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



Gregory Isajewicz

# 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 | Input validation involves the checking and sanitizing of user inputs prior to processing or passing input to another method or function. Adherence to this security principle will help to prevent SQL-injection, Cross-site scripting, and buffer overflow attacks. It is good practice that user input is not passed directly, but rather is validated first to ensure the format and criteria of the input is as expected to prevent both malicious attack, and user-error instances. Some techniques for input validation include whitelisting acceptable inputs if the input expected is specific, blacklisting particular inputs that may indicate an attempted breach of security, and implementing input processing methods that ensure proper escaping and comprehensive error handling. |
| 1. Heed Compiler Warnings | Heeding compiler warnings involves paying close attention to, and addressing the warning generated during code compilation. Compiler warnings can include potential problems such as type mismatches, deprecated functions, logical errors, and uninitialized or ambiguous variables. Ignoring these warnings can result in unexpected bugs in particular use cases, and present vulnerability into the code which can be exploited. By heeding compiler warnings, we can improve the reliability and security of our code. |
| 1. Architect and Design for Security Policies | The principle of architect and design for security policies involves taking a proactive approach to security in programming. System and software developers should consider security at the very beginning of designing a system’s architecture and implement security into each aspect of the code along the way. This includes conducting threat and vulnerability assessments to predict security issues and working a defense-in-depth strategy into the architecture and design of the software being developed. |
| 1. Keep It Simple | In order to Keep It Simple, a programmer should attempt to implement the simplest functional solution in an easy-to-understand manner. Overly complicated solutions, algorithms, and programs are often more difficult to comprehensively test and analyze for security threats. They can also prove difficult to understand by another developer who needs to maintain or test the code. The result of this can be subtle mistakes and vulnerabilities that are not easily traced, caught, and corrected. By keeping our code as simple as feasibly possible, we reduce the likelihood of bugs and improve the readability and maintainability of the code. |
| 1. Default Deny | The principle of Default Deny works alongside the principle of least privilege. It stipulates that access to a systems resources and functionality should (as the name suggests) be denied as a default state. This means that any access to the system must be explicitly given rather than implied or passively allowed. For example, if a program includes a database of names and a method to search that database for a particular name, a user’s access should be authenticated, and their authorization verified prior to accessing that database. Access control lists, user roles, and permissions assignment should be carefully managed in such a way that a user must be explicitly authorized for each point of access. |
| 1. Adhere to the Principle of Least Privilege | Similar to Default Deny, the principle of least privilege dictated that a used should only be granted the minimum necessary privilege and access to as system as is needed to perform their duty or role. By adhering to this principle, we can reduce the risk of misuse of privileges, and also lessen the severity of a potential breach. If all users are given complete access, an attacker need only gain access to a single user’s credentials to conduct a massive breach of security. However, if each user only has very limited and compartmentalized access, the malicious acquisition of a user’s credentials will only compromise that smaller section of the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems involves implementing validation and controls over outgoing data to ensure that it does not contain sensitive content, or information that could be used to exploit vulnerabilities within the system. Like input validation, sanitizing outgoing data involves ensuring proper escaping of special characters, white/blacklisting values, and enforcing particular formats of data. This principle too helps to prevent SQL and command injection, cross-site scripting, and other forms of attack. |
| 1. Practice Defense in Depth | Defense in depth (DiD) is a comprehensive security strategy wherein multiple layers of security are implemented at various levels and with overlapping areas of coverage. A DiD strategy includes several layers of intrusion detection/prevention, firewalls, and authentication so that if an attack manages to circumvent one protection, it may be stopped at another. The resulting redundancy provides a much more robust level of protection compared to a security strategy which relies on a single security method to safeguard itself entirely. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance (QA) techniques are essential to not only the security of a program, but also its overall reliability and functionality. QA can include things as basic as code reviews, wherein a programmer will manually read and review a code in search or logical flaws, vulnerabilities, or violations of security policy. Additional QA techniques involves automated testing, static and dynamic testing, and penetration testing in an attempt to identify vulnerabilities within software which can then be mitigated. QA testing is also useful for ensuring proper functionality of a program at every level. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard within an organization involves establishing a defined set of guidelines and practices that developers must follow when developing code. Having an adopted standard lays out clear expectations of code quality and practices, and ensures that security is consistent throughout a program, regardless of the different developers or teams working on different sections. A secure coding standard also helps to spread awareness of security practices and vulnerabilities by collating the knowledge and experience of many developers and security professionals. The downstream effect of this collective awareness is the ability to approach security in a comprehensive and proactive manner. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not define a C-style variadic function. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a c-style variadic function is defined using an ellipsis. This function adds a series of integers together, however the variadic function could result in undefined behavior if an argument of data type other than int is passed. |
| #include <cstdarg>  int add(int first, int second, …) {  int r = first + second;  va\_list va;  va\_start(va, second);  while(int v = va\_arg(va, int)) {  r += v:  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| The compliant solution makes use of std::enable\_if to ensure that any nonintegral argument value results in an ill-formed program. |
| #include <type\_traits>    template <typename Arg,  typename std::enable\_if<std::is\_integral<Arg>::value>::  type \*= nullptr>  int add(Arg f, Arg s) { f + s; }  template <typename Arg,  typename … Ts, typename std::enable\_if<std::enable\_if  <std::is\_integral<Arg>::value>::type\*= nullptr>  int add(Arg f, Ts… rest) {  return f + add(rest…);  } |

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

| **Principles(s):**  This standard upholds the principle of **Architect and Design for Security Policies,** and **Use Effective Quality Assurance Techniques** by ensuring the design is readable and unpredictable code is avoided which could otherwise result in security vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | Critical | 3: Verify |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.14.0 | Variadic-Function | Static analyzer – detects the use of variadic functions and potential issues of their use. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use offsetof() on valid types and members |

| **Noncompliant Code** |
| --- |
| In this code example, a non-standard type is defined as struct D{ …}. An object of type D is passed to offsetof() resulting in undefined behavior. |
| #include <cstddef>  struct D {  virtual void f() {}  int i;  };  void f() {  size\_t off = offsetof(D, i);  // …  } |

| **Compliant Code** |
| --- |
| To comply with this standard, the struct D{…} definition can be modified to make a standard class within D as shown in the code below. Now, instead of passing invalid type D to offsetof(), the valid type InnerStandardLayout is passed. |
| #include <cstddef>  struct D {  virtual void f() {}  struct InnerStandardLayout {  int i;  } inner;  };  void f() {  size\_t off = offsetof(D::InnerStandardLayout, i);  //…  } |

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

| **Principles(s):**  This standard upholds the principle of **Validate Input Data** and **Use Effective Quality Assurance Techniques** by ensuring that an argument of an invalid type is not passed. Improper handling here could result in undefined behavior and security vulnerability.  Also upheld, is the principle of **Adopt a Secure coding Standard.** By keeping a uniform standard, the maintainability and readability of the application is improved. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3: Verify |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.14.0 | invalid-offsetof | Static analysis of code alerts developer to invalid usage of offsetof() and potential issues that may arise from it. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Check the range for element access |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, no range check is implemented, and thus the get\_index call may be out of range of the string. |
| #include <string>  extern std::size\_t get\_index();    void f() {  std::string s(“01234567”);  s[get\_index()] = ‘1’;  } |

| **Compliant Code** |
| --- |
| This solution complies with the standard by implementing a try/catch block to handle an out\_of\_range exception. |
| #include <string>  #include <stdexcempt>    extern std::size\_t get\_index();    void f() {  std::string s(“01234567”);  try {  s.at(get\_index()) = ‘1’;  } catch (std::out\_of\_range &) {  //Handle Error  }  } |

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

| **Principles(s):**  This standard upholds the principle of **Head Compiler Warnings,** and **Architect and Design for Security** within the program by detecting and handling errors related to out-of-bounds access and buffer overflow. When accessing an element, the design of the application should include a try/catch block implemented to handle the event of an exception. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | Critical | 6: Prevent |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 19.0.0 | [alpha.unix.cstring.OutOfBounds](https://clang.llvm.org/docs/analyzer/checkers.html#alpha-unix-cstring-outofbounds-c) | Checks for out of bounds access in string functions and string literals. |
| CPPCheck | 2.14.0 | arrayIndexOutOfBounds | Detects an array index access the is out-of-bound or potential out-of-bounds. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| The code block below is vulnerable to SQL injection because the SQL query statement is formed by placing the user input directly into the query statement without proper sanitization or validation. |
| …  String pwd = hashpassword(password);  String sqlString =  “SELECT \* FROM db\_user WHER username = ‘” + username +  “ AND password = ‘” + pwd + “’”;  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);  if (!rs.next()) {  throw new SecurityException(“Username or Password incorrect”);  }  … |

| **Compliant Code** |
| --- |
| This compliant code adheres to the standard by implementing parameterization to validate the input and construct a prepared statement rather than passing the user input directly into the SQL query. It also validates the maximum length of the username. |
| …  String pwd = hashpassword(password);  //Validate the username length  if (username.length()> 8) {  //HANDLE THIS  }  String sqlString =  “SELECT \* FROM db\_user WHER username=? AND password=?”;  PreparedStatement stmt = connection.perparedstatement(sqlString);  stmt.setstring(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException(“Username or Password incorrect”);  } |

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

| **Principles(s):**  This standard upholds the principles of **Validate Input Data, Sanitize Data Sent to Other Systems,** and **Architect and Design for Security Policies.** The design of a system or application should always validate input taken into it, and be structured to use parameterized queries to prevent SQL injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | High | Low | Critical | 7: Detect |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 19.0.0 | [alpha.security.taint.TaintPropagation](https://clang.llvm.org/docs/analyzer/checkers.html#alpha-security-taint-taintpropagation-c-c) | This tool helps to identify potential security vulnerabilities where the attacker can inject malicious data to the program to execute an attack. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| In this example, an array of int is created using ::operator new[](std::size\_t) without checking the resulting allocation. The function is marked noexcept – thus assuming the function does not throw and exceptions. An exception resulting from the failure of the allocation of memory in the array creation could lead to the abnormal termination of the program. |
| #include <cstring>  void f(const int \*arrayt, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // …  delete [] copy;  } |

| **Compliant Code** |
| --- |
| To comply with the standard, a simple solution is to implement a check prior to referencing the pointer. In this case, if the pointer is null the function exits prior to the pointer call, thus preventing the error. |
| #include <cstring>  void f(const int \*arrayt, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // HANDLE  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // …  delet [] copy;  } |

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

| **Principles(s):**  This standard aligns with the principle of **Head Compiler Warnings**  and **Use Effective Quality Assurance Techniques** by reducing the risk of buffer overflow and memory corruption by properly detecting and handling allocation errors. It also applies to the principle (not mentioned above) of **Resource Management** by ensuring proper allocation and deallocation as well as resource availability to prevent memory leaks. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 6: Prevent  7: Detect |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.14.0 | memleak, nullPointer | Detects memory leaks and null pointer dereferences related to memory allocation. |
| Clang Static analyzer | 19.0.0 | [core.NullDereference](https://clang.llvm.org/docs/analyzer/checkers.html#core-nulldereference-c-c-objc) | Checks for dereference of null pointers |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression. |

| **Noncompliant Code** |
| --- |
| This noncompliant code calls the runtime assert() macro instead of static\_assert. This incurs runtime cost and can call abort(). In the example below, assert() is used on a memory-mapped structure that is essential for proper code behavior. Therefore, if assert() calls abort() the program may behave incorrectly or crash. |
| #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** |
| --- |
| Below, the assert() statement is replaced with static\_assert(), which will run during compilation and result in an informative diagnostic message is failure occurs. |
| // static\_assert(constant-expression, string-literal);  #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):** This standard upholds the principle of **Adopt a Secure Coding Standard,** and **Use Effective Quality Assurance Techniques** by preventing the improper use of asset() macro in a manner than may cause unexpected behavior or for the program to exit prematurely. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Medium | 2: Create  3: Verify |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.4 | S3346 | This tool can be used to detect incorrect use of assertions as well as other compile-time checks. |

#### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| The following code implements f() – containing throwing\_func() - and main() – containing f() - without any exception handling. This can result in the call to std::terminate() if an exception is thrown. |
| void throwing\_func() noexcept (false);  void f() {  throwing\_func();  }    int main () {  f();  } |

| **Compliant Code** |
| --- |
| This compliant code corrects the above code by incorporating a try/catch block to handle exceptions thrown by throwing\_func(). |
| void throwing\_func() noexcept (false);  void f() {  throwing\_func();  }    int main () {  try{  f();  } catch (…) {  // HANDLE ERROR  }  } |

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

| **Principles(s):** This standard upholds the principles of **Use Effective Quality Assurance Techniques, Head Compiler Warnings,** and **Architect and Design for Security Policies**. By predicting and handling all potential errors and exceptions, the developer prevents the risk of data leakage resulting from an exception and reduces the risk of the application exiting prematurely. Error handling of this sort should be incorporated in the initial design of the application. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | Critical | 1: Plan  9: Predict  8: Respond |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.14.0 | exceptionSafety | Detects potential problems with exception handling and ensures exceptions are caught appropriately. |
| Coverity SAST | 2024.3.1 | [UNCAUGHT\_EXCEPT](https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/webhelp-files/help_center_start.html) | Helps to identify uncaught exceptions and ensures proper handling. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | [STD-008-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below opens a file with fstream for some purpose, but does not close the file. By failing to close the file, the user risks the integrity of the file contents, resource leaks, and unnecessary resource consumption in memory. |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &filename) {  std::fstream file (filename);  if (!file.is\_open()) {  //HANDLE ERROR  return;  }  // …  std::terminate(); |

| **Compliant Code** |
| --- |
| in the compliant code below, the file opened with fstream is properly close, thus protecting the data and prevent leakage and resource cluttering. Additionally, a check is made to ensure the file has closed properly before terminate() is called. |
| #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):**  This standard upholds the principle of **Use Effective Quality Assurance Techniques** by ensuring that files are properly closed once open which prevents the unnecessary use of resources. Also applicable is **Adopt a Secure Coding Standard**; ensuring the proper closure of streams prevents memory leakage, resource cluttering, and other unpredicted errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | High | 3: Verify  6: Prevent |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 19.0.0 | [unix.Stream](https://clang.llvm.org/docs/analyzer/checkers.html#unix-stream-c) | Detects when an open steam has not been closed, along with other stream related issues. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-009-CPP] | Write constructor member initializers in the canonical order. |

| **Noncompliant Code** |
| --- |
| In the code block below the initializer list for C::C() attempts to initialize someVal before dependsOnSomeVal. Because this order does not follow the member initializer order canonically, attempting to read someVal results in the storage of some unspecified value into dependsOnSomeVal. |
| class C {  int dependsOnSomeVal;  int someVal;  public:  c(int val) : someVal(val), dependsOnSomeVal(someVal +1) {}  }; |

| **Compliant Code** |
| --- |
| The compliant code below corrects the mis-ordered declarations so that the dependency can be properly ordered into the initializer list. |
| 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):**  This standard upholds the principles of **Keep it Simple,** and **Adopt a Secure Coding Standard** by keeping a consistent and easily readable structure and organization of variables, and preventing unspecified value assignment. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | Low | 1: Plan |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 19.0.0 | [core.uninitialized.Assign](https://clang.llvm.org/docs/analyzer/checkers.html#core-uninitialized-assign-c)  [core.uninitialized.Branch](https://clang.llvm.org/docs/analyzer/checkers.html#core-uninitialized-branch-c) | These two checkers within Clang check for improper initialization of variables. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-010-CPP] | Use valid iterator ranges. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code example below, the first iterator does not precede the second. Since the for\_each() loop compares the incremented value of the first iterator to the second iterator on each pass, and the second iterator in this case will begin above the first (in most cases) the iterator will be incremented past-the-end element resulting in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| The compliant code block below corrects the iteration into a valid range such that that c.begin() will be incremented with each iteration until it reached c.end(). |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):**  This standard upholds the principles of **Head Compiler Warnings, Architect and Design for Security Policies,** and **Adopt a Secure Coding Standard** by ensuring the proper and consistent use of args in the for\_each() iterator. Failure to abide this standard could result in undefined behavior, memory error or corruption, of infinite iteration; ultimately resulting in a crash of the program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 1: Plan  6: Prevent  9: Predict |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang Static Analyzer | 19.0.0 | [alpha.cplusplus.IteratorRange](https://clang.llvm.org/docs/analyzer/checkers.html#alpha-cplusplus-iteratorrange-c) | This tool checks for iterators used outside their valid ranges. |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

The DevOps process will require some modification as it is transitioned into DevSecOps. Firstly, in the Assess and Plan stage of pre-production, an initial security and threat analysis should be conducted. The most likely type of threat faced should be identified and mitigation techniques should be brainstormed to address them. In the Design stage, the application should be designed in a way that adheres to the standards and policies laid out here, as well as any additional needs identified in the threat assessment conducted previously. Next, the Build phase begins where the plan and design are put into action. Throughout this phase, developers should be considering the policies in place, and in the event an aspect of the design is identified that does not adhere to the standard, that aspect should be cycled back for redesign. The Verify and Test phase will happen in series with the Build phase. The tools defined above: CPPCheck, Clang, Coverity, and SonarQube should be implemented throughout this phase in order to verify compliance and test for vulnerability.

Once the Build, Verify and Test are complete, the application moves into the production phase beginning with Transition and Health Check. Here, penetration testing can be conducted manually or with automation in an effort to identify and security vulnerabilities that have made it through. Log collection, alerting, and intrusion detection should be fully implemented and tested. These should be monitored on an ongoing basis in order to identify any attacks, breaches, or other issues as early as possible. If an attack is detected, immediate action should be taken in response to stop the intrusion. Services may be halted, or a rollback may be conducted if necessary.

Finally, in the Maintain and Stabilize phase, security should be measured against the baseline laid out in this document. Review may be conducted in order to make any adjustments to security policy as deemed necessary by the ever-evolving threat landscape. The ultimate goal of this phase is to return to a stable secure state.

### 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 | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Medium | Low | High | 3 |
| STD-003-CPP | High | High | Low | Critical | 6 |
| STD-004-CPP | Critical | High | Low | Critical | 7 |
| STD-005-CPP | High | Medium | Low | High | 6 |
| STD-006-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-007-CPP | High | High | Medium | Critical | 1 |
| STD-008-CPP | Medium | Medium | Low | High | 3 |
| STD-009-CPP | Low | Unlikely | Low | Low | 1 |
| STD-010-CPP | High | Medium | Low | 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 at rest | Encryption at rest refers to the cryptographic protection of data that is being stored on a database or hard drive by converting it into a format that cannot be easily read or understood if accessed without proper authorization and decryption. By ensuring proper encryption at rest, data is rendered inactionable so that, should a breach occur and the data is stolen, it can not be easily used. Encryption at rest applies only to data not actively being used by a program. |
| Encryption in flight | Encryption in flight refers to the encryption used to protect data as it is moving within a network. Data in motion (in flight) is especially vulnerable to exploitation. The Encryption in flight policy is an important aspect of protecting that data and is achieved through a Secure Sockets Layer(SSL) and Transport Layer Security(TLS) connection between server and database. |
| Encryption in use | Encryption in Use refers to the protection of data that is being actively processed or used in memory by a program or user through encryption. This can be accomplished through homomorphic encryption or a Trusted Execution Environment (TEE) – a segregated area of memory that is encrypted from the rest of the CPU. The Encryption in use policy makes the targeted data harder to reach and identify, and more difficult to exploit as the data itself can take an exhaustively long time to decrypt without a key. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The authentication policy requires that all users be validated using secure methods of identification to ensure that the user is in fact who they claim to be. One method of conducting this identification is multi-factor identification (MFA) in which a user must use multiple methods of identification to authenticate themselves. These may include biometrics, passwords, and tokens. By implementing robust authentication of users, developers prevent attacks in which a bad actor gains access by pretending to be someone with authorization. |
| Authorization | Authorization involves the assignment of permissions to determine what an authenticated user is allowed to do or access within a system. This is where the principle of least privilege is implemented – a user should only have the bare minimum privilege within a system. Authorization may be implemented with role-based access control in which the access of an authenticated user is determined based on the role assigned to that user within the system.  By limiting the authorization of each user in a system, developers reduce the risk of catastrophic damage done should an attacker gain access to an authenticated users account. |
| Accounting | The Accounting policy with the triple-A framework involves the implementation of logging mechanisms to track the actions and accesses on the system. User login events, new user creation, data/ file access, data modification, and resource actions are some of the things that should be logged for each user. This policy provides an audit trail for the monitoring of security and identification of breaches or potential attacks. Having robust logs can also aid response to attacks and enable developers to quickly stop attacks in progress. |

**\***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 |  |
| 1.1 | 05/20/2024 | Module 3 Milestone | Gregory Isajewicz |  |
| 1.2 | 06/12/2024 | Project Completion | Gregory Isajewicz |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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