**Green Pace Developer: Security Policy Guide for Project one**



# 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 | In an effort to minimize potential security vulnerabilities, it is essential to rigorously validate all incoming input data—whether it originates from untrusted sources or even those deemed reliable. This process ensures that no malicious or harmful data infiltrates the system, thereby safeguarding its integrity and functionality. By implementing thorough validation measures, we can create a more secure environment, actively preventing any threats from entering the system. |
| 1. Heed Compiler Warnings | To ensure the highest quality of our code, it is essential to compile and test it as thoroughly as possible. While a warning might seem like a mere annoyance at first glance, it could hint at more significant underlying issues. Therefore, it’s crucial to examine every warning diligently, as overlooking them could lead to complications down the line. Prioritizing this process helps us maintain robust and reliable code. |
| 1. Architect and Design for Security Policies | When you approach your coding with a strong emphasis on security from the very beginning, you can significantly reduce the amount of time and effort required later on. By carefully planning and designing your code to prioritize security, all while ensuring it fulfills its intended purpose, you can avoid the complexities of implementing security measures post-development. This proactive mindset helps protect against vulnerabilities and minimizes the need for future fixes, ultimately streamlining the entire development process and enhancing the overall integrity of your software. |
| 1. Keep It Simple | Maintaining clean and straightforward code significantly enhances its overall manageability, including aspects related to security. When code is easy to read and comprehend, it becomes simpler to identify vulnerabilities and implement protective measures. Thus, prioritizing clarity not only fosters better functionality but also establishes a robust foundation for safeguarding the integrity of the code. |
| 1. Default Deny | Implementing denial as a default standard is an effective strategy, as it safeguards sensitive information by preventing unauthorized individuals from accessing it. This approach not only protects valuable resources but also ensures that those who legitimately need access are required to follow the correct procedures to obtain it. By establishing this framework, organizations can maintain a higher level of security while fostering accountability among users. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege aligns seamlessly with the default denial approach, establishing a framework where only individuals who genuinely require elevated access are permitted to pursue it through designated procedures. This structured process emphasizes that access granted is strictly what is essential for the task at hand, thereby minimizing potential security risks. By ensuring that only the necessary information is accessible, this method effectively reduces the overall quantity of sensitive data in circulation, thereby enhancing the organization's security posture. Through these diligent measures, the risk of unauthorized access and data breaches is significantly limited. |
| 1. Sanitize Data Sent to Other Systems | It is essential to meticulously sanitize any data transmitted to external systems, ensuring it is free of potential vulnerabilities and sensitive information. Failing to do so can expose code to serious risks, leading to exploits that may result in disastrous consequences, which can vary widely depending on the context and specific circumstances involved. Careful scrutiny of data before sharing is vital to mitigate these risks effectively. |
| 1. Practice Defense in Depth | A robust security framework necessitates multiple layers of protection. By integrating various redundant and diverse defense mechanisms, systems can significantly bolster their security. This multilayered approach helps to protect against potential breaches, ensuring that even if one layer is compromised, others remain in place to mitigate the threat and prevent chaos from unfolding. Such a comprehensive strategy is essential for safeguarding sensitive information and maintaining the integrity of the entire system. |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance (QA) is essential in every process, even if it often goes unrecognized. Its primary purpose is to identify flaws and vulnerabilities, which is crucial to ensuring the final product's quality. Addressing these issues internally is far preferable to having customers or malicious actors discover them later, as the consequences can be severe, including loss of trust and reputational damage. Ultimately, strong QA practices lead to a more reliable product that meets user expectations. |
| 1. Adopt a Secure Coding Standard | Establishing a solid secure coding standard complements other key principles, such as incorporating security policies in design and effective quality assurance techniques. By prioritizing these elements early in the development process, you streamline workflow and reduce the risk of costly problems later on, ultimately enhancing both security and project efficiency. |

### 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] | Ensure that division operator does not result in division by zero error |

| **Noncompliant Code** |
| --- |
| This noncompliant code example prevents signed integer overflow in compliance with INT32-C. Ensure that operations on signed integers do not result in overflow but fails to prevent a divide-by-zero error during the division of the signed operands s\_a and s\_b |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_a == LONG\_MIN) && (s\_b == -1)) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution tests the division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow: |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Noncompliant Code 2** |
| --- |
| implement a simple division function that does not check for division by zero, which can lead to runtime errors. |
| double divide(double numerator, double denominator) {  // Non-compliant: No check for division by zero  return numerator / denominator;  }  int main() {  double a = 10.0;  double b = 0.0;  // This will lead to undefined behavior  double result = divide(a, b);  std::cout << "Result: " << result << std::endl; // This line may not execute  return 0;  } |

| **Compliant Code 2** |
| --- |
| safely check for division by zero before performing the division. |
| double divide(double numerator, double denominator) {  // Compliant: Check for division by zero  if (denominator == 0) {  throw std::invalid\_argument("Error: Division by zero is not allowed.");  }  return numerator / denominator;  }  int main() {  double a = 10.0;  double b = 0.0;  try {  double result = divide(a, b);  std::cout << "Result: " << result << std::endl;  } catch (const std::invalid\_argument& e) {  std::cerr << e.what() << std::endl; // Handle the division by zero error  }  return 0;  } |

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

| **Principles(s):** Validating Input Data: Ensure all input is accurate and secure to prevent harmful data from entering the system, minimizing security risks.  Straight to the point code: Prioritize clarity and efficiency in the code to enhance maintainability and reduce errors, adhering to best practices.  Security measures design: Build security measures into the code to proactively address vulnerabilities, creating a robust architecture. (principle 1, 2, and 4 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P18 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | Zrodiv, zerodivcond | Context sensitive analysis of division by zero. Not detected for division by struct member / array element / pointer data that is 0. Detected when there is unsafe division by variable before/after test if variable is zero |
| CodeSonar | 8.1p0 | LANG.ARITH.DIVZERO  LANG.ARITH.FDIVZERO | Division by zero  Float Division By Zero |
| Astrée | 24.04 | int-division-by-zero  int-modulo-by-zero | Fully checked |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT33-a | Avoid division by zero |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002 CPP] | Do not modify constant objects |

| **Noncompliant Code** |
| --- |
| This code example demonstrates a scenario where a constant object is inadvertently modified, highlighting an area for improvement in compliance. |
| const int \*\*ipp;  int \*ip;  const int i = 42;    void func(void) {  ipp = &ip; /\* Constraint violation \*/  \*ipp = &i; /\* Valid \*/  \*ip = 0; /\* Modifies constant i (was 42) \*/  } |

| **Compliant Code** |
| --- |
| The compliant solution depends on the programmer's intent. If the programmer intends for the value of i to be modifiable, then it should not be declared as a constant, as shown in this compliant solution. |
| int \*\*ipp;  int \*ip;  int i = 42;    void func(void) {  ipp = &ip; /\* Valid \*/  \*ipp = &i; /\* Valid \*/  \*ip = 0; /\* Valid \*/  } |

| **Noncompliant Code 2** |
| --- |
| we define a class that has a method intended to modify a member variable, but we attempt to call this method on a constant object. |
| public:  int value;  Counter(int v) : value(v) {}  // Method to increment the value  void increment() {  value++;  }  };  void modifyConstantObject(const Counter& counter) {  // Non-compliant: Attempting to modify a constant object  counter.increment(); // This will cause a compilation error  }  int main() {  Counter c(10);  modifyConstantObject(c); // Trying to modify a constant object  std::cout << "Counter value: " << c.value << std::endl; // This won't be reached  return 0;  } |

| **Compliant Code 2** |
| --- |
| respect the const-correctness and do not attempt to modify constant objects. |
| public:  int value;  Counter(int v) : value(v) {}  // Method to get the current value  int getValue() const {  return value; // Constant method, does not modify the object  }  // Method to increment the value (non-const method)  void increment() {  value++;  }  };  void displayConstantObject(const Counter& counter) {  // Compliant: Accessing the constant object without modifying it  std::cout << "Counter value: " << counter.getValue() << std::endl;  }  int main() {  Counter c(10);  displayConstantObject(c); // Now we are not modifying the constant object  return 0;  } |

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

| **Principles(s):** When you use invalid references, pointers, or iterators to access elements within a container, it can lead to undefined behavior. This means that the program may exhibit unpredictable results, crash, or behave in unintended ways. Invalid references can arise from situations such as dereferencing a pointer that points to an object that has already been destroyed, or using an iterator that has gone past the end of a container or has not been properly initialized. As a result, it is crucial to ensure that all references, pointers, and iterators are valid and point to existing elements within the container to maintain program stability and performance.(principle 1, 2, and 4 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2023.1 | CERT\_C-EXP40-a | A cast shall not remove any 'const' or 'volatile' qualification from the type of a pointer or reference |
| Polyspace Bug Finder | R2024a | CERT C: Rule EXP40-C | Checks for write operations on const qualified objects (rule fully covered) |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified (see the compliant and the non-compliant example). |
| LDRA tool suite | 9.7.1 | 582 S | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003 CPP] | Avoid trying to create a `std::string` from a null pointer. |

| **Noncompliant Code** |
| --- |
| Here, `std::string` object is instantiated from the value returned by the `std::getenv()` function. It's important to note that when `std::getenv()` is called with an environment variable that does not exist, it returns a null pointer. This situation poses a risk of undefined behavior, as attempting to create a `std::string` from a null pointer can lead to potential runtime errors or crashes. Proper error handling should be implemented to check for null pointers before attempting to create the string to avoid these issues. |
| void something() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results of the call to std::getenv() are checked for null before constructing the std::string object. |
| void somthing() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

| **Noncompliant Code 2** |
| --- |
| This example demonstrates an attempt to create a std::string from a null pointer, which can lead to undefined behavior. |
| const char\* nullPtr = nullptr;  // Non-compliant: Attempting to create std::string from a null pointer  std::string str(nullPtr); // This can lead to undefined behavior  std::cout << "String created: " << str << std::endl; |

| **Compliant Code 2** |
| --- |
| In this compliant example, the code checks if the pointer is null before attempting to create a std::string. |
| const char\* nullPtr = nullptr;  // Compliant: Check if the pointer is null before creating std::string  if (nullPtr) {  std::string str(nullPtr);  std::cout << "String created: " << str << std::endl;  } else {  std::cout << "Cannot create string from a null pointer." << std::endl;  } |

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

| **Principles(s):** Dereferencing a null pointer is undefined behavior, typically abnormal program termination. (principle 1 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004 CPP] | Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |

| **Noncompliant Code 2** |
| --- |
| without any sanitization, Sql Query becomes vulnerable to SQL injection. |
| void insertUser(const std::string& username, const std::string& password) {  sqlite3\* db;  sqlite3\_open("users.db", &db);  // Non-compliant: Directly using user input in the SQL query  std::string query = "INSERT INTO users (username, password) VALUES ('" + username + "', '" + password + "');";    char\* errorMessage;  if (sqlite3\_exec(db, query.c\_str(), nullptr, 0, &errorMessage) != SQLITE\_OK) {  std::cerr << "SQL error: " << errorMessage << std::endl;  sqlite3\_free(errorMessage);  }    sqlite3\_close(db);  } |

| **Compliant Code 2** |
| --- |
| rewriting the function using prepared statements to safely handle user input, prevents SQL injection. |
| void insertUser(const std::string& username, const std::string& password) {  sqlite3\* db;  sqlite3\_open("users.db", &db);  // Compliant: Using prepared statements to prevent SQL injection  sqlite3\_stmt\* stmt;  const char\* query = "INSERT INTO users (username, password) VALUES (?, ?);";  if (sqlite3\_prepare\_v2(db, query, -1, &stmt, nullptr) == SQLITE\_OK) {  sqlite3\_bind\_text(stmt, 1, username.c\_str(), -1, SQLITE\_STATIC);  sqlite3\_bind\_text(stmt, 2, password.c\_str(), -1, SQLITE\_STATIC);    if (sqlite3\_step(stmt) != SQLITE\_DONE) {  std::cerr << "Execution failed: " << sqlite3\_errmsg(db) << std::endl;  }  } else {  std::cerr << "Preparation failed: " << sqlite3\_errmsg(db) << std::endl;  }  sqlite3\_finalize(stmt);  sqlite3\_close(db);  } |

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

| **Principles(s):** Failure to sanitize data passed to a complex subsystem can lead to an injection attack, data integrity issues, and a loss of sensitive data. (principle 3, 4,5,6 and 7 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Klocwork | 2024.2 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| Astrée | 24.04 | Supported by stubbing/taint analysis |  |

#### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| 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** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| [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;  } |

| **Noncompliant Code 2** |
| --- |
| allocate memory using new without checking if the allocation was successful. Could cause the program to crash or exhibit undefined behavior. |
| void allocateMemory(int size) {  // Non-compliant: Memory allocation without error handling  int\* arr = new int[size];    // Use the array  for (int i = 0; i < size; ++i) {  arr[i] = i;  std::cout << arr[i] << " ";  }    delete[] arr; // Remember to free allocated memory  } // ...  delete [] copy;  } |

| **Compliant Code 2** |
| --- |
| properly detect and handle memory allocation errors using a try-catch block. |
| void allocateMemory(int size) {  try {  // Compliant: Memory allocation with error handling  int\* arr = new int[size];  // Use the array  for (int i = 0; i < size; ++i) {  arr[i] = i;  std::cout << arr[i] << " ";  }  delete[] arr; // Free allocated memory  } catch (const std::bad\_alloc& e) {  std::cerr << "Memory allocation failed: " << e.what() << std::endl;  }  } |

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

| **Principles(s):** Failing to detect allocation failures can lead to abnormal program termination and denial-of-service attacks. (principle 1, 2 , 4 and 10 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| PVS-Studio | 7.33 | V522, V668 |  |
| Helix QAC | 2024.2 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| LDRA tool suite | 9.7.1 | 45D | Partially implemented |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This noncompliant code utilizes the assert() macro to verify a property related to a memory-mapped structure that is crucial for the code to function correctly. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| For assertions that only involve constant expressions, a preprocessor conditional statement can be utilized |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };  #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

| **Noncompliant Code 2** |
| --- |
| Here we check a constant expression but do so in a way that doesn't utilize static assertions. This could lead to runtime errors or unintended behavior if the constant does not meet our expectations. |
| const int VALUE = 10;  void checkValue() {  // Non-compliant: Checking a constant value at runtime  if (VALUE != 10) {  std::cerr << "VALUE is not equal to 10!" << std::endl;  } else {  std::cout << "VALUE is valid." << std::endl;  }  } };  int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned  int) + sizeof(unsigned int));  } |

| **Compliant Code 2** |
| --- |
| use a static assertion to validate the value of a constant expression at compile time. |
| const int VALUE = 10;  // Compliant: Using static assertion to check the value at compile time  static\_assert(VALUE == 10, "VALUE must be equal to 10"); |

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

| **Principles(s):** Static assertion is a valuable diagnostic tool for finding and eliminating software defects that. (Principle 1, 2, 8, and 9 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | [Insert text.] | [Insert text.] |
| Clang | 3.9 | Misc-static-assert | Checked by Clang-tidy |
| Axivion Bauhaus Suite | 6.9.0 | CertC-DCL03 |  |
| ECLAIR | 1.2 | CC2.DCL03 | Fully Implemented |

#### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| 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** |
| --- |
| In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow. |
| 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  }  // ...  } |

| **Noncompliant Code 2** |
| --- |
| program terminates abruptly using exit() or abort() in response to an error condition. This can lead to resource leaks, incomplete data processing, and a poor user experience. |
| void process(int value) {  if (value < 0) {  // Non-compliant: Abruptly terminating the program  std::cerr << "Error: Negative value encountered. Terminating program." << std::endl;  exit(EXIT\_FAILURE);  }  std::cout << "Processing value: " << value << std::endl;  } |

| **Compliant Code 2** |
| --- |
| compliant version that handles errors gracefully without abruptly terminating the program. |
| void process(int value) {  if (value < 0) {  // Compliant: Throwing an exception to handle the error  throw std::invalid\_argument("Negative value encountered.");  }  std::cout << "Processing value: " << value << std::endl;  }  int main() {  try {  process(-5); // This will throw an exception  } catch (const std::invalid\_argument& e) {  std::cerr << "Error: " << e.what() << std::endl;  // Handle error gracefully, e.g., log it, clean up, or inform the user  }  std::cout << "Program continues running." << std::endl;  return 0;  } |

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

| **Principles(s):** Allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. (Principle 3,8, and 10 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | [Insert text.] |
| Helix QAC | 2024.2 | C++5014 | [Insert text.] |
| Klocwork | 2024.2 | MISRA.TERMINATE  CERT.ERR.ABRUPT\_TERM | [Insert text.] |
| LDRA tool suite | 9.71 | 122 S | [Insert text.] |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-008 CPP] | Value-returning functions must return a value from all exit paths |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

| **Noncompliant Code 2** |
| --- |
| function that is supposed to return an integer but does not return a value in one of its exit paths. This can lead to undefined behavior. |
| int getValue(int condition) {  if (condition > 0) {  return 42; // Return value for positive condition  }  // Non-compliant: No return statement for condition <= 0  }  int main() {  int result = getValue(0); // This will lead to undefined behavior  std::cout << "Result: " << result << std::endl; // Undefined behavior  return 0;  } |

| **Compliant Code 2** |
| --- |
| write a compliant version that ensures every possible exit path returns a value. |
| int getValue(int condition) {  if (condition > 0) {  return 42; // Return value for positive condition  } else {  return -1; // Return a default value for non-positive condition  }  }  int main() {  int result = getValue(0); // This will now safely return -1  std::cout << "Result: " << result << std::endl; // Outputs: Result: -1  return 0;  } |

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

| **Principles(s):** Failing to return a value from a code path in a value-returning function results in undefined behavior that might be exploited to cause data integrity violations. (Principle 3, 9 and 10 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.MRS  LANG.STRUCT.NVNR | Missing return statement  Non-void noreturn. |
| Helix QAC | 2024.2 | DF2888 |  |
| Klocwork | 2024.2 | FUNCRET.GEN  FUNCRET.IMPLICIT |  |
| LDRA tool suite | 9.71 | 2 D, 36 S | Fully Implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input output | [STD-009 CPP] | Do not alternately input and output from a file stream without an intervening positioning call |

| **Noncompliant Code** |
| --- |
| This example of noncompliant code appends data to a file, then reads from it. However, because there is no call to reposition the file pointer between the formatted output and input, the behavior is undefined. |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {    return;  }    file << "Output some data";  std::string str;  file >> str;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::basic\_istream::seekg() function is invoked between the output and input, thus eliminating any undefined behavior. |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Some Output goes here";  std::string str;  file.seekg(0, std::ios::beg);xxxxxxxxxxxxxxxxxxxxxxxxxxx  file >> str;  } |

| **Noncompliant Code 2** |
| --- |
| read from and write to a file stream without repositioning can lead to unexpected behavior. |
| void readWriteFile(const std::string& filename) {  std::fstream file(filename, std::ios::in | std::ios::out | std::ios::trunc);  if (!file) {  std::cerr << "Failed to open file." << std::endl;  return;  }  // Write data to the file  file << "Hello, World!" << std::endl;  // Non-compliant: Attempting to read from the same file without repositioning  std::string line;  std::getline(file, line); // This can lead to undefined behavior  std::cout << "Read from file: " << line << std::endl;  file.close(); |

| **Compliant Code 2** |
| --- |
| compliant version that properly uses positioning calls to ensure safe reading and writing. |
| void readWriteFile(const std::string& filename) {  std::fstream file(filename, std::ios::in | std::ios::out | std::ios::trunc);  if (!file) {  std::cerr << "Failed to open file." << std::endl;  return;  }  // Write data to the file  file << "Hello, World!" << std::endl;  // Compliant: Move the file pointer back to the beginning before reading  file.seekg(0); // Reposition to the start of the file  std::string line;  std::getline(file, line); // Now this is safe  std::cout << "Read from file: " << line << std::endl;  file.close();  } |

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

| **Principles(s):** Inputting and outputting alternately from a stream without a flush or positioning call in between is considered undefined behavior. (Principle 3, 5, 8, and 10 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Loikely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP\_FIO50-a | Do not alternately input and output from a stream without an intervening flush or positioning call |
| Polyspace Bug Finder | R2020a | ECRT C++: FIO50-CPP | Checks for alternating input and output from a stream without f lush or positioning call (rule fully covered) |
| Helix QAC | 2021.1 |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programing | [STD-010 CPP] | Gracefully handle self-copy assignment |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the copy assignment operator does not protect against self-copy assignment. If self-copy assignment occurs, this->s1 is deleted, which results in rhs.s1 also being deleted. The invalidated memory for rhs.s1 is then passed into the copy constructor for S, which can result in dereferencing an invalid pointer. |
| struct S { S(const S &) noexcept; /\* ... \*/ };    class T {  int n;  S \*s1;    public:  T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(\*rhs.s1) : nullptr) {}  ~T() { delete s1; }    // ...    T& operator=(const T &rhs) {  n = rhs.n;  delete s1;  s1 = new S(\*rhs.s1);  return \*this;  }  }; |

| **Compliant Code** |
| --- |
| This compliant solution guards against self-copy assignment by testing whether the given parameter is the same as this. If self-copy assignment occurs, then operator= does nothing; otherwise, the copy proceeds as in the original example. |
| struct S { S(const S &) noexcept; /\* ... \*/ };    class T {  int n;  S \*s1;    public:  T(const T &rhs) : n(rhs.n), s1(rhs.s1 ? new S(\*rhs.s1) : nullptr) {}  ~T() { delete s1; }    // ...    T& operator=(const T &rhs) {  if (this != &rhs) {  n = rhs.n;  delete s1;  try {  s1 = new S(\*rhs.s1);  } catch (std::bad\_alloc &) {  s1 = nullptr; // For basic exception guarantees  throw;  }  }  return \*this;  }  }; |

| **Noncompliant Code 2** |
| --- |
| we have a class that does not check for self-assignment in its copy assignment operator, which can lead to problems when the same object is assigned to itself. |
| public:  char\* data;  String(const char\* str) {  data = new char[strlen(str) + 1];  strcpy(data, str);  }  ~String() {  delete[] data;  }  // Non-compliant: No self-assignment check  String& operator=(const String& other) {  if (data) {  delete[] data; // Free existing resource  }  data = new char[strlen(other.data) + 1];  strcpy(data, other.data);  return \*this;  }  void print() const {  std::cout << data << std::endl;  }  }; |

| **Compliant Code 2** |
| --- |
| write a compliant version that properly handles self-copy assignment by adding a check at the beginning of the copy assignment operator. |
| public:  char\* data;  String(const char\* str) {  data = new char[strlen(str) + 1];  strcpy(data, str);  }  ~String() {  delete[] data;  }  // Compliant: Self-assignment check  String& operator=(const String& other) {  if (this != &other) { // Check for self-assignment  delete[] data; // Free existing resource  data = new char[strlen(other.data) + 1];  strcpy(data, other.data);  }  return \*this;  }  void print() const {  std::cout << data << std::endl;  }  }; |

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

| **Principles(s):** User-provided copy operators must properly handle self-copy assignment. (principle 4, 5 and 10 applies here) |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2024.2 | C++4072, C++4073, C++4075, C++4076 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-OOP54-a | Check for assignment to self in operator= |
| Polyspace Bug Finder | R2024a | CERT C++: OOP54-CPP | Checks for copy assignment operators where self-assignment is not tested (rule partially covered) |
| CodeSonar | 8.1p0 | **IO.DC** **ALLOC.DF** **ALLOC.LEAK** **LANG.MEM.NPD** **LANG.STRUCT.RC** **IO.UAC** **ALLOC.UAF** | Double Close Double Free Leak Null Pointer Dereference Redundant Condition Use After Close Use After Free |

### 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.

The image below shows a visual representation of automation practices. In the image, one can see a clear demonstration of a typical Devops/DevsecOps. In the illustration, a continuous integration and continuous delivery/protection system is shown. This pathway shown is type of pathway that development teams follow to ensure a smooth development and upgrade process for their various projects. The process starts with Assessing and planning the target project or upgrade. Then following the process is the design, building, verifying and test of product. After this step, the security aspect kicks off. During this process, the product and its environment is maintained and checked for attacks, and if any vulnerabilities are found, the are quickly corrected. Also, during this stage, the product and its environment is monitored to detect threats, and penetration test are conducted. Depending on decisions, there maybe need to plan, build and deploy an upgrade as needed.



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.

Explanation:

The implementation of DevSecOps plays a vital role in developing secure applications, ensuring that security is integrated seamlessly throughout the software development lifecycle. By leveraging automation, organizations can significantly enhance this process. It is essential to adhere to the policies and standards outlined in the security policy document during the DevSecOps workflow. One of the primary phases where automation proves to be invaluable is in the build, verify, and test steps of the development process. Continuous Integration and Continuous Deployment (CI/CD) pipelines can be utilized to automate not only the construction of the application but also its testing phase. This automation fosters a more secure environment by ensuring that consistent procedures are followed during deployment. Additionally, it allows for a thorough examination of the pipeline's security measures, which helps in identifying and mitigating potential vulnerabilities early in the development process. By incorporating these practices, teams can create applications that are not only functional but also robustly secure.

Automated tools integrated into the CI/CD pipeline can execute and report on various types of tests, including unit tests, integration tests, front-end UI tests, and end-to-end tests developed for the application. Furthermore, these tools can perform static analysis to identify common coding issues or potential security threats, as well as assess dependency vulnerabilities. After the building and testing phases are complete, the transition and health check stage can leverage the CI/CD pipeline to automate the secure configuration and deployment of the application. After deployment, various tools can automatically conduct penetration testing, evaluating the application against a range of common exploits, including those identified in the coding standards.

Additional automated tools can capture log outputs from the application and monitor for any unusual events. If issues are detected in the logs or any events trigger alerts, notifications can be sent to the relevant personnel, initiating other automated responses. These responses may include tools in the incident response phase that manage intrusions by limiting or cutting off access. Finally, during the maintenance and stabilization phase, automated tools can be utilized to ensure the system's integrity and continue monitoring for new issues. This ongoing vigilance may also involve tracking dependency vulnerabilities, which, if discovered, can be addressed by automatically updating the dependency to a secure version.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | L2 |
| STD-002-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-009-CPP | Low | Likely | Medium | P6 | L2 |
| STD-010-CPP | Low | Probable | High | P2 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |

|  |  |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the process of keeping data in an encrypted state while it is stored. This means that even if someone manages to gain access to the data, it remains unreadable without the key required to decrypt it. Implementing this policy is crucial, as it ensures that if a breach occurs and data is stolen, the information contained within remains secure and cannot be interpreted without the decryption key. This type of encryption is particularly suitable for backup data that is stored away for extended periods. |
| Encryption in flight | Encryption in transit refers to the practice of keeping data encrypted while it travels from one location to another. During its journey, it is essential for the data to remain encrypted. This approach ensures that if the data is intercepted, the information contained within remains secure, as it cannot be read without the appropriate key. Such encryption is commonly implemented in various internet protocols, including HTTPS, FTP, and SMTP. |
| Encryption in use | Encryption in use refers to the practice of encrypting data while it is being actively utilized by the system. This approach is essential for maintaining data security at all times, even as it undergoes changes. As more data is secured through encryption at rest and in transit, the possibility of exploitation increases for data that remains unencrypted during use, making it the weakest link. Therefore, it is crucial to find solutions that allow for data to remain encrypted while being utilized. This methodology is commonly implemented in applications, where certain components may be encrypted to ensure that users do not gain access to sensitive information. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user's identity to ensure that they are who they claim to be. This procedure is commonly employed when adding new users or when a user attempts to log in. This policy is applicable whenever a user seeks access to a system or any specific element within it. It is essential to confirm that the individual accessing the system is indeed the expected user. This verification is typically achieved by validating user identification information, such as a username and password. |
| Authorization | Authorization is the process of verifying that a user is entitled to access the resources they are requesting. It establishes the user level of access that a user has within the system. This policy is applicable whenever a user attempts to access a system or any of its components. Once a user has been authenticated, authorization determines whether they are granted entry to the system or specific items within it. This policy is also responsible for making sure that the right user or administrator is the right person authorized to set and modify authorization features for users, when it comes to changing user privileges or when it comes to the addition of new users. This mechanism helps prevent unauthorized users from accessing restricted areas, ensuring that only individuals with the appropriate permissions can enter. |
| Accounting | Accounting is the process of monitoring and recording all requests and transactions made by a user. Examples include changes made to a database or files accessed by a user. This policy is essential for maintaining awareness of system activities. For instance, if the system begins to behave unexpectedly, you can review the logs to identify the issue and work towards finding a solution. |

**\***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 | 12/15/2023 | Completion of The Ten Core Security Principles, Defense in Depth, and Coding Standard | Francis Ugorji | Francis Ugorji |
| 1.2 | 12/20/2023 | Rounded up Coding standards. Also Completed the Summary of Risk Assessment table | Francis Ugorji | Francis Ugorji |
| 1.3 | 12/23/2023 | Completed the Coding Standards, Automation, Encryption and Triple A policies | Francis Ugorji | Francis Ugorji |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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