secure Coding Standard | Apply secure coding best practices for each programming language and development platform. |

## C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Declare data types in as local a scope as possible and as close to the first use as possible. |

| **Noncompliant Code** |
| --- |
| Noncompliant code example demonstrates how an object is incorrectly declared within a for loop, increasing CPU cycles and taking up memory as the object Foo gets created 1000000 times. |
| // Inefficient implementation:  for (int i = 0; i < 1000000; ++i) {  Foo f; // My ctor and dtor get called 1000000 times each.  f.DoSomething(i);  } |

| **Compliant Code** |
| --- |
| Compliant code example shows more efficient declaration of variable. Note: Primitive types should be declared within the scopes of if, while, and for loops as they are taken care of efficiently by the compiler, while an object will have its constructor and destructor invoked every time the object goes in and out of scope. |
| Foo f; // My ctor and dtor get called once each.  for (int i = 0; i < 1000000; ++i) {  f.DoSomething(i);  } |

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

|  |
| --- |
| **Principles(s):** [Keep it Simple] By limiting the scope of variables we are keeping design of code simple and clean to eliminate complexity. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### 

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that numerical data values do not lose accuracy through arithmetic or other data conversions. Examples include implicit and explicit (using a cast) conversions. |

| **Noncompliant Code** |
| --- |
| Loss of data occurs through truncation in this example. This can occur when converting a value of a signed integer type to a value of a signed integer type with less precision. |
| #include <limits.h>    void func(void) {  signed long int s\_a = LONG\_MAX;  signed char sc = (signed char)s\_a; /\* Cast eliminates warning \*/  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Compliant code validates ranges when converting from a signed type to a signed type with less precision. |
| #include <limits.h>    void func(void) {  signed long int s\_a = LONG\_MAX;  signed char sc;  if ((s\_a < SCHAR\_MIN) || (s\_a > SCHAR\_MAX)) {  /\* Handle error \*/  } else {  sc = (signed char)s\_a; /\* Use cast to eliminate warning \*/  }  /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** [Validate Input Data] Ensuring that numerical data values do no lose accuracy reduces mathematical rounding errors that have been discovered to cause extreme failure in the worst case scenarios. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### Coding Standard 3

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

| **Noncompliant Code** |
| --- |
| Writing more than 12 characters for a single character array will result in a buffer overflow. While using std::ios\_base::width() may be helpful in limiting the number of characters allowed in bufOne, a truncated string can still be stored in bufOne, and the second char array can still be overflowed. |
| #include <iostream>    void f() {  char bufOne[12];  char bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| An easy way to counteract the potential buffer overflow is to use std::string instead of a bounded char array. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

|  |
| --- |
| **Principles(s):** [Validate Input Data] Ensuring that strings have sufficient space to prevent buffer overflow can prevent loss of data and/or undefined behavior that could cause harm of failure to systems. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards: Security | CppCheck is a static analysis tool to check C/C++ code for common security errors and coding standards. |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL injection attacks that could lead to serious vulnerabilities and/or leakage of sensitive information. |

| **Noncompliant Code** |
| --- |
| This example takes user input and directly uses it in an SQL query on a database. This can lead to SQL injection attacks that can leak sensitive information. |
| String sqlString = "SELECT \* FROM db\_user WHERE username = '"  + username +  "' AND password = '" + pwd + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString); |

| **Compliant Code** |
| --- |
| One way to counteract SQL injection attacks is to use ASP.net to create a command object and pass all parameters of the SQL query using a Parameter object. This prepares the statement to be executed all together as exact string values and not as an SQL command. |
| private void UseCommandObject(){  Dim strSQL as String sql  Dim SqlCommand As OleDbCommand  cmdSQL = New SQLDbCommand()  strSQL = "SELECT \* FROM tblUSER "  sql += " WHERE USERId = @UserId"  cmdSQL.CommandText = strSQL  cmdSQL.Parameters.Add(new SqlParameter("@UserId","Test"));  cmdSQL.Connection = new SqlConnection("Server=Localhost;Database=Northwind; +  Integrated Security=Yes")  cmdSQL.ExecuteNonQuery()  } |

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

|  |
| --- |
| **Principles(s):** [Sanitize Data Sent to Other Systems] Ensuring that SQL statements are sanitized and prepared before being sent off to query databases prevents potentially harmful SQL injections attacks that could cause damage to databases or exposure of personal and protected data. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards which include SQL prevention techniques. | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Protect against common memory access errors like buffer overflow and underflow or memory leaks. |

| **Noncompliant Code** |
| --- |
| This example shows how std::fill\_n()is used to fill a buffer with 10 instances of the value 0x42. The buffer has not allocated any space for the elements, which causes this to result in a buffer overflow. |
| #include <algorithm>  #include <vector>    void f() {  std::vector<int> v;  std::fill\_n(v.begin(), 10, 0x42);  } |

| **Compliant Code** |
| --- |
| In the compliant code, we see that initial capacity of the vector is sufficient before we attempt to fill the container. |
| #include <algorithm>  #include <vector>    void f() {  std::vector<int> v(10);  std::fill\_n(v.begin(), 10, 0x42);  } |

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

|  |
| --- |
| **Principles(s):** [Validate Input Data] Validating input data will ensure that memory access errors like buffer overflow and underflow are prevented. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are useful as a diagnostic tool for debugging and corrective software defects which could potentially result in vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant code only executes at runtime and is required to be placed in a function can executed. This indicates that the diagnostic only occurs at runtime if the code path containing this assertion is executed which drastically drops the usefulness and likelihood that errors are found. |
| #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** |
| --- |
| The compliant code uses static\_assert which always runs at compile time, meaning that there are no runtime costs in space or time. This also ensures that meaningful messages are presented in case of failure when compiling whereas catching runtime failures are much harder to catch. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

|  |
| --- |
| **Principles(s):** [Architect and Design for Security Policies] Test Driven Development ensures that software behavior is as intended and can handle unexpected inputs and handle errors. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.10.0 | Unit test automation framework | Google test automates C++ unit tests with user defined assertions, value parameterized, and type parameterized tests. |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions are a great way to force call code that recognizes an error condition and how to handle it. Unhandled exceptions stop program execution. This also ensures that error conditions are handled well and prevent security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant example, although implementing exceptions, is not as useful as it could be. Throwing a general exception does not tell a programmer what exception is expected to be caught and how to properly handle that situation. |
| private void doSomething() throws Exception {  //...  } |

| **Compliant Code** |
| --- |
| This code accurately describes what exception is expected from this method so that a programmer can properly handle this case if it occurs. |
| private void doSomething() throws IOException {  //...  } |

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

|  |
| --- |
| **Principles(s):** [Architect and Design for Security Policies] Handling exceptions is a way to ensure that errors are handled appropriately and so that applications can still continue safely after catching errors. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.10.0 | Unit test automation framework | Google test automates C++ unit tests with user defined assertions, value parameterized, and type parameterized tests. |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [STD-008-CPP] | Input/output as it relates to writing and reading from files involves first opening the file and then closing it. This prevents leaving kernel resources in use that should no longer be used and will lead to lower performance overhead that could lock up system resources and cause vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This noncompliant example incorrectly leaves the file resource open, which leaves the resource occupied on the system. This can also cause unintended behavior to a file if bytes are still in the output stream and not properly flushed out. |
| #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** |
| --- |
| This compliant code properly closes the file after it has been used. Therefore, freeing up the resources to be used elsewhere, and ensuring that all of the information that needs to be accessed from the file is out of the buffer stream. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

|  |
| --- |
| **Principles(s):** [Architect and Design for Security Policies] Ensuring that system resources are properly open and closed within scope of what is needed will make sure that system resources are not acquired for longer than needed and mitigate performance overhead. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Constructor Initialization | [STD-009-CPP] | The order in which members are initialized, including base class initialization, is determined by the declaration order of the class member variables or the base class specifier list. So, writing member initializers other than in canonical order can result in undefined behavior like reading uninitialized memory, which can lead to security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| Noncompliant code example demonstrates member declaration and then member initialization in the constructor. Because the initialization is out of order (non-canonical order), this leads to an unspecified value being stored into dependsOnSomeVal. |
| class C {  int dependsOnSomeVal;  int someVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| In this compliant code example, the initialization occurs in canonical order, avoiding unintended behavior. |
| class C {  int someVal;  int dependsOnSomeVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

|  |
| --- |
| **Principles(s):** [Keep It Simple] Ensuring constructor initialization in the correct order will prevent undefined behavior from reading uninitialized memory. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.4.1 | CERT Coding Standards | CppCheck is a static analysis tool to check C/C++ code for common errors and coding standards. |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | [STD-010-CPP] | Multiple threads invoking the same function can cause concurrency problems, which often result to abnormal behavior and can cause more serious vulnerabilities such as abnormal termination, DOS attack, and data integrity violations. |

| **Noncompliant Code** |
| --- |
| This noncompliant example uses the function strerror which returns a human-readable error string given an error number. This function is not required to avoid data races. For example, and implementation could write the error string into a static array and return a pointer to it, and that array might accessible and modifiable by other threads. |
| #include <errno.h>  #include <stdio.h>  #include <string.h>    void f(FILE \*fp) {  fpos\_t pos;  errno = 0;    if (0 != fgetpos(fp, &pos)) {  char \*errmsg = strerror(errno);  printf("Could not get the file position: %s\n", errmsg);  }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the strerror\_s function which has the same functionality, but guarantees thread-safety. |
| #define \_\_STDC\_WANT\_LIB\_EXT1\_\_ 1  #include <errno.h>  #include <stdio.h>  #include <string.h>    enum { BUFFERSIZE = 64 };  void f(FILE \*fp) {  fpos\_t pos;  errno = 0;    if (0 != fgetpos(fp, &pos)) {  char errmsg[BUFFERSIZE];  if (strerror\_s(errmsg, BUFFERSIZE, errno) != 0) {  /\* Handle error \*/  }  printf("Could not get the file position: %s\n", errmsg);  }  } |

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

|  |
| --- |
| **Principles(s):** [Architect and Design for Security Policies] Preventing common concurrency issues like deadlocks and race conditions will mitigate locking up of system resources and prevent abnormal terminate. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Google Test | 1.10.0 | Unit test automation framework | Google test automates C++ unit tests with user defined assertions, value parameterized, and type parameterized tests. |

## 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.

In the same way that security should be thought about early and often, automation for DevSecOps should be done early and often as well. Within a continuous integration environment, software development happens rapidly, and security needs to be automated and incorporated throughout to keep up from the very beginning. Static code analysis tools like CppCheck and test frameworks like Google Test can be used throughout development to enforce secure coding practices. In addition to this, CI tools like GitLab can be used to automate unit tests as well and check new code against these tests when building, to ensure proper standards are being met .

## Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Low | Low | Low | 3 |
| STD-002-CPP | Medium | Medium | Low | Medium | 2 |
| STD-003-CPP | Medium | Medium | Low | Medium | 2 |
| STD-004-CPP | High | Medium | Medium | Medium | 2 |
| STD-005-CPP | High | High | Medium | High | 1 |
| STD-006-CPP | Medium | Low | Low | Medium | 2 |
| STD-007-CPP | High | High | Low | High | 1 |
| STD-008-CPP | High | Medium | Low | Medium | 2 |
| STD-009-CPP | Medium | Low | Low | Medium | 2 |
| STD-0010-CPP | High | Low | High | 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 | Designed to prevent attacker from accessing unencrypted data by ensuring the data is encrypted when on a disk. This makes it difficult for an attacker to access data without the encryption key. |
| Encryption at flight | Encryption of data while being transmitted. Especially with billions of data transfers over networks every day, secure network connections and sending encrypted data ensures that attackers are less likely to be able to access secure data. |
| Encryption in use | Ensures encryption of data is happening while the data is being accessed. One example is with passwords and the use of hashing to encrypt the password while it is being used to gain access to something. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This is used to identify a user is who they say they are, typically with login credentials like username and password. Lately, increased authentication has come in the form of multi-factor authentication like providing codes sent to mobile devices that the user owns. |
| Authorization | This ensures that a user has the proper authority to access certain areas of a system. Certain users will have different roles and should only be allowed to access the minimum amount of a system that they need to perform their role. |
| Accounting | This is used to keep metrics on how many resources a certain user uses, as analysis can determine whether how many resources the system needs in order to keep a consistent uptime without crashing. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

## Map the Principles

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

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

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

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

# Audit Controls and Management

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

Evidence will include the following:

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

# Enforcement

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

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

# Exceptions Process

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

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

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

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

# Distribution

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

# Policy Change Control

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

# Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 04/17/2021 | Updated with Policies | Mitchell Ibarra | Mitchell Ibarra |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

References

A. (n.d.). SEI CERT C++ Coding Standard. Retrieved April 17, 2021, from <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

What is aaa? (n.d.). Retrieved April 17, 2021, from [https://www.arubanetworks.com/techdocs/ClearPass/6.7/Aruba\_DeployGd\_HTML/Content/802.1X%2](https://www.arubanetworks.com/techdocs/ClearPass/6.7/Aruba_DeployGd_HTML/Content/802.1X%252) 0Authentication/About\_AAA.htm#:~:text=AAA%20stands%20for%20authentication%2C%20authorizati on,necessary%20to%20bill%20for%20services.

Cppcheck. (n.d.). Retrieved April 17, 2021, from <http://cppcheck.sourceforge.net/#features>

Google. (n.d.). Google/googletest. Retrieved April 17, 2021, from https://github.com/google/googletest