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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

[Defense-in-Depth Illustration 25](#_Toc52464069)

[Project One 25](#_Toc52464070)

[1. Revise the C/C++ Standards 25](#_Toc52464071)

[2. Risk Assessment 25](#_Toc52464072)

[3. Automated Detection 25](#_Toc52464073)

[4. Automation 25](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 27](#_Toc52464076)

[7. Map the Principles 27](#_Toc52464077)

[Audit Controls and Management 29](#_Toc52464078)

[Enforcement 29](#_Toc52464079)

[Exceptions Process 29](#_Toc52464080)

[Distribution 30](#_Toc52464081)

[Policy Change Control 30](#_Toc52464082)

[Policy Version History 30](#_Toc52464083)

[Appendix A Lookups 30](#_Toc52464084)

[Approved C/C++ Language Acronyms 30](#_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 | When validating input data, the program verifies if the information is of the correct type, length, and format, and if it falls within the appropriate range. If the data entered is incorrect, the program notifies the user to re-enter valid information. |
| 1. Heed Compiler Warnings | Developers can detect and fix potential security issues early in the software development cycle by paying attention to compiler warnings. Ignoring these warnings could lead to severe security risks like injection attacks, buffer overflows, or other vulnerabilities, which could be exploited by hackers to gain unauthorized access to a system. Therefore, it's crucial to take compiler warnings seriously and address them promptly to improve the overall security of the software. |
| 1. Architect and Design for Security Policies | This principle requires developing security policies and guidelines that align with an organization's security goals and objectives. To implement this principle, architects and designers need to understand potential security threats and risks and consider security requirements when designing software. By integrating security considerations into design, developers can create a more secure system, reducing the likelihood of security incidents and other cyber-attacks that can damage an organization's reputation and financial stability. |
| 1. Keep It Simple | This principle urges developers to avoid unnecessary complexity in software design and development, as complexity can introduce more potential vulnerabilities and weaknesses that can be targeted by attackers seeking to exploit them. Simplifying the software system can help reduce the attack surface and enhance security by reducing the likelihood of security breaches, data leaks, and other security incidents. |
| 1. Default Deny | This approach helps ensure that only authorized entities have access to the resources they need, and all other requests are denied. By following this principle, organizations can improve their overall security posture and reduce the likelihood of successful cyber-attacks. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege is a security principle that suggests that users or processes should only have access to the resources they need to perform their intended function, and no more. This minimizes the risk of unauthorized access, data breaches, and other security incidents by reducing the attack surface. |
| 1. Sanitize Data Sent to Other Systems | The principle of Sanitize Data Sent to Other Systems emphasizes the importance of validating and cleaning data before sending it to other systems. This principle aims to prevent data from being corrupted, lost, or stolen during transmission, which can compromise the integrity and confidentiality of the data. By sanitizing data before transmission, organizations can reduce the risk of data breaches and other security incidents. |
| 1. Practice Defense in Depth | Defense in Depth is a security principle that involves using multiple layers of security controls to protect systems and data. This approach helps to ensure that if one layer is breached, there are other layers in place to prevent or mitigate the impact of an attack. By having a combination of physical, technical, and administrative controls, organizations can create a more robust and resilient security posture. |
| 1. Use Effective Quality Assurance Techniques | The security principle of Use Effective Quality Assurance Techniques highlights the importance of incorporating security considerations into software development through the use of quality assurance techniques. This involves identifying and addressing security vulnerabilities early in the development cycle to prevent security incidents and data breaches. Effective quality assurance techniques include code reviews, vulnerability assessments, and penetration testing to ensure that the software meets security standards and complies with security policies. |
| 1. Adopt a Secure Coding Standard | Adopt a Secure Coding Standard is a security principle that emphasizes the use of established and effective coding practices to minimize the risk of security vulnerabilities in software. This principle involves using industry-standard secure coding guidelines and techniques that can help prevent common coding errors and reduce the likelihood of security breaches. By adopting a secure coding standard, developers can build more secure and reliable software systems. |

### 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 write syntactically ambiguous declarations |

| **Noncompliant Code** |
| --- |
| This code example is not secure because it's ambiguous. It may be interpreted as creating an anonymous object or a named object. In this case, it creates a named object instead of an anonymous one, so the mutex is not locked as intended. |
| #include <mutex>  static std::mutex m;  static int shared\_resource;  void increment\_by\_42() {  std::unique\_lock<std::mutex>(m);  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| This solution is compliant with the coding standard as it gives a name to the lock object and uses the correct constructor. |
| #include <mutex>  static std::mutex m;  static int shared\_resource;  void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m);  shared\_resource += 42;  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "Data Type" coding standard because it involves checking that input data is in the expected format and range. By specifying the data type of a variable or object in a declaration, the code can validate that it is compatible with the expected data type for the intended use.  Keep It Simple: The "Data Type" coding standard is also related to this principle because it encourages keeping code as simple as possible by avoiding syntactically ambiguous declarations. This can help prevent confusion and errors caused by unclear or ambiguous code.  Adopt a Secure Coding Standard: The "Data Type" coding standard can also be related to this principle because it helps ensure that code is written in a way that minimizes the risk of vulnerabilities and exploits caused by ambiguous code. By specifying data types clearly, the code can avoid errors caused by unexpected or unintended data types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Static Code Analysis Tool (PVS-Studio) | 7.20 | V1023 | Detects ambiguously written declarations.  https://pvs-studio.com/en/pvs-studio/ |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not depend on the order of evaluation for side effects |

| **Noncompliant Code** |
| --- |
| The code example provided is not compliant with the standard because i is evaluated in an unsequenced manner, leading to undefined behavior. |
| void f(int i, const int \*b) {  int a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| This instance remains unaffected by the sequence in which the operands are evaluated and can only be comprehended in a singular manner. |
| void f(int i, const int \*b) {  ++i;  int a = i + b[i];  // ...  } |

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

| **Principles(s):**  Keep It Simple: The "Data Value" coding standard is related to this principle because it encourages writing code that is simple and easy to understand. By avoiding code that depends on the order of evaluation for side effects, the code becomes less complex and easier to maintain.  Use Effective Quality Assurance Techniques: This principle is related to the "Data Value" coding standard because effective quality assurance techniques, such as testing and code review, can help identify code that depends on the order of evaluation for side effects. By catching these issues early, the code can be improved to avoid unexpected side effects.  Adopt a Secure Coding Standard: The "Data Value" coding standard can also be related to this principle because it helps ensure that code is written in a way that minimizes the risk of vulnerabilities and exploits caused by code that depends on the order of evaluation for side effects. By avoiding such code, the code can be made more secure and less susceptible to unexpected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | V728 | Detects expressions that contain calls to functions with side effects that are not separated by sequence points. Such expressions may be evaluated in an order that is different from what the programmer intended, leading to unexpected results.  https://pvs-studio.com/en/pvs-studio/ |
| Clang Static Analyzer | 13.0.0 | Alpha.core.SideEffect | Detects expressions that rely on the order of evaluation for side effects, which can lead to undefined behavior if the order is not as expected.  https://clang-analyzer.llvm.org/ |

#### 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 |

| **Noncompliant Code** |
| --- |
| The code in question poses a risk of buffer overflow due to the fact that the input is unconstrained. |
| #include <iostream>  void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Utilizing std::string instead of a limited array, as demonstrated in this compliant solution, provides the optimal approach for preventing data truncation and safeguarding against buffer overflows. |
| #include <iostream>  #include <string>  void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "String Correctness" coding standard because it involves validating input data to ensure that it is in the expected format and range. By guaranteeing that storage for strings has sufficient space for character data and the null terminator, the code can validate that the input string fits within the expected space.  Keep It Simple: The "String Correctness" coding standard is also related to this principle because it encourages keeping code as simple as possible by ensuring that storage for strings has sufficient space. This can help prevent errors caused by unexpected string lengths or lack of space.  Adopt a Secure Coding Standard: The "String Correctness" coding standard can be related to this principle because it helps ensure that code is written in a way that minimizes the risk of vulnerabilities and exploits caused by insufficient storage for strings. By guaranteeing that the storage for strings has sufficient space, the code can avoid issues such as buffer overflows that can be exploited by attackers.  Use Effective Quality Assurance Techniques: This principle is related to the "String Correctness" coding standard because effective quality assurance techniques, such as testing and code review, can help identify code that does not guarantee sufficient storage for strings. By catching these issues early, the code can be improved to avoid unexpected behavior caused by insufficient storage. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity Static Analysis Tool | 2021.03 | Buffer Overflow | Provides a checker for "Buffer Overflow" issues, which can identify instances where there is insufficient space for string data and the null terminator.  https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf |
| CodeSonar | 5.5 | Buffer Overrun | This static analysis tool provides a checker for "Buffer Overrun" issues, which can identify instances where there is insufficient space for string data and the null terminator.  https://www.grammatech.com/codesonar-cc |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Avoid constructing SQL queries using string concatenation with user input. |

| **Noncompliant Code** |
| --- |
| This code block concatenates user input (username and password) into an SQL query string, which can be manipulated by an attacker to inject malicious SQL code. |
| #include <iostream>  #include <string>  int main() {  std::string username = "admin";  std::string password = "password123";    std::string query = "SELECT \* FROM users WHERE username='" + username + "' AND password='" + password + "'";    std::cout << "SQL query: " << query << std::endl;    return 0;  } |

| **Compliant Code** |
| --- |
| This code opens a connection to a SQLite database, prepares a SQL statement with placeholders for the username and password parameters, binds the parameters to the values provided by the user, executes the statement, and retrieves any matching rows. |
| #include <iostream>  #include <string>  #include <sqlite3.h>  int main() {  std::string username = "admin";  std::string password = "password123";    sqlite3\* db;  sqlite3\_stmt\* stmt;  int rc = sqlite3\_open("database.db", &db);    if (rc != SQLITE\_OK) {  std::cerr << "Failed to open database: " << sqlite3\_errmsg(db) << std::endl;  sqlite3\_close(db);  return 1;  }    const char\* sql = "SELECT \* FROM users WHERE username=? AND password=?";  rc = sqlite3\_prepare\_v2(db, sql, -1, &stmt, NULL);    if (rc != SQLITE\_OK) {  std::cerr << "Failed to prepare statement: " << sqlite3\_errmsg(db) << std::endl;  sqlite3\_close(db);  return 1;  }    rc = sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_TRANSIENT);    if (rc != SQLITE\_OK) {  std::cerr << "Failed to bind parameter: " << sqlite3\_errmsg(db) << std::endl;  sqlite3\_finalize(stmt);  sqlite3\_close(db);  return 1;  }    rc = sqlite3\_bind\_text(stmt, 2, password.c\_str(), -1, SQLITE\_TRANSIENT);    if (rc != SQLITE\_OK) {  std::cerr << "Failed to bind parameter: " << sqlite3\_errmsg(db) << std::endl;  sqlite3\_finalize(stmt);  sqlite3\_close(db);  return 1;  }    while ((rc = sqlite3\_step(stmt)) == SQLITE\_ROW) {  // Do something with the results  }    if (rc != SQLITE\_DONE) {  std::cerr << "Failed to retrieve data: " << sqlite3\_errmsg(db) << std::endl;  }    sqlite3\_finalize(stmt);  sqlite3\_close(db);    return 0;  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "SQL Injection" coding standard because it involves validating input data to ensure that it is in the expected format and range. By avoiding constructing SQL queries using string concatenation with user input, the code can validate and sanitize user input to avoid potential SQL injection attacks.  Architect and Design for Security Policies: The "SQL Injection" coding standard can also be related to this principle because it involves designing and implementing security policies to prevent and detect potential security threats. By avoiding constructing SQL queries using string concatenation with user input, the code can implement a security policy to prevent SQL injection attacks.  Default Deny: The "SQL Injection" coding standard can be related to this principle because it involves implementing a default deny policy for all inputs and requests. By avoiding constructing SQL queries using string concatenation with user input, the code can adopt a default deny policy to prevent unauthorized access to the database.  Practice Defense in Depth: This principle is related to the "SQL Injection" coding standard because it involves implementing multiple layers of security controls to protect against potential threats. By avoiding constructing SQL queries using string concatenation with user input, the code can implement a defense in depth strategy to prevent SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 8.9 | SQL Injection rule | Checks for potentially vulnerable code where user input is concatenated directly into an SQL query  https://www.sonarsource.com/products/sonarqube/ |
| Checkmarx | 9.4 | SQL Injection rule | The SQL Injection rule checks for instances of user input being directly concatenated into SQL queries, which can leave the code vulnerable to SQL injection attacks.  https://checkmarx.com/resource/documents/en/34965-46311-checkmarx-sast-overview.html |

#### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| This example creates a group of numbers without checking if there is enough space to store them. The function is labeled as "noexcept," which means it should not cause any problems, but it could make the program stop working correctly if there isn't enough room for the numbers. |
| #include <cstring>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| Another approach is to utilize ::operator new[] without std::nothrow and handle the possibility of insufficient memory by catching a std::bad\_alloc exception. |
| #include <cstring>  #include <new>  void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new (std::nothrow) int[size];  if (!copy) {  // Handle error  return;  }  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "Memory Protection" coding standard because it involves validating input data to ensure that it is in the expected format and range. By detecting and handling memory allocation errors, the code can validate input data to prevent errors caused by invalid memory allocation.  Use Effective Quality Assurance Techniques: The "Memory Protection" coding standard can also be related to this principle because effective quality assurance techniques, such as testing and code review, can help identify code that does not detect and handle memory allocation errors. By catching these issues early, the code can be improved to prevent errors caused by memory allocation issues.  Adopt a Secure Coding Standard: The "Memory Protection" coding standard can be related to this principle because it involves adopting a secure coding standard to minimize the risk of vulnerabilities and exploits caused by memory allocation issues. By detecting and handling memory allocation errors, the code can avoid issues such as buffer overflows that can be exploited by attackers.  Keep It Simple: The "Memory Protection" coding standard is also related to this principle because it encourages keeping code as simple as possible by detecting and handling memory allocation errors. This can help prevent errors caused by unexpected memory allocation, which can be difficult to debug and resolve. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.17.0 | Memcheck | A memory error detector that can detect common programming errors such as using uninitialized memory, reading/writing memory after it has been freed, and memory leaks.  https://valgrind.org/docs/manual/mc-manual.html |
| AddressSanitizer | LLVM 16.0.1 | ASan | a memory error detector that detects memory corruption bugs such as buffer overflows, use-after-free errors, and heap buffer overflows. It provides a fast and efficient way to detect memory errors in C and C++ programs at runtime.  https://learn.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170 |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assert() to check for violations of assumptions |

| **Noncompliant Code** |
| --- |
| In this example, the code is checking for a division by zero by manually outputting an error message to std::cerr. |
| int divide(int x, int y) {  if (y == 0) {  std::cerr << "Error: division by zero\n";  return 0;  }  return x / y;  } |

| **Compliant Code** |
| --- |
| The code uses the assert() macro to check for the division by zero condition. The assertion includes an informative error message that will be printed to the standard error stream if the condition is violated. |
| int divide(int x, int y) {  assert(y != 0 && "Error: division by zero"); // Check for division by zero  return x / y;  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "Assertions" coding standard because it involves validating input data to ensure that it is in the expected format and range. By using assert() to check for violations of assumptions, the code can validate input data to prevent unexpected behavior caused by invalid input.  Use Effective Quality Assurance Techniques: The "Assertions" coding standard can also be related to this principle because effective quality assurance techniques, such as testing and code review, can help identify code that does not use assert() to check for violations of assumptions. By catching these issues early, the code can be improved to prevent unexpected behavior caused by invalid assumptions.  Adopt a Secure Coding Standard: The "Assertions" coding standard can be related to this principle because it involves adopting a secure coding standard to minimize the risk of vulnerabilities and exploits caused by invalid assumptions. By using assert() to check for violations of assumptions, the code can avoid issues such as buffer overflows that can be exploited by attackers.  Keep It Simple: The "Assertions" coding standard is also related to this principle because it encourages keeping code as simple as possible by using assert() to check for violations of assumptions. This can help prevent unexpected behavior caused by complex and difficult-to-understand code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.12.2 | MISSING\_ASSERT | Coverity can automatically detect assert() statements that are not present in the code and flag them as a potential issue.  https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant since the call to f(), which was registered as an exit handler using std::at\_exit(), may invoke std::terminate() due to the possibility of throwing\_func() throwing an exception. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution involves f() handling any exceptions that might be thrown by throwing\_func(), without propagating them further. |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }  int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):**  Practice Defense in Depth: This principle is related to the "Exceptions" coding standard because it involves using multiple layers of defense to protect against unexpected behavior that could lead to abrupt termination of the program. By using exception handling, the code can gracefully handle errors and avoid abruptly terminating the program.  Use Effective Quality Assurance Techniques: The "Exceptions" coding standard can also be related to this principle because effective quality assurance techniques, such as testing and code review, can help identify code that may abruptly terminate the program. By catching these issues early, the code can be improved to prevent unexpected program termination.  Keep It Simple: The "Exceptions" coding standard is also related to this principle because it encourages keeping code as simple as possible by using exception handling to avoid abruptly terminating the program. This can help prevent unexpected program behavior caused by complex and difficult-to-understand code.  Architect and Design for Security Policies: The "Exceptions" coding standard can be related to this principle because abrupt program termination can leave the system in an unpredictable state that can be exploited by attackers. By using exception handling to gracefully handle errors, the code can avoid these issues and maintain the security of the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | V730 | PVS-Studio has a rule that detects calls to exit()  https://pvs-studio.com/en/pvs-studio/ |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-008-CPP] | Do not cast to an out-of-range enumeration value |

| **Noncompliant Code** |
| --- |
| This code example is not compliant because it checks whether a given value is within the acceptable range of enumeration values after casting it to the enumeration type. On a two's complement system, if a value outside the valid range of values for EnumType is passed to f(), the cast to EnumType would result in an unspecified value, leading to unspecified behavior when used within the if statement. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| To ensure that the conversion to the enumeration type does not result in an unspecified value, this compliant solution verifies that the value can be represented by the enumeration type before performing the conversion. It accomplishes this by limiting the converted value to a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };  void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):**  Validate Input Data: This principle is related to the "Integers" coding standard because it involves validating input data to ensure that it is in the expected range. By avoiding casting to an out-of-range enumeration value, the code can prevent unexpected behavior caused by invalid input.  Adhere to the Principle of Least Privilege: The "Integers" coding standard can also be related to this principle because it involves limiting access to resources and functionality to the minimum necessary. By avoiding casting to an out-of-range enumeration value, the code can prevent unexpected behavior caused by accessing resources or functionality outside of the intended scope.  Use Effective Quality Assurance Techniques: The "Integers" coding standard can also be related to this principle because effective quality assurance techniques, such as testing and code review, can help identify code that casts to an out-of-range enumeration value. By catching these issues early, the code can be improved to prevent unexpected behavior caused by invalid casting.  Adopt a Secure Coding Standard: The "Integers" coding standard can be related to this principle because it involves adopting a secure coding standard to minimize the risk of vulnerabilities and exploits caused by invalid casting. By avoiding casting to an out-of-range enumeration value, the code can avoid issues such as buffer overflows that can be exploited by attackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | enumCastOutOfRange | The rule detects when an enumeration value is cast to an out-of-range value.  https://cppcheck.sourceforge.io/ |
| Clang-tidy | 17.0.0 | Bugprone-enum-integer-conversion | Flags when an enumeration value is cast to an out-of-range integer value  https://clang.llvm.org/docs/DiagnosticsReference.html |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output (FIO) | [STD-009-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| This noncompliant code creates a std::fstream object called "file". The constructor for std::fstream invokes std::basic\_filebuf<T>::open(). Since the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors, the underlying std::basic\_filebuf<T> object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>  void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| This compliant solution guarantees proper closure of file resources by calling std::fstream::close() before calling std::terminate(). |
| #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):**  Practice Defense in Depth: This principle is related to the "FIO" coding standard because it involves implementing multiple layers of security controls to protect against vulnerabilities and attacks. By closing files when they are no longer needed, the code can prevent unauthorized access to sensitive information stored in those files.  Adopt a Secure Coding Standard: The "FIO" coding standard can also be related to this principle because it involves adopting a secure coding standard to minimize the risk of vulnerabilities and exploits caused by leaving files open. By closing files when they are no longer needed, the code can avoid issues such as file descriptor leaks that can be exploited by attackers.  Heed Compiler Warnings: The "FIO" coding standard can also be related to this principle because compiler warnings can alert developers to potential issues such as leaving files open. By paying attention to these warnings and closing files when they are no longer needed, the code can avoid issues that could lead to unexpected behavior or security vulnerabilities.  Keep It Simple: The "FIO" coding standard can also be related to this principle because it involves keeping code simple and easy to understand. By closing files when they are no longer needed, the code can avoid unnecessary complexity and ensure that resources are used efficiently. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CppCheck | 2.10 | ResourceLeak | Detects when a file is opened but not closed before the function returns, leading to a resource leak.  https://cppcheck.sourceforge.io/ |
| PVS-Studio | 7.20 | V528 | The V528 check detects when a file is opened but not closed before the function returns, leading to a resource leak.  https://pvs-studio.com/en/pvs-studio/ |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initialization | [STD-010-CPP] | Obey the one-definition rule |

| **Noncompliant Code** |
| --- |
| This noncompliant code example creates a class named S with a single public, non-static data member int a defined with two different definitions in two different translation units. Although the two definitions are equivalent in functionality, they are not defined using the same sequence of tokens. This violates the ODR and leads to undefined behavior. |
| // a.cpp  struct S {  int a;  };  // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| The solution for mitigating this issue depends on the programmer's intent. If the programmer intends for the same class definition to be visible in both translation units due to common usage, a header file should be used to introduce the object into both translation units, as demonstrated in this compliant solution. |
| // S.h  struct S {  int a;  };  // a.cpp  #include "S.h"  // b.cpp  #include "S.h" |

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

| **Principles(s):**  Default Deny: This principle is related to the "Declarations and Initialization" coding standard because it involves setting defaults and limiting access to only what is necessary. By obeying the one-definition rule, the code can avoid accidentally creating multiple definitions for the same variable or function, which could lead to unexpected behavior.  Architect and Design for Security Policies: The "Declarations and Initialization" coding standard can also be related to this principle because it involves designing code with security policies in mind. By obeying the one-definition rule, the code can ensure that there are no conflicting definitions that could lead to security vulnerabilities.  Use Effective Quality Assurance Techniques: The "Declarations and Initialization" coding standard can also be related to this principle because it involves using effective quality assurance techniques to identify and correct issues in code. By ensuring that there is only one definition for each variable or function, the code can avoid issues such as undefined behavior or crashes.  Keep It Simple: The "Declarations and Initialization" coding standard can also be related to this principle because it involves keeping code simple and easy to understand. By avoiding multiple definitions and obeying the one-definition rule, the code can avoid unnecessary complexity and ensure that the code is easy to read and maintain. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| PVS-Studio | 7.20 | V1044 | This diagnostic warns about duplicate function definitions across multiple files.  https://pvs-studio.com/en/pvs-studio/ |

### 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.

On the pre-production side, automation can be applied to all four sections: Build, Design, Assess and plan, and Verify and test. For example, tools can be used to automate code reviews and vulnerability scanning, which can help identify potential security issues early in the development process. Additionally, automated testing tools can help ensure that code is free of bugs and other vulnerabilities before it is released.

On the production side of things, automation can be applied to the Monitor and detect section as well as the Respond section. For instance, automated monitoring tools can help detect potential threats and breaches in real-time, allowing for a more rapid response. Additionally, automated incident response tools can help quickly and efficiently address security incidents as they arise.

Overall, incorporating automation into the development process can help ensure compliance with cybersecurity and secure coding standards. By applying automation to the appropriate sections of the workflow, organizations can achieve a more efficient and effective development process while also reducing the risk of security breaches and other vulnerabilities.

### 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 | Medium | Likely | Low | Medium | 3 |
| STD-002-CPP | High | Likely | High | High | 5 |
| STD-003-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Low | Unlikely | Low | Medium | 2 |
| STD-007-CPP | Medium | Likely | Medium | High | 4 |
| STD-008-CPP | Medium | Likely | Low to Medium | Medium | 3 |
| STD-009-CPP | Medium | Likely | Low | High | 3 |
| STD-010-CPP | Low | Unlikely | Low | Medium | 1 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | **Policy:** All sensitive data at rest must be encrypted using strong, industry-standard encryption algorithms.  **What is it?**  Encryption in rest refers to the process of encrypting data that is stored in databases or file systems. This policy is put in place to protect sensitive information from unauthorized access in case of a security breach or theft of physical storage media.  **Implementation:** All data storage systems must be configured to encrypt data at rest using strong, industry-standard encryption algorithms. Data encryption must be implemented on all storage systems, including databases, file servers, cloud storage, and backup systems. Encryption should be applied to all sensitive data.  **Why should it be used?**  Encryption in rest is essential to protect sensitive data from unauthorized access. With the increasing number of cyberattacks and data breaches, it is important to implement strong encryption to protect sensitive data from theft, unauthorized access, or tampering. Encryption in rest helps to ensure the confidentiality, integrity, and availability of data, and helps to prevent reputational damage and legal liability. |
| Encryption at flight | **Policy:** All sensitive data that is being transmitted between systems or over a network must be encrypted using strong, industry-standard encryption algorithms.  **What is it?**  Encryption in flight, also known as transport layer encryption, refers to the encryption of data that is being transmitted between systems over a network.  **Implementation:** Configure the systems and applications to use the appropriate encryption protocols and algorithms. Verify that all traffic is encrypted, and that the encryption is properly configured. Periodically review and update the encryption protocols and algorithms to ensure they remain up-to-date and effective.  **Why should it be used?**  Encryption in flight is a critical security measure that should be implemented to protect the confidentiality and integrity of data transmitted over a network. By following this policy, Green Pace can ensure that their systems and applications are protected against unauthorized access or modification of sensitive data during transmission. |
| Encryption in use | **Policy:** All sensitive data that is being processed or accessed in memory, CPU, caches, or registers must be encrypted using strong, industry-standard encryption algorithms.  **What is it?**  Encryption in use is a security measure to protect sensitive data while it is being processed or accessed in memory, CPU caches, or registers. Encryption in use helps to prevent unauthorized access to sensitive information by rendering it unreadable to attackers.  **Implementation:** Identify which processes or data in the system require encryption in use and ensure that encryption is applied to those specific areas. Ensure that encryption keys used for encryption in use are kept secure and managed properly, including key rotation and proper key management. Monitor and log activities related to encryption in use to detect any unusual activities and respond to potential security breaches promptly.  **Why should it be used?**  Encryption in use is an essential security measure that helps to protect sensitive information from unauthorized access and compromise. Encryption in use provides an additional layer of protection against attacks such as memory-based attacks or other forms of cyber-attacks that aim to gain access to sensitive data in the system memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | **Policy:** Establish guidelines for authentication practices to ensure that users are appropriately identified and authenticated before accessing organizational resources  **What is it?**  Authentication is the process of verifying the identity of a user, system, or device attempting to access an application, network, or other resources.  **Implementation:** Implement strong passwords, multi-factor authentication, account lockout, and role-based access control  **Why should it be used?**  Authentication is critical to preventing unauthorized access to systems and data. Failure to implement proper authentication practices can lead to data breaches, system compromise, and loss of sensitive information. |
| Authorization | **Policy:** Authorization should be implemented based on the principle of least privilege. Access should be granted to only those resources that are necessary to perform job functions.  **What is it?**  Authorization refers to determining what level of access the authenticated user or system should have. This involves defining user roles and access permissions to specific resources.  **Implementation:** Access controls should be put in place to limit users' ability to perform actions based on their roles and permissions, and the policy should be regularly reviewed and updated to ensure its effectiveness in protecting sensitive resources.  **Why should it be used?**  Authorization is a critical component of an organization's security. By implementing the principles of least privilege, organizations can reduce the risk of unauthorized access to sensitive resources and information. Proper authorization can also help to ensure accountability and traceability and can support compliance with industry regulations and standards. |
| Accounting | **Policy:** All information systems must implement an accounting mechanism to ensure the accurate recording of all security-related events that occur within the system.  **What is it?**  Accounting refers to the process of tracking and recording the actions of authenticated and authorized users or systems. This includes monitoring and logging of user activity and system behavior to detect and respond to security incidents or policy violations.  **Implementation:** All users of all information systems must ensure that an accounting mechanism is implemented in their systems. The accounting Mechanism must be stored securely and backed up regularly.  **Why should it be used?**  The purpose of this policy is to ensure that all security-related events within an information system are accurately recorded, and that the information is readily available to support investigations and audits. |

**\***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 | 04/09/2023 | Security Policy | Gabriel Balicki |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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