**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** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All data that the user inputs should be validated. Validating input data prevents invalid data from being entered. Invalid input data can be a source of attacks. One such type of attack is SQL injection. Because of this, validating user data is a critical security principle to protect against hackers. When invalid data is handled, that eliminates a potential vector of attack for a hacker. |
| 1. Heed Compiler Warnings | Compilers will often show errors and warnings. A program will not compile with errors, but it can with warnings. This is bad practice however because lines of code that can cause these warnings can also cause exploits in the code. For keeping code secure, best practices dictate that programmers should attempt to address all warnings regardless of what the causes of the warnings are. |
| 1. Architect and Design for Security Policies | System architectures should be designed to be as secure as possible. Security needs to be considered at all stages of software design. It is easier to build a system from the ground up to be secure than it is to tighten up security later. This saves time, and it also helps make any security policies that are in place more effective because the software is designed with those policies in mind. |
| 1. Keep It Simple | Systems should be designed in as simple a manner as possible. A simpler design is a more efficient design in almost every area of software development. In addition to that, a more complex system can have more potential areas of attack than a system with a simpler design. It is easier to protect against security vulnerabilities when a system has a simpler design. |
| 1. Default Deny | By default, a user should be denied access to sensitive data. The only time a user should be allowed access is if they provide proper credentials. This is important because allowing access by default means that it is too easy for a hacker to access sensitive information. The default behavior of a program should always be to deny users access to data that they do not have credentials to access. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege dictates that a user’s privilege should be limited only to data that they need to access to accomplish required tasks. Most users do not need access to the entire program or all the data within the program’s databases. Allowing unnecessary access poses a massive security risk should that user’s credentials be compromised. It is important to prevent that from happening by adhering to the principle of least privilege. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization is the practice of permanently erasing data from a system. This is important because it ensures that sensitive data will not be left behind on a system when that system is taken out of service. If sensitive data is left on a system when it is no longer needed there, then it is possible for that data to be recovered by a bad actor. It is important to make sure that data is sanitized when necessary to prevent this. |
| 1. Practice Defense in Depth | Defense in depth is the practice of implementing many layers of defense. This is important because it means that an attacker would have to make it through multiple layers of defense to access important systems. While there is eventually a point of diminishing returns, more layers of defense means that a hacker would require more time and more resources to be able to make it through all those layers. This means that it is important to have many layers of defense rather than one or even just a few. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance testing is an important part of software development. It is just as important for security as it is for functionality of a program. Quality assurance testing can help with finding vulnerabilities in code that could otherwise go unnoticed by the developers. Quality assurance can be done either manually or through automation, but the most important thing is that quality assurance happens in some form. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard means that security is being taken seriously at all levels. Doing this ensures that hackers are less likely to gain access to any data. The reason that adopting a secure coding standard helps achieve this goal is that it gives a target to shoot for. Secure coding standards became standards because the practices used in them have traditionally worked well. Adopting a secure coding standard means that good practices related to security are being followed at all times. |

### 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** | **DCL52-CPP. Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Data types can be const or volatile qualified. However, this should not be done with reference types. This is because attempting to const or volatile qualify a reference type will result in undefined behavior, which poses a security risk. The data types being referenced on the other hand can safely be const or volatile qualified. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it is a const-qualification of a reference type. |
| char &const p; |

| **Compliant Code** |
| --- |
| This line of code is compliant because it is const-qualifying a non-reference type. |
| char const &p; |

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

| **Principles(s):** The principle “Validate Input Data” applies here because it has to do with validating data types. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL52 |  |
| Helix QAC | 2024.2 | C++0014 |  |
| Klocwork | 2024.2 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE | Never qualify a reference type with ‘const’ or ‘volatile’ |
| Clang | 3.9 |  | Clang checks for rule violations and produces an error. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **DCL60-CPP. Obey the one-definition rule** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Variables should only have one definition. Giving a variable more than one definition causes undefined behavior. Any undefined behavior poses a security risk, and it should be addressed. Avoiding giving variables more than one definition helps avoid undefined behavior. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it defines a variable multiple times. |
| // a.cpp  struct S {  int a;  };    // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| This block of code is compliant because it uses the same definition for int a rather than defining it multiple times. |
| // S.h  struct S {  int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** The principle “Keep it Simple” applies here because this standard is easier to follow when things are kept simple. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL60 |  |
| CodeSonar | 8.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Helix QAC | 2024.2 | C++1067, C++1509, C++1510 |  |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **STR51-CPP. Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Creating a string using a null pointer requires dereferencing a null pointer. Dereferencing a null pointer can lead to undefined behavior. Undefined behavior can pose a security risk. Because of this, it is important to not create a string from a null pointer. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it is attempting to create a string using std::getenv(). In this instance, std::getenv() returns a null pointer when in fails, which violates this rule. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it checks whether std::getenv() returned a null pointer before it attempts to create a string using that value. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** The principle “Validate Input Data” applies here because strings are often filled with data from user input. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2024.2 | DF4770, DF4771, DF4772, DF4773, DF4774 |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **IDS00-J. Prevent SQL injection** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-JAV] | SQL injection vulnerabilities should always be avoided. SQL injection vulnerabilities allow attackers to bypass authentication measures. A common example of this is the “or 1=1” vulnerability, which can allow an attacker to easily bypass authentication measures if it is not addressed. It is important to prevent such attacks from happening by preventing SQL injection vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because it does not sanitize user input for the username argument, which means that it vulnerable to a SQL injection attack. |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;    class Login {  public Connection getConnection() throws SQLException {  DriverManager.registerDriver(new  com.microsoft.sqlserver.jdbc.SQLServerDriver());  String dbConnection =  PropertyManager.getProperty("db.connection");  // Can hold some value like  // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  return DriverManager.getConnection(dbConnection);  }    String hashPassword(char[] password) {  // Create hash of password  }    public void doPrivilegedAction(String username, char[] password)  throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  try {  String pwd = hashPassword(password);    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** |
| --- |
| This code is compliant because it uses prepared statements to allow prevent a SQL injection attack. |
| public void doPrivilegedAction(  String username, char[] password  ) throws SQLException {  Connection connection = getConnection();  if (connection == null) {  // Handle error  }  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 principles “Default Deny” and “Adhere to the Principle of Least Privilege” both apply here because they both relate to preventing a user from bypassing authentication and accessing data that they do not need to access. “Architect and Design for Security Policies,” “Practice Defense in Depth,” and “Adopt a Secure Coding Standard” also apply here because following them also helps prevent SQL injections. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 8.1p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Parasoft Jtest | 2024.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE  SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **MEM50-CPP. Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Memory should not be accessed after it is freed. Trying to access a pointer to a place in memory that has been freed can result in undefined behavior. Any undefined behavior should be treated as a potential security risk. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because an s is being dereferenced after that part of memory has been deallocated. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because s is not deleted until it is no longer needed. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** “Sanitize Data Sent to Other Systems” applies here because it relates to permanently deleting data. Once that data is deleted, no further attempts should be made to access it. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |
| Helix QAC | 2024.2 | C++4303, C++4304 |  |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |

#### Coding Standard 6

| **Coding Standard** | **Label** | **MET01-J. Never use assertions to validate method arguments** |
| --- | --- | --- |
| **Assertions** | [STD-006-JAV] | Assertions should not be used to check a method’s arguments. One reason for this is that it can prevent arguments from throwing the correct exceptions. When this happens, it is harder to troubleshoot what is causing the exceptions. This can result in undefined behavior going unaddressed, which can create security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it is using assertions to validate method arguments. |
| public static int getAbsAdd(int x, int y) {  assert x != Integer.MIN\_VALUE;  assert y != Integer.MIN\_VALUE;  int absX = Math.abs(x);  int absY = Math.abs(y);  assert (absX <= Integer.MAX\_VALUE - absY);  return absX + absY;  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it does not use assertions to validate method arguments. |
| public static int getAbsAdd(int x, int y) {  if (x == Integer.MIN\_VALUE || y == Integer.MIN\_VALUE) {  throw new IllegalArgumentException();  }  int absX = Math.abs(x);  int absY = Math.abs(y);  if (absX > Integer.MAX\_VALUE - absY) {  throw new IllegalArgumentException();  }  return absX + absY;  } |

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

| **Principles(s):** The principle “Use Effective Quality Assurance Techniques” applies here because it relates to using quality assurance techniques. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 |  |  |
| SpotBugs | 4.6.0 |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **ERR51-CPP. Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | This standard is important because all unexpected behaviors in code need to be handled. Unexpected behaviors can lead to many issues in code. They can also cause security problems in code. Because of this, it is important to handle all exceptions that can occur when a program is running. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it does not handle the exception that is thrown. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it does handle the exception that is thrown. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** “Heed Compiler Warnings” applies here because compiler warnings help identify the issues addressed here. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 8.1p0 | LANG.STRUT.UCTCH | Unreachable Catch |
| Klocwork | 2024.2 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **ERR50-CPP. Do not abruptly terminate the program** |
| --- | --- | --- |
| **Exceptions** | [STD-008-CPP] | While it is important to handle all exceptions, it is also important to make sure that they are handled safely. Abruptly terminating the program can leave behind data that has not been sanitized. This means that there are still files and memory that are vulnerable to attacks. Because of this, it is important to take care of data that has been initialized before exiting the program. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it exits the program without handling the exception. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it handles all exceptions without exiting the program. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):** The principle “Sanitize Data Sent to Other Systems” applies here because data should be sanitized in the event of an exception before a program is exited. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use | Partially checked |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| Helix QAC | 2024.2 | C++5014 |  |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced enforcement |

#### Coding Standard 9

| **Coding Standard** | **Label** | **STR53-CPP. Range check element access** |
| --- | --- | --- |
| **String Correctness** | [STD-009-CPP] | It is important to check the range of a string before attempting to modify a value from it. Attempting to access an out-of-range value will result in undefined behavior. Because of this, the range of the string needs to be checked. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it fails to check the range of the string before performing an operation on it. |
| #include <string>    extern std::size\_t get\_index();    void f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it checks the range of the string before performing an operation on it. |
| #include <string>    extern std::size\_t get\_index();    void f() {  std::string s("01234567");  std::size\_t i = get\_index();  if (i < s.length()) {  s[i] = '1';  } else {  // Handle error  }  } |

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

| **Principles(s):** “Heed Compiler Warnings” applies here because the compiler will explain the error if you try to access a string that is out of range. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Helix QAC | 2024.2 | C++3162, C++3163, C++3164, C++3165 |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |

#### Coding Standard 10

| **Coding Standard** | **Label** | **MEM52-CPP. Detect and handle memory allocation errors** |
| --- | --- | --- |
| **Memory Protection** | [STD-010-CPP] | Failure to handle memory allocation errors can result in a null pointer being returned. A null pointer being returned leads to undefined behavior, which poses a security risk. If memory cannot be allocated, then it needs to be handled in some form. |

| **Noncompliant Code** |
| --- |
| This block of code is noncompliant because it does not check the results of the memory allocation, which means that it is possible for unhandled exceptions to occur. |
| #include <cstring>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy = new int[size];  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

| **Compliant Code** |
| --- |
| This block of code is compliant because it uses std::bad\_alloc to check the results of the memory allocation. This means that any potential errors in memory allocation that can occur are handled appropriately. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {  int \*copy;  try {  copy = new int[size];  } catch(std::bad\_alloc) {  // Handle error  return;  }  // At this point, copy has been initialized to allocated memory  std::memcpy(copy, array, size \* sizeof(\*copy));  // ...  delete [] copy;  } |

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

| **Principles(s):** The principle “Use Effective Quality Assurance Techniques” applies here because effecting quality assurance techniques will help prevent a situation like this from occurring. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Helix QAC | 2024.2 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| PVS-Studio | 7.32 | V522, V668 |  |

### 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.



One area where the automation can help is in the testing stage during pre-production. In this stage of development, automation can be used in the form of unit tests. Unit tests can help make sure that the program is functioning properly. They can be implemented without changing the existing code. Static code analysis can also be implemented here.

Automation is also useful in the monitoring and detecting stage during production. In this stage of development, automation can help handle the tasks listed in this stage. Some responses could potentially be automated as well, but there are some parts where a developer might have to respond. The best way to implement automation in both areas would be to have it work alongside the developers.

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

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the process of encrypting data in storage. This helps protect data that is saved in a file system or a database. This is important because data in a database can be vulnerable even when it is not in use. For example, personal information for most services is stored in a database. With no safeguards in place, this data would be vulnerable if a hacker was to gain access to that database. With a policy of encryption at rest in place, that information will be protected. |
| Encryption in flight | Encryption in flight is the process of encrypting data while it is in transit from one point to another. This helps protect data that is being transmitted from one source to another. An example of how this can be used is using secure protocols such as HTTPS when transmitting data. Sending data over a network can create potential opportunities for hackers to compromise that data, so that data needs to be protected. Having a policy of encryption in flight helps achieve this goal because it helps prevent attacks in which an attacker gains access to the network. |
| Encryption in use | Encryption in use is the process of encrypting data that is actively being used. For example, data being entered by a user is encrypted in this process. This is important because while encryption at rest helps secure data that is in storage, a policy of encryption in use helps protect data that is still in memory. Having a policy of encryption in use along with following the coding standards related to memory protection can help prevent attacks in which a hacker is targeting data still in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves getting information from the user, such as a user login. This helps make sure that only authorized users are gaining access to information. The process of authentication by user login verifies that a user is who they say they are. |
| Authorization | Authorization is the stage in which an administrator gives a user privileges. Authorization controls users’ level of access. This is important because not every user needs the same level of access in most cases. This would also be the stage in which new users are added. |
| Accounting | Accounting is the process of keeping track of user activity. This includes changes to the database and files accessed by users. This is important because keeping track of this information helps keep track of how all data is used. In the event of a data breach, this could help with understanding how and where the data breach occurred. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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