**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 9](#_Toc52464061)

[Coding Standard 4 12](#_Toc52464062)

[Coding Standard 5 15](#_Toc52464063)

[Coding Standard 6 18](#_Toc52464064)

[Coding Standard 7 20](#_Toc52464065)

[Coding Standard 8 22](#_Toc52464066)

[Coding Standard 9 24](#_Toc52464067)

[Coding Standard 10 26](#_Toc52464068)

[Defense-in-Depth Illustration 31](#_Toc52464069)

[Project One 31](#_Toc52464070)

[1. Revise the C/C++ Standards 31](#_Toc52464071)

[2. Risk Assessment 31](#_Toc52464072)

[3. Automated Detection 31](#_Toc52464073)

[4. Automation 31](#_Toc52464074)

[5. Summary of Risk Assessments 32](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 32](#_Toc52464076)

[7. Map the Principles 33](#_Toc52464077)

[Audit Controls and Management 35](#_Toc52464078)

[Enforcement 35](#_Toc52464079)

[Exceptions Process 35](#_Toc52464080)

[Distribution 36](#_Toc52464081)

[Policy Change Control 36](#_Toc52464082)

[Policy Version History 36](#_Toc52464083)

[Appendix A Lookups 36](#_Toc52464084)

[Approved C/C++ Language Acronyms 36](#_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 | Validation of inputted data involves checking to make sure that only properly formatted data enters the system or database. This will happen when any data is received from a user or any other external party. Doing thorough input validation for all data will prevent errors from being triggered in the system, and guard against bad data or any errors happening in the database. Validating input data will also help to guard against SQL injection attacks and is one step in keeping the data and database safe. |
| 1. Heed Compiler Warnings | When compiling code, the warnings that show will be looked at and the code will be modified to eliminate this warning. This will help with eliminating any security vulnerabilities or the potential for a warning to turn into a bug or a risk in the future. |
| 1. Architect and Design for Security Policies | Software will be designed with security in mind and when building out the architecture and design for the code, security will be implemented at all levels. Things such as creating different roles and using the principle of least privilege or dividing the system into subsystems that each have different access levels will be looked at and the best method with the highest possible level of security for the software will be used. |
| 1. Keep It Simple | The design of the application will be kept simple and maintainable. When the design is cohesive throughout and simple, it is easier to maintain the code, easier to update the code, and creates an environment where the is less likelihood for mistakes to be made. The code will also be less confusing and will be easier to configure and implement if the design is kept simple. |
| 1. Default Deny | At the default level, any sort of permissions or access will deny entry at their base level. Things such as roles or permissions will be extra and added levels of access to the system and will not be exclusions. As a user is given more roles, for instance, they will gain more access, and if these roles are taken away, they will lose access because the default behavior of the system will be to deny users’ access to areas of the system. |
| 1. Adhere to the Principle of Least Privilege | Users will be given only access to the exact parts of the system that they need. The roles that are given to the users will only include access to parts of the system that allow them to functionally perform their roles and will not include any extra access. If a user requires different access at different times, perhaps different parts of the system need to be subdivided and the user needs to be given different access levels to each part of the system so that they do not have extra access beyond what is necessary. |
| 1. Sanitize Data Sent to Other Systems | Before sending any data to other systems, the data should be both validated and cleaned. Even though the data may not be intended to attack the current system, there could be something in it, such as a SQL injection attack, a command, or other sort of attack that could be meant for another system, so data that enters the system needs to be sanitized before passing along to other systems. |
| 1. Practice Defense in Depth | Multiple security and defense strategies at different levels that overlap will be utilized. This is so if one layer fails, has a flaw or hole, or is found to be inadequate in some other way, other layers will be able to hold up and prevent some sort of vulnerability in the code from the flaw in one layer of security. Having multiple security levels will ensure that if a security flaw is trying to be exploited at one layer, another layer will be able to keep the system secure as it will be able to block this threat successfully. |
| 1. Use Effective Quality Assurance Techniques | Making sure that the code and system is regularly tested and thoroughly tested before deploying any changes will help to make sure that errors, bugs, nor security vulnerabilities are introduced. Strategies such as writing good unit tests, using penetration testing, integration tests, and source code reviews and audits help to make sure the code is of good quality and that the system will be safe when the code is deployed. |
| 1. Adopt a Secure Coding Standard | Coding securely using secure patterns and practices during the development process will help keep the code secure. This means that security is not an afterthought in development that is added later in the process, but something that is designed and then coded and implemented throughout the development process so that the security of the application is kept in focus during development. |

### 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** | DCL52-CPP | Never qualify a reference type with const or volatile |
| **Rationale** | | |
| When a reference value is declared as either const or volatile, this will result in undefined behavior, and will also show a warning in Visual Studio since the constant qualifier will be ignored, therefore it should be removed as to not cause any undefined behavior and to get rid of warnings that are thrown in the compiler. | | |

| **Noncompliant Code** |
| --- |
| In the below example, p is declared as a const-qualified reference to a char instead of a reference to a const-qualified char, this will result in undefined behavior since the cv-qualifier will be ignored and in visual studio this will throw a warning that the const-qualifier will be ignored. |
| #include <iostream>    void f(char c) {  char &const p = c;  p = 'p';  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| In the below example, the const qualifier was removed. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

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

| **Principles(s):** One principle that this applies to is “#2 Heed Compiler Warnings” as if you declare a reference variable as const or volatile, the compiler will throw a warning for this, stating that this will result in undefined behavior. Another principle that can apply to this is “#9 Use Effective Quality Assurance Techniques” as this is not good coding practice and can cause code that has undefined or unpredictable behavior, and this should be caught while testing, and should be changed so that there is no undefined behavior, nor any sorts of errors or warnings thrown while the code is running or during the testing process. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 |  |
| Helix QAC | 2023.1 | C++0014 |  |
| Klocwork | 2023.1 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE | Never qualify a reference type with 'const' or 'volatile' |
| Polyspace Bug Finder | R2023a | CERT C++: DCL52-CPP | Checks for:   * const-qualified reference types * Modification of const-qualified reference types   Rule fully covered. |
| PRQA QA-C++ | 4.4 | 0014 |  |
| Clang | 3.9 |  | Clang checks for violation of this rule and produces an error without the need to specify any special flags or options |
| SonarQube C/C++ Plugin | 4.10 | S3708 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | EXP50-CPP | Do not depend on the order of evaluation for side effects |
| **Rationale** | | |
| Sometimes, the order of operations is unclear in a statement, and when this happens, the expression will result as being evaluated to undefined since the order of operations and the intent of the statement is unknown since there can be multiple solutions to the statement depending on the order it is evaluated. | | |

| **Noncompliant Code** |
| --- |
| In the below example, since i is evaluated more than once without any sort of sequence, the expression will be undefined. The order of operations here is unclear and will cause the expression to be undefined since there are two possible ways for the order of operations to be interpreted. |
| void f(int i, const int \*b) {  int a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| There are two possible solutions, depending on what the order of operations was intended to be. With either of these solutions the order is very clear and can only be interpreted in one way. |
| void f(int i, const int \*b) {  ++i;  int a = i + b[i];  // ...  }  // OR, second solution depending on intended order of operations:  void f(int i, const int \*b) {  int a = i + b[i + 1];  ++i;  // ...  } |

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

| **Principles(s):** One principle that applies is “#4 Keep it Simple.” When writing complicated and unclear code, it makes it very hard to maintain and update. The non-compliant solution is not clear, and there is no way to know what the correct way to interpret or evaluate that statement is, so by keeping it simple and making sure that the code is very clearly written and easy to interpret and maintain, this will be avoided. Another principle that applies is “#10 Adopt a Secure Coding Standard.” This one applies because it is not a secure practice to write statements that have an unclear order of operations, since this could lead to it being interpreted in an unintended way and lead to a security threat or a ripple effect of errors in the code down the line. Making sure to make the code very clear and not rely solely on order of operations will ensure that the statement is only able to be interpreted in one way and will not throw any errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-EXP50** |  |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required |
| CodeSonar | 7.3p0 | **LANG.STRUCT.SE.DEC** **LANG.STRUCT.SE.INC** | Side Effects in Expression with Decrement Side Effects in Expression with Increment |
| Compass/ROSE |  |  | Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator |
| Coverity | v7.5.0 | **EVALUATION\_ORDER** | Can detect the specific instance where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently |
| ECLAIR | 1.2 | **CC2.EXP30** | Fully implemented |
| GCC | 4.9 |  | Can detect violations of this rule when the -Wsequence-point flag is used |
| Helix QAC | 2023.1 | **C++3220, C++3221, C++3222, C++3223, C++3228** |  |
| Klocwork | 2023.1 | **PORTING.VAR.EFFECTS** **CERT.EXPR.PARENS** **MISRA.EXPR.PARENS.INSUFFICIENT** **MISRA.INCR\_DECR.OTHER** |  |
| LDRA tool suite | 9.7.1 | **35 D, 1 Q, 9 S, 134 S, 67 D, 72 D** | Partially implemented |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-EXP50-a** **CERT\_CPP-EXP50-b** **CERT\_CPP-EXP50-c** **CERT\_CPP-EXP50-d** **CERT\_CPP-EXP50-e** **CERT\_CPP-EXP50-f** | The value of an expression shall be the same under any order of evaluation that the standard permits Don't write code that depends on the order of evaluation of function arguments Don't write code that depends on the order of evaluation of function designator and function arguments Don't write code that depends on the order of evaluation of expression that involves a function call Between sequence points an object shall have its stored value modified at most once by the evaluation of an expression Don't write code that depends on the order of evaluation of function calls |
| Polyspace Bug Finder | R2023a | CERT C++: EXP50-CPP | Checks for situations where expression value depends on order of evaluation (rule fully covered). |
| PRQA QA-C++ | 4.4 | **3220, 3221, 3222, 3223, 3228** |  |
| PVS-Studio | 7.24 | **V521**, **V708** |  |
| Splint | 5.0 |  |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator |
| **Rationale** | | |
| If there is not enough space for character data in a character array when allowing a user to put input into this array, when data is inputted, this will cause a buffer overflow if they enter too many characters. To solve for this, you can use a string or limit the characters allowed in the std::cin statement. | | |

| **Noncompliant Code** |
| --- |
| In this example below, since the input is not bounded, meaning there is no limit to how many characters are entered into the cin statement, this could lead to a buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Instead of using a char[] which is bounded, to guard against buffer overflows use a std::string for inputs to make sure that the data is not truncated nor will cause a buffer overflow. |
| #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):** One principle that applies for this standard is “#10 Adopt a Secure Coding Standard.” When using a char[], this can cause a buffer overflow if the input is greater than can fit in the array, so by using a string, this is making the code more secure by preventing an error from happening. Another principle that applies is “#1 ValidateInput Data.” When gathering input data from a user and allowing them to put data that will enter the ecosystem of the system or application, it is important to make sure that you are using a data format for the input that will not cause errors. By using a string instead of a char[] for data that is not limited to a certain amount of characters, this will help with validating the data, and will keep an overflow error from happening. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| CodeSonar | 7.3p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO** **LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| Helix QAC | 2023.1 | **C++5216**  **DF2835, DF2836, DF2839,** |  |
| Klocwork | 2023.1 | **NNTS.MIGHT** **NNTS.TAINTED** **NNTS.MUST** **SV.UNBOUND\_STRING\_INPUT.CIN** |  |
| LDRA tool suite | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| Parasoft C/C++test | 2022.2 | **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' |
| Polyspace Bug Finder | R2023a | CERT C++: STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| RuleChecker | 22.10 | **stream-input-char-array** | Partially checked |
| SonarQube C/C++ Plugin | 4.10 | **S3519** |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS00-J | Prevent SQL Injection |
| **Rationale** | | |
| Guarding against SQL injection is important so as to not allow users to enter a “OR 1=1” type of statement and return all data in the database. If SQL injection is not guarded against, this can cause data from the database to be exposed to an unauthorized user. | | |

| **Noncompliant Code** |
| --- |
| In the below code, the username and password inputs are not sanitized before being added to the SQL statement string. The below example is vulnerable to a SQL injection attack since an attacker can inject something like “OR 1=1” into either the username or password field. |
| public void doPrivilegedAction(String username, string password) {  try {  String sqlString = "SELECT \* FROM db\_user WHERE username = '"  + username +  "' AND password = '" + pwd + "'";  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);    if (!rs.next()) {  throw new SecurityException(  "User name or password incorrect"  );  }  // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  } |

| **Compliant Code** |
| --- |
| The below example is not vulnerable to SQL Injection attacks because the username is checked for length and an error will be thrown if it is too long, and the password is hashed before being entered into the SQL string query. Further, the query is parameterized, and so ? are used as placeholders for the argument, and these placeholders are swapped for the username and password values to prevent an “OR 1=1” from being tagged onto the sql statement. |
| public void doPrivilegedAction(  String username, char[] password  ){  try {  String pwd = hashPassword(password);    // Validate username length  if (username.length() > 8) {  // Handle error  }    String sqlString =  "select \* from db\_user where username=? and password=?";  PreparedStatement stmt = connection.prepareStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, pwd);  ResultSet rs = stmt.executeQuery();  if (!rs.next()) {  throw new SecurityException("User name or password incorrect");  }    // Authenticated; proceed  } finally {  try {  connection.close();  } catch (SQLException x) {  // Forward to handler  }  } |

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

| **Principles(s):** The first is “#8 Practice Defense in Depth,” as this is one layer of security and in the defense of a system. By preventing SQL injection through using query parameterization, this is one defense method to stop potential attackers from accessing the database and getting information that they should not have access to. Another principle is “#7 Sanitize Data Sent to Other Systems.” By making sure that the data sent to the database is sanitized before sending it, this will prevent against injection attacks by not letting a query with something like “OR 1=1” or data that is in the incorrect format to query the database get to the database, so this will thwart an attacker’s chance at getting access to the data, or to crash the database. A third principle is “#10 Adopt a Secure Coding Standard.” By making sure to use query parameterization, this makes sure that security is added in at the code level during the coding process. This is one extra layer of security that is used in the coding process and is good practice to be coding securely when looking at the code to database connection layer. A fourth principle that applies here is “#1 Validate Input Data.” By taking the data that was inputted and validating it and making sure that you are precisely placing it as a string query so it is only being used to query the database in one possible way and not adding something like an “OR 1=1” to the query, this helps keep the database safe and secure through only allowing the correct types of queries to be ran on the database at any time. A fifth principle that applies is “#6 Adhere to the Principle of Least Privilege.” By preventing SQL injection attacks, this will ensure that users only are able to get the amount of data that they are authorized to get from the database and will not gain unauthorized access to the entire database. The sixth principle that applies is “#3 Architect and Design for Security Policies.” |
| --- |
| **(Principles, continued).** When looking at creating a connection between the code and database, it is important to make sure that the connection is kept secure, and that any risk of a user gaining access to data they are not authorized to have access to is stopped, so SQL Query Parameterization will guard against this happening. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | **Tainting Checker** | Trust and security errors (see Chapter 8) |
| CodeSonar | 7.3p0 | **JAVA.IO.INJ.SQL** | SQL Injection (Java) |
| Coverity | 7.5 | **SQLI** **FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_** **FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| Findbugs | 1.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** | Implemented |
| Fortify | 1.0 | **HTTP\_Response\_Splitting** **SQL\_Injection\_\_Persistence** **SQL\_Injection** | Implemented |
| Klocwork |  | **SV.DATA.BOUND** **SV.DATA.DB** **SV.HTTP\_SPLIT** **SV.PATH** **SV.PATH.INJ** **SV.SQL** | Implemented |
| Parasoft Jtest | 2022.2 | **CERT.IDS00.TDSQL** | Protect against SQL injection |
| SonarQube | 6.7 | **S2077**  **S3649** | Executing SQL queries is security-sensitive  SQL queries should not be vulnerable to injection attacks |
| SpotBugs | 4.6.0 | **SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE** **SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING** | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM51-CPP | Properly deallocate dynamically allocated resources |
| **Rationale** | | |
| Memory must be deallocated properly depending on how it was allocated. If it is not deallocated correctly, then the destructors are not called. This can cause undefined behavior since the memory was not deallocated correctly. | | |

| **Noncompliant Code** |
| --- |
| In the below code block, std::free() was used to deallocate memory for s which was allocated by using the new keyword. This is the incorrect deallocation function and not the correct way to deallocate s. This will cause undefined behavior since the destructors will not be called for s. |
| #include <cstdlib>    struct S {  ~S();  };    void f() {  S \*s = new S();  // ...  std::free(s);  } |

| **Compliant Code** |
| --- |
| In the below code block, the variable s that is created when using new is deallocated using delete which is the correct deallocation function as opposed to std::free(). |
| struct S {  ~S();  };    void f() {  S \*s = new S();  // ...  delete s;  } |

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

| **Principles(s):** One principle that applies to this standard is “#2 Heed Compiler Warnings.” Writing code in a non-compliant way will cause the compiler to throw a warning, so it should be fixed when a warning for not deallocating memory is revealed. Another principle that applies is “#10 Adopt a Secure Coding Standard.” This principle applies as when writing code and neglecting to deallocate memory, this can cause undefined behavior, which means that it could introduce some sort of error into the code or a bug, and when there are bugs and errors in the code, it is less secure, so the code should be written correctly and memory should be deallocated to avoid bugs or errors from being introduced into the code, as these bugs and errors can be exploited, causing a security threat. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | **invalid\_dynamic\_memory\_allocation** **dangling\_pointer\_use** |  |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MEM51** |  |
| 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 | 7.3p0 | **ALLOC.FNH** **ALLOC.DF** **ALLOC.TM** **ALLOC.LEAK** | Free non-heap variable Double free Type mismatch Leak |
| Helix QAC | 2023.1 | **C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264** |  |
| Klocwork | 2023.1 | **CL.FFM.ASSIGN** **CL.FFM.COPY** **CL.FMM** **CL.SHALLOW.ASSIGN** **CL.SHALLOW.COPY** **FMM.MIGHT** **FMM.MUST** **FNH.MIGHT** **FNH.MUST** **FUM.GEN.MIGHT** **FUM.GEN.MUST** **UNINIT.CTOR.MIGHT** **UNINIT.CTOR.MUST** **UNINIT.HEAP.MIGHT** **UNINIT.HEAP.MUST** |  |
| LDRA tool suite | 9.7.1 | **232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D** | Partially implemented |
| Parasoft C/C++test | 2022.2 | **CERT\_CPP-MEM51-a** **CERT\_CPP-MEM51-b** **CERT\_CPP-MEM51-c** **CERT\_CPP-MEM51-d** | Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2023a | CERT C++: MEM51-CPP | Checks for:   * Invalid deletion of pointer * Invalid free of pointer * Deallocation of previously deallocated pointer   Rule partially covered. |
| Astrée | 22.10 | **invalid\_dynamic\_memory\_allocation** **dangling\_pointer\_use** |  |
| Axivion Bauhaus Suite | 7.2.0 | **CertC++-MEM51** |  |
| 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 |
| PRQA QA-C++ | 4.4 | **2110, 2111, 2112, 2113, 2118,**  **3337, 3339, 4262, 4263, 4264** |  |
| PVS-Studio | 7.24 | **V515, V554, V611, V701, V748, V773, V1066** |  |
| SonarQube C/C++ Plugin | 4.10 | **S1232** |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL03-C | Use a static assertion to test the value of a constant expression |
| **Rationale** | | |
| If you are testing the value of a constant expression, this should be tested in a static function instead of a memory-mapped function so that these assumptions that are being asserted can be diagnosed at compile time and not throw a runtime error or silent malfunction. | | |

| **Noncompliant Code** |
| --- |
| This uses assert() to assert a property concerning memory mapped function but is not placed inside of a function and executed. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| In the example below, a static\_assert is used instead of assert() so that incorrect assumptions can be diagnosed at compile time instead of a silent malfunction or runtime error. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** One principle that applies here is “#10 Adopt a Secure Coding Standard.” By making sure that a static assertion is used at compile time instead of an assert() statement, this will make sure that errors are being thrown at compile time, rather than during runtime, and this will also help to catch any silent malfunctions. Another principle that applies is “#9 Use Effective Quality Assurance Techniques.” By making sure that assert() statements aren’t used, this will make the code higher quality and easier to test as static assert statements will run during compile time and be able to catch errors in the code before it runs or throws a silent malfunction, thus leading to code that throws less errors and is more secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL03** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP | Handle all exceptions |
| **Rationale** | | |
| When exceptions are not handled, this will cause the code to crash and terminate unexpectedly. They should be handled so that if one is thrown, it is caught and handled to not break the entire code and cause the program to terminate unexpectedly. | | |

| **Noncompliant Code** |
| --- |
| In the below example, the exceptions are not caught by the throwing\_func(). This will cause the thread function to exit due to an exception being thrown, and the thread will be terminated using std::terminate(). |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start() {  throwing\_func();  }    void f() {  std::thread t(thread\_start);  t.join();  } |

| **Compliant Code** |
| --- |
| In the below example, a try catch block is used and any exceptions thrown by throwing\_func() are handled. Because of this, errors are not rethrown and the thread is able to terminate normally. |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start(void) {  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    void f() {  std::thread t(thread\_start);  t.join();  } |

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

| **Principles(s):** The first principle that applies is “#9 Use Effective Quality Assurance Techniques.” For this standard, using try catch blocks and making sure that all errors are accounted for and handled will assure that the code will behave as intended, and that if there are any errors at runtime, these errors are all safely handled to not introduce any sort of security threat or risk into the system. This also will make sure that the program will not crash or behave in any unexpected manner while it is running. The second principle that applies is “#10 Adopt a Secure Coding Standard.” By using try-catch blocks and making sure that all errors are handled, catalogued, and that when an error is thrown it is caught and the program can recover and continue running despite the error, this will ensure that the program does not crash and is kept secure and that any possible error will not introduce a threat or program crash. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all** **early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++4035, C++4036, C++4037** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **MISRA.CATCH.ALL** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [PRQA QA-C++](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=142409849) | 4.4 | **4035, 4036, 4037** |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Random Numbers** | MSC50-CPP | Do not use std::rand() for generating pseudorandom numbers |
| **Rationale** | | |
| When using std::rand(), this is not random and has both modulo bias and will also generate random numbers in a predictable fashion, so a random number algorithm as well as a distribution function should be used to guard against these issues in the std::rand() method when randomness is desired. | | |

| **Noncompliant Code** |
| --- |
| In the below example, the rand() function is called to generate a random ID. This function will generate IDs in a predictable fashion with limited randomness so it is not random. Also depending on what RAND\_MAX value is, this can cause modulo bias. |
| #include <cstdlib>  #include <string>    void f() {  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed  // by a random integer in the range [0-10000].  id += std::to\_string(std::rand() % 10000);  // ...  } |

| **Compliant Code** |
| --- |
| There are other ways to generate random numbers besides std::rand(). The below example uses an algorithm to provide random values as well as one that runs through a density function to make sure that the distribution is regular and properly distributed so that the bias that was described above is mitigated. |
| #include <random>  #include <string>    void f() {  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed  // by a random integer in the range [0-10000].  std::uniform\_int\_distribution<int> distribution(0, 10000);  std::random\_device rd;  std::mt19937 engine(rd());  id += std::to\_string(distribution(engine));  // ...  } |

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

| **Principles(s):** One principle that applies is “#10 Adopting a Secure Coding Standard.” If randomness is desired in the code, then std::rand() is bad practice to use, as it generates numbers in a predictable pattern, and not randomly. Another principle that applies is “#9 Using Effective Quality Assurance Techniques.” For random numbers where you need them to be random and not in a predictable fashion with bias, it is important to use an algorithm that is able to distribute the numbers evenly and also to make sure they are not generated in a predictable pattern. This increases the quality of the code and is good practice for making truly random numbers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **bad-function (AUTOSAR.26.5.1A)** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MSC50** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 4.0 (prerelease) | cert-msc50-cpp | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **BADFUNC.RANDOM.RAND** | Use of rand |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Rose) |  |  |  |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.MSC30** | Fully implemented |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++5028** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **CERT.MSC.STD\_RAND\_CALL** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **44 S** | Enhanced Enforcement |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-MSC50-a** | Do not use the rand() function for generating pseudorandom numbers |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: MSC50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmsc50cpp.html) | Checks for use of vulnerable pseudo-random number generator (rule partially covered) |
| [PRQA QA-C++](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046345) | 4.4 | **5028** | Fully implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **File In/Out** | FIO51-CPP | Close files when they are no longer needed. |
| **Rationale** | | |
| Closing the files will clear the memory allocated for that file and close the file. Using the close() function to close the file allows the destructors to be called and the file to be properly closed after it is used. | | |

| **Noncompliant Code** |
| --- |
| Below, the file stream object is created, it then opens a file using open(), then the file is not closed, which will lead to std::terminate() being called before the file being closed, meaning destructors are not called due to the file object not being closed properly. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In the below solution, file.close() is called before terminate(), meaning that all of the resources are 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;  }  // ...  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):** The first principle that applies is “#9 Use Effective Quality Assurance Techniques.” When writing code, it is important to write high-quality code that is free from errors and bugs, so it is important to make sure that files are closed after they are open and processing the files is done as this could introduce errors or bugs into the code. The second principle that applies is “#10 Adopt a Secure Coding Standard.” If a file is not closed after being opened, this could lead to errors and bugs, and if an error or bug is introduced into the system, this could become exploitable and cause a security threat or risk. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | **ALLOC.LEAK** | Leak |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **DF4786, DF4787, DF4788** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **RH.LEAK** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) |  |  | Runtime detection |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Arrays** | ARR30-C | Do not form or use out-of-bounds pointers or array subscripts |
| **Rationale** | | |
| In arrays, if a value is tried to be called or inserted at a location larger than the array’s size, this will cause an overflow error or throw an exception if trying to call to the out-of-bounds location for the array. Additionally, an array needs to be properly resized or else this could also cause an overflow error. | | |

| **Noncompliant Code** |
| --- |
| In the below example, in the insert\_in\_table() function, it looks at position and size, and when position equals size, it will try and insert a value in a memory location past the end of the buffer. When index is greater than size, the function also modifies size before growing the size of the buffer. |
| #include <stdlib.h>    static int \*table = NULL;  static size\_t size = 0;    int insert\_in\_table(size\_t pos, int value) {  if (size < pos) {  int \*tmp;  size = pos + 1;  tmp = (int \*)realloc(table, sizeof(\*table) \* size);  if (tmp == NULL) {  return -1; /\* Failure \*/  }  table = tmp;  }    table[pos] = value;  return 0;  } |

| **Compliant Code** |
| --- |
| In the below example, the index position is validated to make sure that there will be no overflow, and it won’t modify size until realloc() call was successful. |
| #include <stdint.h>  #include <stdlib.h>    static int \*table = NULL;  static size\_t size = 0;    int insert\_in\_table(size\_t pos, int value) {  if (size <= pos) {  if ((SIZE\_MAX - 1 < pos) ||  ((pos + 1) > SIZE\_MAX / sizeof(\*table))) {  return -1;  }    int \*tmp = (int \*)realloc(table, sizeof(\*table) \* (pos + 1));  if (tmp == NULL) {  return -1;  }  /\* Modify size only after realloc() succeeds \*/  size = pos + 1;  table = tmp;  }    table[pos] = value;  return 0;  } |

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

| **Principles(s):** The first principle that applies is “# 10 Adopt a Secure Coding Standard.” By referencing an out-of-bounds pointer, this will cause an error to be thrown, which could lead to a security vulnerability or something exploitable, so by making sure that compliant code is written that does not reference an out-of-bounds pointer, this will avoid having an error introduced into the code. The second principle that applies is “#9 Use Effective Quality Assurance Techniques.” By making sure that the code is written well, tested, and that no errors such as this are being thrown, it will help make sure that the code is free from errors such as one that would be introduced by referencing an out-of-bounds pointer location for an array. A third principle that applies is “#2 Heed Compiler Warnings,” as this will throw a warning or error on compile and this should not be ignored and should be addressed and fixed when the warning is thrown. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 22.04 | **array-index-range** **array-index-range-constant** **null-dereferencing** **pointered-deallocation** **return-reference-local** | Partially checked  Can detect all accesses to invalid pointers as well as array index out-of-bounds accesses and prove their absence.  This rule is only partially checked as invalid but unused pointers may not be reported. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-ARR30** | Can detect out-of-bound access to array / buffer |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **LANG.MEM.BO** **LANG.MEM.BU** **LANG.MEM.TBA** **LANG.MEM.TO** **LANG.MEM.TU** **LANG.STRUCT.PARITH** **LANG.STRUCT.PBB** **LANG.STRUCT.PPE** **BADFUNC.BO.\*** | Buffer overrun Buffer underrun Tainted buffer access Type overrun Type underrun Pointer Arithmetic Pointer before beginning of object Pointer past end of object A collection of warning classes that report uses of library functions prone to internal buffer overflows. |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Could be configured to catch violations of this rule. The way to catch the noncompliant code example is to first hunt for example code that follows this pattern:  for (LPWSTR pwszTemp = pwszPath + 2; \*pwszTemp != L'\\';  \*pwszTemp++;)  In particular, the iteration variable is a pointer, it gets incremented, and the loop condition does not set an upper bound on the pointer. Once this case is handled, ROSE can handle cases like the real noncompliant code example, which is effectively the same semantics, just different syntax |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **OVERRUN**  **NEGATIVE\_RETURNS**  **ARRAY\_VS\_SINGLETON**  **BUFFER\_SIZE** | Can detect the access of memory past the end of a memory buffer/array  Can detect when the loop bound may become negative  Can detect the out-of-bound read/write to array allocated statically or dynamically  Can detect buffer overflows |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **arrayIndexOutOfBounds, outOfBounds, negativeIndex, arrayIndexThenCheck, arrayIndexOutOfBoundsCond,  possibleBufferAccessOutOfBounds** | Context sensitive analysis of array index, pointers, etc.  Array index out of bounds  Buffer overflow when calling various functions memset,strcpy,..  Warns about condition (a[i] == 0 && i < unknown\_value) and recommends that (i < unknown\_value && a[i] == 0) is used instead  Detects unsafe code when array is accessed before/after it is tested if the array index is out of bounds |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **C2840**  **DF2820, DF2821, DF2822, DF2823, DF2840, DF2841, DF2842, DF2843, DF2930, DF2931, DF2932, DF2933, DF2935, DF2936, DF2937, DF2938, DF2950, DF2951, DF2952, DF2953** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **ABV.GENERAL** **ABV.GENERAL.MULTIDIMENSION** **NPD.FUNC.CALL.MIGHT** **ABV.ANY\_SIZE\_ARRAY** **ABV.STACK** **ABV.TAINTED** **ABV.UNICODE.BOUND\_MAP** **ABV.UNICODE.FAILED\_MAP** **ABV.UNICODE.NNTS\_MAP** **ABV.UNICODE.SELF\_MAP** **ABV.UNKNOWN\_SIZE** **NNTS.MIGHT** **NNTS.MUST** **NNTS.TAINTED** **SV.TAINTED.INDEX\_ACCESS** **SV.TAINTED.LOOP\_BOUND** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-ARR30-a** | Avoid accessing arrays out of bounds |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) |  |  | Runtime analysis |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **413, 415, 416, 613, 661, 662, 676** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule ARR30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulearr30c.html) | Checks for:   * Array access out of bounds * Pointer access out of bounds * Array access with tainted index * Use of tainted pointer * Pointer dereference with tainted offset   Rule partially covered. |
| [PRQA QA-C](https://wiki.sei.cmu.edu/confluence/display/c/PRQA+QA-C) | 9.7 | **2820, 2821, 2822, 2823, 2840, 2841, 2842, 2843, 2930, 2931, 2932, 2933, 2935, 2936, 2937, 2938, 2950, 2951, 2952, 2953** | Partially implemented |
| [PRQA QA-C++](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046348) | 4.4 | **2820, 2821, 2822, 2823, 2840, 2841, 2842, 2843, 2930, 2931, 2932, 2933, 2935, 2936, 2937, 2938, 2950, 2951, 2952, 2953** | Partially implemented |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.24 | [**V512**](https://pvs-studio.com/en/docs/warnings/v512/), [**V557**](https://pvs-studio.com/en/docs/warnings/v557/), [**V582**](https://pvs-studio.com/en/docs/warnings/v582/), [**V594**](https://pvs-studio.com/en/docs/warnings/v594/), [**V643**](https://pvs-studio.com/en/docs/warnings/v643/), [**V645**](https://pvs-studio.com/en/docs/warnings/v645/), [V694](https://pvs-studio.com/en/docs/warnings/v694/)**,**[**V1086**](https://pvs-studio.com/en/docs/warnings/v1086/) |  |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/c/RuleChecker) | 22.04 | **array-index-range-constant** **return-reference-local** | Partially checked |

### 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.

One way to better automate enforcement of security standards would be to add different automation tools that can be used to detect new bugs or security vulnerabilities in the code. Just because no bugs or vulnerabilities are found at the time the code is built, doesn’t mean that one won’t be found later, and it is important to know if any of the code, or any of the packages or dependencies from the code has any vulnerabilities. So, this is why it is important to scan the code periodically with an automated tool, and not just once during the verifying testing phase listed above in pre-production.

Also, another important automation tool to add in the pre-production process would be one where the code is scanned and compiled, and unit tests are written and run periodically to see if there are any areas of the code that are vulnerable to attacks or exploitation, or if there are any insecure coding practices being used. It’s important to not just build and test the code once, but to incrementally build sections of the code, write strong unit tests for these sections of code, and then compile the code and test to see if any compile-time warnings or any bugs/errors will come to the surface so they can be fixed, and the code reworked before anything ever goes to production. Building and running unit tests for each section of the code to ensure that there is good code coverage, and each error case can be tested to make sure that the correct code path is being taken in the case of good code as well as errored code.

For the above policy, it appears that there are two “pathways” that are taken for security. One is the pre-production pathway, and one is the production pathway. To change this diagram, I would make sure that areas like monitor and detect are not just done in a production environment, but that these steps are also done in the pre-production pathway. It is much easier to fix a bug or coding error if it is detected early in the coding process, so it’s important that there is also monitoring and detection of any bugs happening while a developer is writing the code or adding in different packages and dependencies to write the code. Each package and dependency should be monitored, and the correct version selected so that any security vulnerabilities or bugs have been remediated, and these packages should continue to be monitored to make sure that there aren’t any new security vulnerabilities that are introduced during the development process.

I would also add in a section in the pre-production process to make sure that while designing, and while coding, that security is at the forefront of the developers’ process, so that they aren’t writing insecure code. For instance, secure coding patterns should be designed and coded and not thought of as a production after-thought. The developer should be thinking about things like avoiding SQL Parameterization, making sure that the Principle of Least Privilege is followed, adding in security and roles to the code during the code process, and making sure they are validating any user input because all of this will cut down on potential re-work down the road that would happen if security was an afterthought.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL52-CPP | Low | Unlikely | Low | P3 | L3 |
| EXP50-CPP | Medium | Probable | Medium | P8 | L2 |
| STR50-CPP | High | Likely | Medium | P18 | L1 |
| IDS00-J | High | Probable | Medium | P12 | L1 |
| MEM51-CPP | High | Likely | Medium | P18 | L1 |
| DCL03-C | Low | Unlikely | High | P1 | L3 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| MSC50-CPP | Medium | Unlikely | Low | P6 | L2 |
| FIO51-CPP | Medium | Unlikely | Medium | P4 | L3 |
| ARR30-C | High | Likely | High | P9 | L2 |

### 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 | Encryption in rest is data that is being stored, normally in a database. This data is not actively being used nor is it actively moving through the system or network. When stored in the database, this data will be stored behind a firewall, behind anti-virus and anti-malware software and any sensitive data will be encrypted using a key so that it must be decrypted before it is able to be used. The data will also only be able to be accessed by certain roles and be protected by requiring users to login with a username and password, and by using multi-factor authentication for instance with a duo-push to verify their identity before gaining access to this data, applying the principle of least privilege. In addition, this follows the principles of Defense in Depth and Default Deny as this is one layer of security that is added to the system to keep unauthorized users out of the system, and only allow those who can have access to the data access only to what they are least privileged to have access to. |
| Encryption at flight | Encryption in flight is data that is currently moving through the system or network. It is currently in transit and in the process of being transferred. To encrypt data that is currently moving through the network, TLS (transport layer security) will be used. This is a security protocol that is used to encrypt this data as well as keeps this data from being transferred to a third party, makes sure to authenticate who is sending and/or receiving the data, as well as by making sure that the data has not been tampered with during the transfer. This follows the principles of Default Deny, and Architect and Design for Security Policies, as when encryption in flight is enabled, it will keep those who do not have a valid security certificate from accessing the data, since they will not be able to decrypt the data without a valid certificate and key. This needs to be designed into the system so that anywhere that data is being transferred through or into the system, or even between systems, the data is encrypted before transport, and kept encrypted while it is being transported through the system so that the data is not vulnerable to attacks while it is at flight. |
| Encryption in use | Data in us is data that is currently being read, accessed, updated, modified, or deleted. It may be data that is regularly used or updated and does not just sit in the database, and it is data that is being used in the system rather than being transported between systems. Data that is currently being used must be kept secure by making sure that the system is secure as a whole. Also, data in use should still be encrypted, and the system itself will have the ability to decrypt when needed so that it can be actively used so that even if a breach occurs, any data that is currently in use will not be stolen in its unencrypted state. The principles that apply to this are the Principle of Least Privilege, Architect and Design for Security Policies, Default Deny, and Practice Defense in Depth. The principle of least privilege and default deny apply as users should only be given access to areas of the system that they need access to, and no additional access. This way, they are only able to access the areas of the system dictated by their role and have less access to data that is in use in other areas of the system. Any other areas of the system that they do not have access to should be set up to deny by default unless the correct role has access, to limit the amount of data in use that has the possibility to be vulnerable. When designing the system, the principle of architect and design for security policies is important for encryption in use, as the system should be designed to keep all in-use data encrypted as long as possible and provide a mechanism for the system to decrypt and use the data, and then re-encrypt the data. This way, the data has less time spent unencrypted and less of a chance of being vulnerable to a security breach. Additionally, the principle of practice defense in depth applies here, as there should be multiple layers set up, with secure coding patterns to protect data in use, using the principle of least privilege to set up access levels and roles and using default deny to deny anyone without the correct role, but also things like firewall protection, malware and anti-virus software to protect the system. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the way that a user is identified. For instance, this may be through having the user providing a username and password to login to a system, or may be through Auth0 authentication where the user has to provide a valid token, or it could be providing a valid client-id and client-secret in order to authenticate that they are who they say they are. If the user’s credentials match, then the user is authenticated and gains access to the system, otherwise they are denied access to the system. The policy of authentication follows the principles of security because this is one layer in practicing defense in depth, where users must have valid credentials to access the system. If they do not, they are denied access, which also follows the policy of default deny. By setting up a system to have users present credentials and login to the system, this is also good architecture design for security policies. |
| Authorization | The principle of authorization is that a user must have the correct access level and must have the authority to perform certain tasks or access certain areas of the system. If the user does not have the correct authority or access level, they will be denied from performing certain tasks. This can be achieved by following the principle of least privilege, where a user is given a role, and if their role does not have certain access, they are denied from performing certain actions. This also follows the principle of default deny, as a user will be denied by default from certain areas of the system or from performing certain actions if they do not have the correct role. A user first must be authenticated to gain entry to the system, and then must also be authorized to make sure that they have the correct access level to perform the task they are trying to perform. This is an example of using the principle of architecting and designing for security policies as well as practicing defense in depth, as this is good coding practice and design for security and is adding multiple layers for a user to gain access to areas of the system. |
| Accounting | The principle of accounting is making sure to log and keep track of the resources that a user is consuming during their access of the system. Not only which resources, but how many resources. This means keeping detailed logging of data that was used, the amount of data that was accessed, and keep track of the user’s footprint and statistics while accessing the system. This is important, because an owner of the system needs to know how many resources a user is consuming, when, and how the data is being used by each user. This can protect against a user who is trying to maliciously use the data and access a large amount of data and can help the owner of the system justify revoking access to certain areas of the system or revoking access altogether if a user is showing suspicious activity in the system. This follows the principles of defense in depth, as well as default deny and the principle of least privilege, since adding logging to keep track of users’ activity in the system to make sure they aren’t accessing too many resources or acting suspiciously while in the system is an important extra level of defense, and if a user is acting suspiciously, they can be revoked their role and will be denied by default from accessing the system. This will also fall under the principle of architecting and designing for security policies, as logging a user’s footprint on a site cannot be done as an afterthought and must be designed and implemented throughout the code, so that a user’s action’s will be logged and can be tracked. |

**\***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 |  |
| 2.0 | 03/16/2023 | Added Principle definitions and standards | Erin Walter |  |
| 3.0 | 04/08/2023 | Added standards, risks, automation tools, AAA, and encryption standards | Erin Walter |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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