**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 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** | **Description** |
| --- | --- |
| 1. ValidateInput Data | Verify that input values from all external sources match the intended data type. Input values that do not match the intended data type can cause undefined behavior in the program. |
| 1. Heed Compiler Warnings | A program will still run with compiler warnings, but they need to be reviewed. They indicate unusual conditions that could lead to a problem in the code base (IBM, 2021). |
| 1. Architect and Design for Security Policies | Reviews how the securities and safeguards are implemented in a system based on business requirements. This involves the organization’s IT network which includes the computer systems and the data. |
| 1. Keep It Simple | Complex software results in technical debt. By only keeping the necessary security defined by the business requirements, the organization can prevent unnecessary costs with implementation, maintenance, and upgrades (Naor, 2020). |
| 1. Default Deny | This model focuses on restricting access by default to all users. Activities can only be accessed by authorization and explicit permissions (Akyoo, 2023). |
| 1. Adhere to the Principle of Least Privilege | Each user is only given minimal access to system resources and authorization they need to perform a function (NIST, 2023). |
| 1. Sanitize Data Sent to Other Systems | All values from an outside source are considered tainted. All values should be checked to ensure they conform to the required constraints especially before sending to other systems (Seacord, 2013). |
| 1. Practice Defense in Depth | A security strategy that utilizes variable barriers across multiple layers to prevent vulnerability from multiple exploits. The intent is to ensure that attacks missed by one barrier will be caught by another (NIST, 2023). |
| 1. Use Effective Quality Assurance Techniques | There are many different quality assurance techniques available. Utilize current best practice when determining which techniques to include on a project. Testing must be thorough and complete to prevent vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are rules used to follow to properly develop a program based on the organization’s policy and principles (SNHU, 2023). |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | No ambiguous declarations – this prevents the compiler from trying to infer the variables data type which can lead to undefined behavior (Carnegie Mellon, 2018). |

| **Noncompliant Code** |
| --- |
| This code is syntactically ambiguous since the declaration of the local variable, n, could either be a function declaration or variable declaration. When the program is compiled, the constructor is not invoked and does not print output. |
| struct Name {  Name(){ std::cout << “New Name” << std::endl; }  };  void changeName() {  Name n();  } |

| **Compliant Code** |
| --- |
| By removing the parentheses from local variable n, it ensures variable declaration by the compiler. |
| struct Name {  Name(){ std::cout << “New Name” << std::endl; }  };  void changeName() {  Name n;  } |

| **Principles(s):** 1, 2. Ambiguous variables can cause undefined behavior in an application. Verify that the instantiation from compilation or input will not create an ambiguous variable. If this is flagged by the compiler, review the code for improvements. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |
| Helix QAC | 2023.3 | C++1109, C++2510 |  |
| Klocwork | 2023.3 | CERT.DCL.AMBIGUOUS\_DECL |  |
| LDRA tool suite | 9.7.1 | 296 S | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | * Parameter names in function declarations should not be enclosed in parentheses. * Local variable names in variable declarations should not be enclosed in parentheses. * Avoid function declarations that are syntactically ambiguous. |
| Polyspace Bug Finder | R2023b | CERT C++: DCL53-CPP | Checks for declarations that can be confused between:   * Function and object declaration * Unnamed object or function parameter declaration   Rule fully covered. |
| Clang | 3.9 | -Wvexing-parse |  |
| SonarQube C/C++ Plugin | 4.10 | S3468 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Value check – verify the values from declarations and input meet the requirements of the data type. Unchecked variables are susceptible to overflow/underflow errors that can cause undefined behavior. |

| **Noncompliant Code** |
| --- |
| Since the temp variable is not checked after input, it has the potential to be invalid. This could cause a runtime error if not corrected. Also, the value is not compared to the number variable and could be greater than it. Since this is an unsigned integer, the value will wrap and not be correct. |
| int main() {  unsigned int number = 50;  unsigned int temp;  std::cin >> temp;  number –= temp;  } |

| **Compliant Code** |
| --- |
| The input will now be checked to make sure it is the correct data type before continuing. The temp variable is now compared to the number it is subtracting from. This will catch if the value causes an underflow. |
| int main() {  unsigned int number = 50;  unsigned int temp;  std::cin >> temp;  while (!std::cin.good()) {  std::cin.clear();  std::cin >> temp;  }    if (number < temp) {  negative();  }  else {  number -= temp;  }  } |

| **Principles(s):** 1. All input must be validated by the application. Erroneous or malicious input can cause undefined behavior if not properly checked. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 8.0p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | * Addition overflow of allocation size * Integer overflow of allocation size * Multiplication overflow of allocation size * Subtraction underflow of allocation size * Addition overflow of size * Unreasonable size argument * Multiplication overflow of size * Subtraction underflow of size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Out of bounds check - Strings are implemented as char arrays and susceptible to the same problems as arrays such as overflow. All string input must have their length verified prior to implementation. |

| **Noncompliant Code** |
| --- |
| This code is susceptible to overflow since it does not check the input size and uses a relatively low array size of 10. |
| void changeName() {  char name[10];  std::cin >> name;  } |

| **Compliant Code** |
| --- |
| This code increases the array size for the variable name to a reasonably expected size. Next, it creates a local std string variable to accept the input and compare it to the name variable. If the input is not within the name variable’s parameters, the program will try again. |
| void changeName() {  char name[30];  std::string temp; // max size of 4,294,967,291    std::cin >> temp;  while (temp.size() > sizeof(name)) {  std::cin.clear();  std::cin >> temp;  }  strcpy\_s(name, sizeof(name), temp.c\_str())  } |

| **Principles(s):** 2, 10. A compiler may catch a possibility for out of bounds errors. Review the code to look for improvements. Implementing a secure coding standard related to verifying string correctness can remind developers to verify size when writing to a char array. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 8.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | * Avoid overflow due to reading a not zero terminated string. * Avoid overflow when writing to a buffer. * Prevent buffer overflows from tainted data. * Avoid buffer write overflow from tainted data. * Do not use the 'char' buffer to store input from 'std::cin'. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Injection prevention – SQL injection is a common exploit. Passing the query directly to the database could cause a breach in secure information. The input query needs to be scrubbed first. |

| **Noncompliant Code** |
| --- |
| This code does not scrub the input string prior to appending it to the sql query string. The name variable could potentially have malicious code that needs to be checked. |
| void getQuery() {  std::string name;  std::string sql = “SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME=”;  std::cin >> name;  sql.append(name);  } |

| **Compliant Code** |
| --- |
| This code checks the input prior to adding it to the sql query string. The integer found will locate the end of the name. the if statement checks if additional parameters have been passed in an ‘or’ statement. If it is found, the local name is changed to flag it. |
| void getQuery() {  std::string name;  std::string sql = “SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME=”;  std::cin >> name;  int found = name.find(‘\’’, 1) // find end of the name  if (name.find(“or”, found) != -1) { // check for ‘or’ after name  name = “PossibleInjection”  }  sql.append(name);  } |

| **Principles(s):** 5, 6, 7, 9, 10. SQL injection is a known security attack that must be anticipated. Default deny and Principle of Least Privilege should limit access to database results. All data that is being sent from the application must be sanitized for correctness, especially when the data originated from an external source. Testing measures should be in place to check for this common attack. Utilizing a secure coding standard can help developers prevent malicious attacks from SQL injection. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 8.0p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Deallocation - Memory that has been deallocated by a memory management function will have undefined behavior. Accessing pointers to deallocated memory can result in vulnerabilities (Carnegie Mellon, 2018). |

| **Noncompliant Code** |
| --- |
| This code shows a linked list being deleted. During the deletion process node n is freed before n->next is executed. This makes n a dangling pointer which can cause vulnerabilities. |
| void delete(struct node \*head) {  for (struct node \*n = head; n != NULL; n = n->next) {  free(n);  }  } |

| **Compliant Code** |
| --- |
| This code shows a linked list being deleted. During the deletion process node m stores the reference to the next pointer location before node n is deleted. |
| void delete(struct node \*head) {  struct node \*m  for (struct node \*n = head; n != NULL; n = m) {  m = n->next;  free(n);  }  } |

| **Principles(s):** 9, 10. Dangling pointers may not be immediately noticed if the program compiles. Proper testing techniques such as unit testing can catch these vulnerabilities. Adopting a secure coding standard can prevent developers from creating an instance of a dangling pointer. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | invalid\_dynamic\_memory\_allocation  dangling\_pointer\_use |  |
| Clang | 3.9 | * clang-analyzer-cplusplus.NewDeleteLeaks-Wmismatched-new-delete * clang-analyzer- unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.0p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| LDRA tool suite | 2023.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assert Expression – the assertion statements only compile in debug. In release versions it is treated as null. Verify is an alternative statement that is safe to use since it is not checked in release (Microsoft, 2023). |

| **Noncompliant Code** |
| --- |
| Since assertion statements will have different values in debug and release, it is unsafe to use since it could cause undefined behavior. |
| int main {  ASSERT(value > 0);  } |

| **Compliant Code** |
| --- |
| The verify statement is not checked in the release version. \*Do not call a function in the verify statement because it will still run causing an increase in time complexity. |
| int main {  VERIFY(value > 0);  } |

| **Principles(s):** 4, 9, 10. In production use a simple if statement if the value must be checked or change assert to verify if not. This can prevent unexpected errors that could cause technical debt from refactoring. Assertions can be useful when writing tests for code. Proper usage based on secure coding standards can promote the use of assertion in testing will preventing its use in production. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  | Can detect some violations of this rule. However, it can only detect violations involving abort() because assert() is implemented as a macro |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced enforcement |
| Parasoft C/C++test | 2023.1 | CERT\_C-ERR06-a | Do not use assertions |
| PC-lint Plus | 1.4 | 586 | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exception handling – To prevent stopping a program due to an error, the condition should use the try/catch method. Recent depreciation of the throw( type-name ) means the preferred alternative is throw, throw(), or noexcept (Microsoft, 2022). |

| **Noncompliant Code** |
| --- |
| In this code, the function in main is passing an invalid argument. In the func() function an error is thrown. Since main does not contain a try/catch method, the program will terminate. |
| void func(int a) {  if (invalid) {  throw invalid\_arg(“Too Large”);  }  }  int main() {  func(1); // invalid argument  } |

| **Compliant Code** |
| --- |
| In this code, the function in main is passing an invalid argument. In the func() function an error is throw. Since main has a try/catch method, the program will continue and output the error message to the command line interface. |
| void func(int a) {  if (invalid) {  throw invalid\_arg(“Too Large”);  }  }  int main() {  try {  func(1);  }  Catch (invalid\_arg& e) {  Std::cerr << e.what() << std::endl;  return -1;  }  } |

| **Principles(s):** 1, 10. Validating data is important, but so is understanding why there is an issue. If input is not valid, a statement should be thrown to provide a reference to the issue. Secure coding standards should be followed for proper implementation as well as what messages need to be returned. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2023b | CERT C: Rec. ERR00-C | Checks for situations where error information is not checked (rec. partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declaration | [STD-008-CPP] | Appropriate Storage – Attempting to access an object outside of its lifetime will cause undefined behavior which creates vulnerability (Carnegie Mellon, 2018). |

| **Noncompliant Code** |
| --- |
| This code violates the appropriate storage rule. Variable n will go out of scope since it is a local variable and m will still hold the address. |
| const char \*m  void func() {  const char n[] = “name change”;  m = n;  } |

| **Compliant Code** |
| --- |
| This code is compliant since variable m is also a local variable and is not available outside of the func() scope. |
| void func() {  const char n[] = “name change”;  const char \*m = n;  } |

| **Principles(s):** 9, 10. Invalid pointers may not be immediately noticed if the program compiles. Proper testing techniques such as unit testing can catch these vulnerabilities. Adopting a secure coding standard can prevent developers from creating an instance of an invalid pointer. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | pointered-deallocation  return-reference-local | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL30 | Fully implemented |
| CodeSonar | 8.0p0 | LANG.STRUCT.RPL | Returns pointer to local |
| Coverity | 2017.07 | RETURN\_LOCAL | Finds many instances where a function will return a pointer to a local stack variable. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory protection | [STD-009-CPP] | Null Dereference – null pointer dereference is not recognized and will cause a crash or exit (OWASP, 2024). |

| **Noncompliant Code** |
| --- |
| This shows a pointer being freed without checking that it is NULL beforehand. This will terminate the program causing vulnerabilities. |
| free(pointer) |

| **Compliant Code** |
| --- |
| This shows an if statement checking if the pointer is null. If it is, the pointer is freed and set to NULL. |
| if (pointer != NULL) {  free(pointer);  pointer = NULL;  } |

| **Principles(s):** 9, 10. Null pointer dereference may not be immediately noticed if the program compiles. Proper testing techniques such as unit testing can catch these vulnerabilities. Adopting a secure coding standard can prevent developers from creating an instance of a null pointer dereference. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Null-dereferencing | Fully checked |
| CodeSonar | 8.0p0 | LANG.MEM.NPD  LANG.STRUCT.NTAD  LANG.STRUCT.UPD | Null pointer dereference  Null test after dereference  Unchecked parameter dereference |
| Coverity | 2017.07 | CHECKED\_RETURN  NULL\_RETURNS  REVERSE\_INULL  FORWARD\_NULL | * Finds instances where a pointer is checked against NULL and then later dereferenced. * Identifies functions that can return a null pointer but are not checked. * Identifies code that dereferences a pointer and then checks the pointer against NULL. * Can find the instances where NULL is explicitly dereferenced or a pointer is checked against NULL but then dereferenced anyway. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary. |
| Cppcheck | 1.66 | nullPointer, nullPointerDefaultArg, nullPointerRedundantCheck | * Context sensitive analysis. * Detects when NULL is dereferenced (Array of pointers is not checked. Pointer members in structs are not checked.) * Finds instances where a pointer is checked against NULL and then later dereferenced. * Identifies code that dereferences a pointer and then checks the pointer against NULL. * Does not guess that return values from malloc(), strchr(), etc., can be NULL (The return value from malloc() is NULL only if there is OOMo and the dev might not care to handle that. The return value from strchr() is often NULL, but the dev might know that a specific strchr() function call will not return NULL.) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | [STD-010-CPP] | Division – Division operations must be checked to prevent divide-by-zero errors (Carnegie Mellon, 2018). |

| **Noncompliant Code** |
| --- |
| This shows variables a being divided by variable b. Without a proper check, it is susceptible to a divide-by-zero error. |
| void divide(int a, int b) {  result = a / b;  } |

| **Compliant Code** |
| --- |
| This shows variables a being divided by variable b. Prior to division an if statement is used to make sure the denominator is not equal to zero. This will prevent a divide-by-zero error. |
| void divide(int a, int b) {  if (b != 0) {  result = a / b;  }  } |

| **Principles(s):** 1, 2, 9, 10. Divide by zero may not be immediately noticed if the program compiles. The function must validate the input data prior to performing an operation. The compiler may provide a compiler warning for this error, and it should be reviewed. Proper testing techniques such as unit testing can catch these vulnerabilities. Adopting a secure coding standard can prevent developers from creating an instance of divide by zero error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | int-division-by-zero  int-modulo-by-zero | Fully checked |
| CodeSonar | 8.0p0 | LANG.ARITH.DIVZERO  LANG.ARITH.FDIVZERO | Division by zero  Float Division By Zero |
| Coverity | 2017.07 | DIVIDE\_BY\_ZERO | Fully implemented |
| Cppcheck | 1.66 | zerodiv  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. |

### Defense-in-Depth Illustration

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



## Project One

### 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.

Automation is an excellent way to implement coding standards by utilizing static analysis. Resources such as Astree, Cppcheck, and SonarSource provide lightweight technology that helps developers deliver clean and secure code with the analysis review. The automation will be included in the design and verify and test phases of Green Pace’s DevOp process. This will allow developers to catch areas for improvement before reaching the production cycle. This will limit the amount of refactoring required and reduce the overall technical debt.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-007-CPP | Medium | Probable | High | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Likely | Medium | P18 | L1 |
| STD-0010-CPP | Low | Likely | Medium | P6 | L2 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest protects data that is stored on a disk. All data at rest should be encrypted using a standard best practice Advanced Encryption Standard (AES) algorithm. This will prevent access to the data. If the data is breached, it will be encrypted, and the attacker will be unable to read it. |
| Encryption at flight | Encryption in flight protects data that is being sent over a network. All data that is sent over the network will be encrypted with AES and have a key the receiver will use to decrypt the message. This prevents someone from intercepting the message and reading the sensitive information. |
| Encryption in use | Encryption in use protects data that is being accessed, processed, read, updated, or erased. Data in use will be encrypted with AES and prevent unauthorized access in real time. Adding encryption in use will protect the data throughout the entire lifecycle and prevent traditional vulnerabilities such as cloud infrastructure and unsecure endpoints. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication stands for how users are identified. This usually involves unique information such as usernames, passwords, certificates, and keys. Secure authentication will prevent malicious users from gaining access to the system. |
| Authorization | Authorization defines the level of privilege that a user has. A customer should have less authorization than an administrator but both profiles should adhere to the Principle of Least Privileged. A user with lower authorization would be able to send requests to the database. A user with higher authorization would be able to make changes to the database and add new users. |
| Accounting | Accounting tracks the user’s actions in the system. This provides an audit trail for review as needed. If an incident were to occur, Green Pace would be able to track which files were accessed and what parts of the application were used at what time. This is useful for a retrospective view to determine vulnerabilities. |

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 | 1/26/2024 | Updated:  Ten Core Security Principles  Data Type Coding Standard  Data Value Coding Standard  String Correctness Coding Standard  SQL Injection Coding Standard  Memory Protection Coding Standard  Assertions Coding Standard  Exceptions Coding Standard  Additional Standard 8  Additional Standard 9  Additional Standard 10 | Jeremy Baldner | Prof. Sarah North |
| 1.2 | 2/18/24 | Updated:  Ten Core Security Principles with a risk assessment and automation tool.  Automation section.  Summary of Risk Assessments.  Create policies for encryption and triple A. | Jeremy Baldner | Prof. Sarah North |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

# References

Akyoo, G. (2023). *Fundamental Security Postures: Default Allow vs. Default Deny*. Retrieved from Medium: https://medium.com/@gakyoo/fundamental-security-postures-default-allow-vs-default-deny-539abfbbdba9#:~:text=The%20default%20deny%20model%20is,authorized%20and%20explicitly%20permitted%20activities.

Carnegie Mellon. (2018). *SEI CERT C Coding Standard.* Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

IBM. (2021). *Compiler Warnings*. Retrieved from IBM: https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=warnings-compiler

Microsoft. (2022). *Modern C++ best practices for exceptions and error handling.* Retrieved from Microsoft: https://learn.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=msvc-170

Microsoft. (2023). *C/C++ Assertions.* Retrieved from Microsoft: https://learn.microsoft.com/en-us/visualstudio/debugger/c-cpp-assertions?view=vs-2022

Naor, A. (2020). *Keep it Simple, Stupid – How to Use the KISS Principle in Design*. Retrieved from Free Code Camp: https://www.freecodecamp.org/news/keep-it-simple-stupid-how-to-use-the-kiss-principle-in-design/

NIST. (2023). *defense-in-depth*. Retrieved from NIST: https://csrc.nist.gov/glossary/term/defense\_in\_depth

NIST. (2023). *least privilege*. Retrieved from NIST: https://csrc.nist.gov/glossary/term/least\_privilege

OWASP. (2024). *Null Dereference*. Retrieved from OWASP: https://owasp.org/www-community/vulnerabilities/Null\_Dereference#:~:text=CWE%2D476%3A%20NULL%20Pointer%20Dereference,causing%20a%20crash%20or%20exit.

Seacord, R. (2013). *Secure Coding in C and C++.* Pearson Education, Inc.

SNHU. (2023). *The Building Blocks of the Security Policy.* Retrieved from SNHU: https://learn.snhu.edu/content/enforced/1461064-CS-405-R3362-OL-TRAD-UG.24EW3/course\_documents/CS%20405%20Building%20Blocks%20of%20the%20Security%20Policy%20Infographic.pdf?ou=1461064