**Green Pace Developer: Security Policy Guide**



# 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 | **Validate Input Data** refers to ensuring the input data entered by external sources, such as users, meets the predefined criteria of the program. This involves checking data for the expected format, type and constraint before allowing it to interact with the rest of the application. The intent is to prevent attackers from exploiting the system or accessing sensitive information. As such it also extends to APIs & involves notifying users if their input does not meet the expected requirements. |
| 1. Heed Compiler Warnings | **Heed Compiler Warnings** can often alert developers to potential issues in code that may not result in compiling errors but could lead to risk in unexpected behavior, performance or security vulnerability. It’s important to equally consider these warning for preventing further risks. |
| 1. Architect and Design for Security Policies | **Architect & Design for Security Policies** means integrating security considerations to every step of the software development process, including but not limited to planning, developing, testing, etc. This approach allows security to be integrated proactively rather than reactively. |
| 1. Keep It Simple | **Keep it simple** emphasizes the importance to minimize complexity to reduce potential vulnerabilities and make security easier to manage. Lower complexity means less time required for testing, reducing encountering errors and speeding up development. |
| 1. Default Deny | **Default deny** bases the security model on access decisions through permission rather than exclusion. This ensures unwanted traffic is denied by default. |
| 1. Adhere to the Principle of Least Privilege | **Adhere to the principle of least privilege** ensures that processes have only the minimal level of access and permissions needed to perform their tasks. Any elevation to permissions should only be granted for the least amount of time required to complete the privileged task. |
| 1. Sanitize Data Sent to Other Systems | **Sanitize Data sent to other System** involves cleaning, filtering and transforming data that may be harmful or used to exploit vulnerabilities in set systems. This may include removing special characters or validating format of inputs. The goal is to prevent injection attacks by external systems. |
| 1. Practice Defense in Depth | **Defense in Depth** is the practice of including multiple defensive strategies to protect data & systems and mitigate risk of exploitable vulnerability. Additional layers of defense can limit the damage in successful exploits. |
| 1. Use Effective Quality Assurance Techniques | **Use Effective Quality Assurance Techniques** ensure software is reliable, secure and meets any required standard before production. This process includes various testing such as penetration testing, security audits and other vulnerability assessment. |
| 1. Adopt a Secure Coding Standard | **Adopt a Secure Coding Standard** ensures that software is developed with security in mind and is tailored to the specific language and platform in use. This approach provides guidelines to best enforce best practices. |

*Source:* [*Top 10 Secure Coding Practices*](https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices)

### 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. |

*Source:* [*DCL50-CPP. Do not define a C-style variadic function*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function)

| **Noncompliant Code** |
| --- |
| We suggest avoiding the use of C-style variadic functions as they rely on generic argument types which makes them unsafe and prone to unexpected behavior. Variadic functions provide limited to no information about the arguments passed to them which allows for no type checking and makes it easy to accidentally pass incompatible types. Incorrect use of a variadic function can lead to abnormal program termination, unintended information disclosure, or execution of arbitrary code.  In the non-compliant example below, a series of integers are retrieved and printed to the console. Due to the lack of type safety, passing an argument of a different type may result in this undefined behavior. |
| #include <cstdarg>    void printValues(int num, ...) {  va\_list va;  va\_start(va, num);  for (int i = 0; i < num; ++i) {  int value = va\_arg(va, int);  std::cout << value << std::endl;  }  va\_end();  } |

| **Compliant Code** |
| --- |
| In the compliant code below, we avoid C-style variadic functions and replace them with an initializer list which ensures all elements are type int. This reduces the risk of incorrect types being passed such as with variadic function. |
| #include <initializer\_list>  void printValues(std::initializer\_list<int> list) {  for (int value : values) {  std::cout << value << std::endl;  }  } |

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

| **Principles(s):**  The principle that best applies to the standard of avoiding C-style variadic function is **‘Adopt a Secure Coding Standard’ (#10)**. This principle ensures that software is developed with security in mind, focusing on being cautious of the safety lacking functionality within specific languages. C-style variadic functions are often considered dangerous due to their vulnerabilities to buffer overflows and improper handling of arguments which is why teams are encouraged to use safer alternatives such as initializer lists. The **‘Heed Compiler Warnings’ (#2)** is also a valid principle, as compilers often provide warnings for these type of functions and developers should heed these warnings to prevent further security risk. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube C/C++ | 4.10 | FunctionEllipsis - [SonarRule](https://next.sonarqube.com/sonarqube/coding_rules?open=c%3AS5270&rule_key=c%3AS5270) | SonaQube uses static code analysis which can help flag security vulnerabilities, bugs and unsafe practices such as the use of variadic functions. It does this through ellipsis notation. |
| Polyspace Bug Finder | R2024a | CERT C++: DCL50-CPP | Polyspace Bug Finder detects runtime errors, code correctness and helps follow standards like CERT. It looks for dangerous code such as variadic functions by checking for function definition with ellipsis notation. |
| CPPcheck | 2.11 | FunctionEllipsis | CPPcheck is an open source static analysis tool which helps detect common C++ issues. Similarly, it looks for function definition with ellipsis notation. |
| Clang | 3.9 | Cert-dc150-cpp | Clang is a static analyzer which check for violations against CERT. It checks for varidiac functions through clang-tidy. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that operations on signed integers do not result in overflow |

*Source:* [INT32-C. Ensure that operations on signed integers do not result in overflow](https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow)

| **Noncompliant Code** |
| --- |
| We emphasize the importance of checking that arithmetic operations does not cause the program to overflow, as signed integers can lead to unexpected behavior. In C++, signed integer overflow is considered undefined which means the compiler does not have to handle it as predicted. Integer overflow can result to buffer overflows, potentially allowing attackers to execute arbitrary code.  The code below is an example of a function that does not follow our compliance as the addition may result in a value larger than the maximum value can hold. |
| void function(signed int a, signed int b) {  signed int sum = a + b;  } |

| **Compliant Code** |
| --- |
| Instead in this example, this follows compliant code as it checks the values of the integers before performing addition operation ensuring overflow cannot occur. Its encouraged that developers use INT\_MAX to check if the result will exceed or return the maximum value of an integer. |
| #include <limits.h>    void func(signed int a, signed int b) {  signed int sum;    if (((b > 0) && (a > (INT\_MAX - b))) ||        ((b < 0) && (a < (INT\_MIN - b)))) {     /\* Handle error \*/    } else {      sum = a + b;    }  } |

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

| **Principles(s):**  The principle that best applies to this standard is ‘**Validating Input data’ (#1).** With this principle, we want to ensure integers passed into the program’s arithmetic operations are within our expected ranges, preventing overflows during calculations. The **‘Use of effective Quality Assurance Techniques’** **(#9)** is also an applied principle as it focuses on ensuring code is secure with the use of testing, audits and assessments. Applying these quality techniques can catch these arithmetic issues before they overflow in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPcheck Premium | 24.9.0 | premium-cert-int32-c | CPPcheck through its premium version can focus on detecting issues such as signed 32-bit integer overflow. That said, it’s only partially implemented as it can only handle basic cases. |
| Polyspace Bug Finder | R2024a | CERT C: Rule INT32-C | Polyspace uses formal methods to partially cover checking for: Integer overflow, Tainted division operand and Tainted modulo operand. |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Parasoft uses data flow analysis to avoid signed integer overflows or underflow in constant expression in '+', '-', '\*' operators. Additionally, it checks for integer overflow or underflow in constant expression in '<<' operator. |
| TrustInSoft Analyzer | 1.38 | Signed\_overflow | TrustInSoft Analyzer through its formal methods identifies difficult to find vulnerabilities. It is exhaustively verified which checks for all possible scenarios & inputs. |

#### 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 the null terminator |

*Source:* [STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator](https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator)

| **Noncompliant Code** |
| --- |
| It is important to ensure that enough memory is allocated for storing strings including the null terminator. Failing to do so can result in a buffer overflow which is when data exceeds the allocated capacity. Attackers can exploit this vulnerability to execute arbitrary code with the privileges of the vulnerable process.  In the example below, the non-compliant code allocates space for four characters. However, since the user is not restricted to how much they can enter, it can lead to risk for a potential buffer overflow. |
| int main()  {  char str[5];  std::cin >> str;  std::cout << “You said: “ << str << std:endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Alternatively in this example, the application reads only up to the memory it was allocated which is 4 characters. This ensures no risk to a buffer overflow. It is encouraged developers provide proper memory allocation to ensure enough space of input data and null terminator while adding input-handling to prevent overflow. |
| int main()  {  char str[5];  std::cin.get(str, 5);  std::cout << "You said: " << str << std::endl;  return 0;  } |

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

| **Principles(s):**  The principle that applies to the standard of guaranteeing that storage for strings has sufficient space for character data and null terminator is **‘Validate Input Data’ (#1)**. This principle focuses on verifying that input such as strings meets the expected format before running through the program. This helps prevent overflows which can be a common vulnerability with a high severity effect. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| TrustInSoft Analyzer | 1.38 | Mem\_access | TrustInSoft Analyzer through its formal methods identifies difficult to find vulnerabilities in memory. It is exhaustively verified which checks for all possible scenarios & inputs. |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Coverity targets improper memory allocation for character arrays, ensuring that there is enough space for both the string content and the null terminator. |
| Astrée | 24.04 | Supported, but no explicit checker | Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Parasoft avoids accessing arrays out of bounds, overflow when writing to a buffer , overflows from tainted data  and unsafe string functions which may cause buffer overflows. |

#### Coding Standard 4

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

*Source:* [IDS00-J. Prevent SQL injection](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection)

| **Noncompliant Code** |
| --- |
| A key focus is to sanitize and validate input before passing it to system commands or other sensitive components. Input validation is crucial as it ensures data by the user meets the expected format and does not pass malicious content to the system. Failure to sanitize user input before processing or storing it can result in injection attacks.  This code below shows an example of non-compliant since it directly uses user input in the SQL query, allowing attackers to manipulate their input to possibly alter or access the data in the SQL database. |
| #include <string>  void executeSQL(const std::string& query) {  std::cout << "Executing query: " << query << std::endl;  }  int main() {  std::string username;  std::cout << "Enter username: ";  std::cin >> username;  std::string query = "SELECT \* FROM users WHERE username = '" + username + "';";  executeSQL(query);  return 0;  } |

| **Compliant Code** |
| --- |
| Instead, this code below follows compliant practices as it passes the user input into a prepared statements function, which safely handles data and automatically removes special characters ensuring that user input does not alter the query.  Its encouraged for teams to define this as a coding standard and ensure a good understanding of input validation and use of prepared statements. |
| int main() {  sqlite3\* db;  std::string username; std::cout << "Enter username: ";  std::cin >> username;  statements executePreparedSQL(db, username);  sqlite3\_close(db);  return 0;  } |

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

| **Principles(s):**  The principle that applies to the standard of preventing SQL injection is **‘Validate Input Data’ (#1)**. As mentioned, this principle focuses on verifying that input from untrusted sources meet the expected format before running it through the program. This includes proper validation such as types, format & value ranges. Helping prevent overflows which can be a common vulnerability with a high severity effect. Another applied principles is **‘Sanitize Data Sent to Other System’ (#7)** which includes removing specifically characters that may alter SQL queries causing unauthorized alteration to the database. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.9 | S2077 - [SonarRules](https://rules.sonarsource.com/java/RSPEC-2077/) | SonarQube through its static analysis can identify lines of code that may be potential vulnerable to SQL injection. Rule S2077 looks for the use of non-parameterized SQL queries. |
| Coverity | 7.5 | SQLI | Coveritiy uses SQLI rules to identify dangerous patterns and track data flow. It also suggest techniques to mitigate SQL injection risk. |
| Parasoft C/C++test | 2024.1 | CERT.IDS00.TDSQL | Parasoft scans for when user inputs are used directly with system commands without proper sanitization. |
| Fortify | 1.0 | SQL\_Injection\_\_Persistence  SQL\_Injection | Fortify through its static analysis identifies codes where user data is used in SQL queries allowing for a possible injection. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory |

*Source:* [*MEM50-CPP. Do not access freed memory*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

| **Noncompliant Code** |
| --- |
| This standard is to ensure that memory that is freed, is not accessed again preventing any use-after-free vulnerabilities. Accessing previously deallocated memory can result to abnormal program termination and denial-of-service attacks.  This is an example of non-compliant code that attempts to access memory after it has been freed. Resulting in risk of the program crashing or introducing a security vulnerability. |
| struct S {    void f();  };    void g() noexcept(false) {  S \*s = new S;  delete s;  s->f();  } |

| **Compliant Code** |
| --- |
| Instead, its encouraged developers set the pointer to *nullptr* after deleting the memory to ensure the code is compliant and to prevent accessing free memory. This sets a safeguard from a security vulnerability or an unexpected crash. Below is an example of this. |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;  delete s;  s = nullptr;  if (s != nullptr) {     s->f();  }  } |

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

| **Principles(s):**  The principle that applies to the standard of not accessing freed memory is **‘Adopt a secure coding standard’ (#10)**. Following a secure coding standard provides guidelines on safe memory practices which typically include promoting safer alternatives such as smart pointers. **‘Architect and Design for Security Policies’ (#3)** is also a valid principle as it encourages the use of smart pointers during the design phase to reduce the risk of creating a dangling pointer later on. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | Coverity can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM50-a | Parasoft can scan the code base to identified memory that has not been freed. The checker recommends setting pointers to nullptr to avoid accidental access. |
| Astrée | 22.10 | dangling\_pointer\_use | Astree can track the lifecycle of memory from allocation to deallocation to check if memory that has deallocated is accessed again. |
| Polyspace Bug Finder | R2024a | CERT C++: MEM50-CPP | Polyspace checks for pointers access out of bounds, deallocation of previously deallocated pointer and use of previously freed pointer. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assert() only for debugging purposes |

*Source:* [*MSC11-C. Incorporate diagnostic tests using assertions*](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions)

| **Noncompliant Code** |
| --- |
| This rule focuses on the importance of using assert() for debugging purposes but should be turned off before the code is deployed to production.  The function code below is an example of noncompliant since it exposes the end user to a possible leak of internal logic which can be exploited by attackers to discover vulnerabilities. Furthermore, it’s important to add a failed assertion can lead to a denial of service attack if triggered by a malicious user. |
| #include <cassert>  int divide(int numerator, int denominator) {  assert(denominator != 0 && "Error: Division by zero is not allowed!");  return numberator / denominator  } |

| **Compliant Code** |
| --- |
| Alternatively, when pushed to production, the assert statement should be removed or disabled and exception handling should be used for non-debugging purposes as shown in the compliant function code below. |
| int divide(int numerator, int denominator) {  if (denominator == 0) {  throw std::invalid\_argument("Error: Division by zero is not allowed!");  }  return numerator / denominator;  } |

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

| **Principles(s):**  The principle that best applies to this standard is ‘**Adopt a Secure Coding Standard” (#10).** While assertions are an important part of quality assurance and can help catch programming errors, it’s important to distinguish between debugging and error handling. This ensures unexpected behaviors does not occur from using assert(). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Coverity can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++ test | 2023.1 | [CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |
| CodeSonar | 8.1p0 | LANG.FUNCS.ASSERTS | CodeSonar can highlight the use of assertions in the code base that is in development. |

#### Coding Standard 7

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

*Source:* [*ERR51-CPP. Handle all exceptions*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions)

| **Noncompliant Code** |
| --- |
| This rule focuses on ensuring all exceptions thrown by the program are caught by an exception handler. Proper handling is crucial for stability, reliability and security. Allowing the application to terminate abnormally can result to resources not being freed, closed, and so on. It is often a vector for denial-of-service attacks.  In the example below, the executed function raises an exception but because there is no handler, the process is abnormally terminated. |
| void throwingFunction() noexcept(false);  void executeFunction()  {  raiseException();  }  int main()  {  executeFunction();  } |

| **Compliant Code** |
| --- |
| Instead, in this compliant example, the main class captures the exceptions allowing the application to terminate in a normal and controlled fashion. We encourage developers to ensure that every function that throws an exception is handled in a controlled manner. Its key to also add a catch-all in order to catch any errors that may be hiding. |
| void throwingFunction() noexcept(false)  {  throw std::runtimeError("ERROR!");  }    void executeFunction()  {    raiseException();  }    int main()  {  try {      executeFunction();     } catch (...) {      // Handle error     }  } |

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

| **Principles(s):**  The principle that applies to the standard of handling all exceptions is **‘Adopt a Secure Coding Standard’(#10)**. This ensures guidelines for exception handling to avoid unexpected crashes or termination of the program. Secure coding standards provide best practices to help with exceptions and make certain they are handled consistently and securely. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Parasoft ensures that exceptions are caught by providing an appropriate handler. Each exception should have a handler in the call path, ensuring that no exception is left unhandled. |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Polyspace uses analysis to check for unhandled exceptions but is partially covered given it does not detect every unhandled exception |
| Klocwork | 2024.2 | MISRA.CATCH.ALL | Klockwork identified instances where general catch handlers are used without specific handling logic. This ensures MISRA guidelines for error handling. |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Runchecker provides checkers to ensure the main function has a catch-all handler and used as early as possible to minimize risk. This is a partially checked as it doesn’t detect all unhandled exceptions. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-008-CPP] | Properly deallocate dynamically allocated resources |

*Source:* [*MEM51-CPP. Properly deallocate dynamically allocated resources*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources)

| **Noncompliant Code** |
| --- |
| This standard focuses on ensuring allocated memory and resources are properly deallocated to avoid memory leaks and maintain efficient resource utilization. Passing a pointer value to a deallocated function that was not previously obtained by the matching allocated function results in undefined behavior, which can create exploitable vulnerabilities.  In the example below, the main function fails to release allocated memory which may lead to excessive memory over time. |
| int main() {    int \*array = new int[10];    delete array;  } |

| **Compliant Code** |
| --- |
| Rather, this compliant example replaces the call to delete with delete[] which is the proper practice when deallocating an array of objects. It ensures the memory for the entire array is freed. We encourage developers to instead use smart pointers or ensure properly deallocate resources to manage memory and reduce the risk of dangling pointers. |
| int main() {    int \*array = new int[10];    delete[] array;  } |

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

| **Principles(s):**  The principle that applies to this standard is **‘Heed Compiler Warnings’ (#2)**. Many compilers provide warnings when memory allocation or deallocation is not handled properly. This ensures potential issues are resolved before they become security vulnerabilities. **‘Practice Defense in Depth’ (#8)** is also a valid principle as it involves putting multiple layers of controls in place to mitigate risks. This means using smart pointers, having a code review process & also using automated tools to detect memory issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024a | CERT C++: MEM51-CPP | Polyspace checks for invalid deletion of pointer, invalid free of pointer and deallocation of previously deallocated pointer |
| CodeSonar | 8.1p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | CodeSonar identifies free non-heap variable, double free, type mismatch  & memory leak. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Parasoft can scan code to identified memory that has not been deallocated. The checker recommends providing empty backers for delete and using the same form in corresponding call to delete/free |
| Astrée | 22.10 | invalid\_dynamic\_memory\_allocation  dangling\_pointer\_use | Astree can track the lifecycle of memory from allocation to deallocation to check if memory that has deallocated is accessed again. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | [STD-009-CPP] | Detect and handle standard library errors |

*Source:* [*Detect and handle standard library errors*](https://wiki.sei.cmu.edu/confluence/display/c/ERR33-C.+Detect+and+handle+standard+library+errors)

| **Noncompliant Code** |
| --- |
| This standard emphasizes the importance of detecting and handling errors from library functions especially when interacting with inputs. Failure to detect error conditions can lead to unpredictable results, including abnormal program termination, denial-of-service attacks or, in some cases, allow an attacker to run arbitrary code.  In the noncompliant example below, the program does not check if the file is successfully opened before writing to it which exposes the risk to data loss or corruption. |
| #include <fstream>  void openFile() {  std::ofstream file("example.txt");  file << "Write test";  file.close();  }  int main() {  openFile();  return 0;  } |

| **Compliant Code** |
| --- |
| Instead the function should checks if the file was successfully opened before writing to a possible invalid file stream. It Is encourage that developers establish a standard error handling framework to ensure standard practices are followed such as in exceptions. |
| #include <fstream>  void openFile() {  std::ofstream file("example.txt");  if (!file.is\_open()) {  std::cout << "Could not open the file" << std::endl;  return;  }  file << "Write test";  file.close();  }  int main() {  openFile();  return 0;  } |

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

| **Principles(s):**  The principle that applies to this standard is **‘Adopt a Secure Coding Standard’ (#10)**. This ensures safe practices and consistency is followed across all programs when detecting and handling errors. Secure standards such as CERT C/C++ provide guidelines for checking values from library functions and handling errors to undefined behavior or security vulnerabilities. **‘Practice Defense in Depth (#8)** also applies as applying multiple levels of controls can mitigate risk such as validation checks, notifications and even logging to detect early vulnerabilities and minimize impact. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.FUNCS.IRV  LANG.ERRCODE.NOTEST  LANG.ERRCODE.NZ | CodeSonar checks that return values of library functions are inspected to determine an error. This includes ignored return value, missing test of error code or non-zero error code. |
| Parasoft C/C++test | 2023.1 | CERT\_C-ERR33-a  CERT\_C-ERR33-b  CERT\_C-ERR33-d | Parasoft enforfces that the return value of the library is checked to prevent issues. The checker recommends a function that is not needed to be checked should be vast to void and all non-void function should always be checked. |
| Polyspace Bug Finder | R2024a | CERT C: Rule ERR33-C | Polyspace checks if errors returned by library functions are detected and handled. This includes return value of a sensitive function not checked. |
| TrustInSoft Analyzer | 1.38 | pointer arithmetic | TrustInSoft Analyzer through its formal methods identifies difficult to find vulnerabilities. It is exhaustively verified which checks for all possible scenarios & inputs. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Environment | [STD-010-CPP] | Do not call system() |

*Source:*[*ENV33-C. Do not call system()*](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177)

| **Noncompliant Code** |
| --- |
| This standard emphasizes that calling the system() function can be potentially unsafe and should be avoided. The system function allows the execution of commands in the operating system which can introduce security risks and compromise the local system integrity.  In the non-compliant code below, the function executes an operating system command to delete files which, if from a malicious input, can be altered to delete critical system files. |
| #include <cstdlib>  void deleteFile(const std::string& filename) {  std::string command = "del " + filename;  system(command.c\_str());  }  int main() {  std::string filename = "example.txt";  deleteFile(filename);  return 0;  } |

| **Compliant Code** |
| --- |
| Instead, a compliant example uses remove(), which is a safer way to delete files, avoiding the risk of running shell commands and allows errors to be handled in a controlled manner. We encourage developers to use safer alternatives such as trusted standard libraries to provide the necessary functionality which does not execute shell commands directly. |
| #include <cstdlib>  void deleteFile(const std::string& filename) {  if (std::remove(filename.c\_str()) != 0) {  std::perror("File deletion failed"); } else { std::cout << "File successfully deleted." << std::endl; }}  int main() {  std::string filename = "example.txt";  deleteFile(filename);  return 0;  } |

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

| **Principles(s):**  The principle that applies to this standard is **‘Default Deny’ (#5)**. This involves denying access by default unless it’s allowed and needed. This minimizes the exposure of shell commands and ensures they are never executed unless as a last resort. **‘Keep it simple (#4)** also applies as a principle as the function system() can add unpredictability & complexity to the program due to its interaction with the shell. Instead standard libraries should be used which are much more predictable, secure and easier to understand. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.PATH.SYSTEM  IO.INJ.COMMAND | CodeSonar checks for the use of system() & if there is possibility of a command injection. |
| Polyspace Bug Finder | R2024a | CERT C: Rule ENV33-C | Polyspace checks for unsafe call to a system function. |
| SonarQube C/C++ Plugin | 3.11 | S990 | SonarQube detects use of "abort", "exit", "getenv" and "system" from <stdlib.h> |
| Clang | 3.9 | cert-env33-c | Clang generates warnings when dangerous functions such as system() are used. |

### 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.

*As developers, our role is to continuously address security concerns that may be involved in any step of the development process which includes but is not limited to adhering to security best practices, regulations, standards and laws in order to deliver a secure solution that meets the customer needs. This requires engaging efforts in securing the CI/CD pipeline which Involves working with the DevOps existing process and infrasture in order to ensure security best practices are embedded to all phases of the software development lifecycle. While DevOps is a great framework which guides companies to release product effectively it does not address the security aspect. In order to build a more secure oriented model we would like to add a few modifications to align closer to a DevSecOps approach.*

*Before diving into these modifications we would first like to mention that this works only when a security first mindset is adopted. This create a culture in which teams prioritize integrating security and are encouraged to collaborate, improve and share responsibility. With this in mind, in pre-production we would integrate security driven procedures into the early phases of the lifecycle. This would mean during accessing & planning integrating a threat landscape and regulatory compliance checks in order to identify early threats that may need attention. Helping mitigate risk from the very beginning. During design, our thoughts involve implementing security best practices into development such as the standards we listed earlier and others that follow the security policies we have set in place. This means Defense in Depth, Least Privileged and Default Deny all which help ensure a comprehensive defense. When building, it’s important to incorporate processes that automatically scan for vulnerabilities. Using automated tools that ensure libraries are verified and repositories are trusted in the build process. Then in testing, I would incorporate automated unit, functional and security testing to ensure all builds are secure before proceeding to development.*

*In production, it’s easy to let security slip but it’s just as important. Right before deployment, during health checks we would encourage implementing deployment verification using penetration testing tools. This detects security weakness before going live. After deployment, monitoring should be proactive rather than reactive which means automated alerts and real time analytics to help detect and respond to attacks. With responding, it’s crucial to be ready for attacks and use tools to minimize the impact such as blocking IPs or shutting down compromised services. Then finally for maintaining, it’s key to have a post incident process to prevent downtime such as automating system recovery and conducting audits to ensure the system remains stable. These modifications enable security at every stage of the lifecycle in both pre-production and production and can reduce long term burden for the teams.*

### 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 | Probable | Medium | High | 4 |
| STD-002-CPP | High | Likely | High | High | 4 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CPP | Low | Unlikely | High | Medium | 2 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | High | Likely | Medium | High | 5 |
| STD-010-CPP | High | Probable | Medium | High | 4 |

### 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 protecting data that is stored on a disk, solid state drive or backup media. The goal of this policy to use advanced encryption standards or other similar algorithms to ensure if physical storage is stolen or accessed, the data would be unreadable without the necessary encryption keys. This reduces the surface of an attack and allows teams to focus their security strategy more on encryption then trying to protect all data. This policy applies to databases, cloud storage, archives or off-site backups. |
| Encryption in flight | **Encryption at flight** refers to a concept of protecting data that is transferred between two endpoints such as between two services or between different servers over a network. The goal of this policy is protect data where it may have the most risk of being exposed. Examples such as implementing VPN or Transport Layer Security (TLS) ensures data transmitted over a network remains secure. This applies in scenarios such as enabling TLS encryption for emails or using confidential computing features. While it may be difficult for this to be implemented in every case it should always be used involving sensitive business data. |
| Encryption in use | **Encryption at Use** refers to protecting data that is actively used by an application or being processed. The goal is to keep data secure while in memory by using partially homomorphic encryption or fully homomorphic encryption which allows a computer to perform specific type of mathematical operations on encryption data. Utilizing these encryption safeguards limit data usefulness and keep highly sensitive data secure. This policy applies in cloud computing, financial data or handling of other highly sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Authentication** is the process of verifying the identity of a user or entity that is attempting to access the system. The goal of this policy is to ensure only authorized individuals are able to access the resources of the network or application. Typically this is seen through user logins with usernames/password, multi-factor authentication or biometric methods. This prevents unauthorized access and controls who has access to sensitive information. |
| Authorization | After Authentication, **Authorization** refers to determing what level of actions a user or entity is able to perform once they are authenticated. This is seen through access control lists, policy based control or role based control. This ensures a user only has access to resources they are allowed to access with. This is key to our security principle of ‘Default Deny’ which enforces users to have access only to resources that are required and only for the time period needed. In this case new users should have the most basic of accessibility while level of access should require a chain of approvals. |
| Accounting | **Accounting** refers to the process of monitoring and recording user and system activity to maintain an activity log. This involves recording user actions such as changes made to data, login attempts or access to sensitive resources. This is helpful in detection and responding during incidents and can also allow teams to detect suspicious activity earlier. Furthermore it also holds individuals responsible. In practice this can be implemented with the use of Log monitoring/alerts and database activity monitoring. |

**\***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 | 09/22/2024 | Initial Template | Andres Sanchez |  |
| 2.0 | 10/13/2024 | Completed Template | Andres Sanchez |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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