**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 | Input Validation is used to exclusively allow properly formed data into the workflow and to prevent unauthorized/incorrect input from triggering errors, bugs, and other malfunctions within the system.  Input Validation is applied syntactically and semantically. Examples of this include, but are not limited to data type validators, type conversion with strict exception handling, range checks on numerical characters, length and format checks on Strings, and that characters meet programmer-defined criteria (e.g. no use of special characters). |
| 1. Heed Compiler Warnings | Compiler Warnings are messages from the compiler regarding code that should be looked over by the developer as it could contain software bugs. These warnings are not an error according to the syntax of the used programming language but can cause problems if left unchecked.  Acting on compiler warnings can fix defects earlier in the Software Development Life Cycle and save time and money in the process. Code should be compiled using the highest warning level available and static analysis tools can be used in conjunction to further fix warnings. |
| 1. Architect and Design for Security Policies | The architecture and overall design of your program should enforce security policies. This can include authentication (distinctions between different users) and authorization (what each type of user is privileged to access) decisions. It assists in protecting the integrity of data and the confidentiality of the user within the system.  Enforcing the security principles within your design can save your organization reputation and time lost trying to recover from security breaches. |
| 1. Keep It Simple | Keep the design and development of your system as simple and small as can be afforded. With increasing complexity comes a higher chance of mistakes to be caused in your system as well as time involved in maintenance of said system.  In terms of security, it is often best to not use too many security mechanisms as this adds more layers of complexity and instead base your choices of security off analysis such as risk assessments.  In terms of development, proper coding guidelines and patterns should be followed and implemented to keep code as maintainable as possible. Using concise inline comments is also a best practice in keeping things simple yet effective. |
| 1. Default Deny | Contrary to Default Allow, Default Deny means making decisions on permissions instead of exclusions. By default, unless permission is granted or permitted through predefined rules and policies, access will be denied. This is particularly used within the realms of network traffic, system access, and application requests. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege involves giving each role/category of users within a system the least amount of privilege that they need to perform their assigned duties. An example of this is a viewer of a document who is given access to read the file, but not write to it. The Principle of Least Privilege is a tradeoff between usability and security which is designed to minimize the impact of human error and security breaches.  Practicing the Principle of Least Privilege can save companies from the impact a security breach could possess such as loss of cost, time, and resources. |
| 1. Sanitize Data Sent to Other Systems | Data Sanitization involves securely and permanently deleting or destroying data from a storage device so it cannot be recovered. This should be done when data is passed to complex systems such as databases, disk drives, and command shells. Hackers may be able to use techniques such as SQL Injection Attacks if not sanitized and this can lead to consequences within your system such as invasions of data privacy. |
| 1. Practice Defense in Depth | Defense in Depth involves adding multiple security layers to enhance the general security of your system. The logic is that if one layer of defense is insufficient for the threat that another layer can prevent an inadequacy in the security design from becoming an exploit. An example of this is using Object Oriented Programming/secure coding with a protected development environment so that there is more than 1 layer of security against code vulnerabilities producing an area for attackers to take advantage of. |
| 1. Use Effective Quality Assurance Techniques | A variety of strategic Quality Assurance Techniques should be used to increase the likelihood of a secure system. Some examples of this are bringing in a QA team that assists developers in providing assurance as well as regression testing when there are big changes within the code of the system.  Many popular and emerging methodologies such as Agile have begun practicing Test Driven Development in which code is tested in segments as they are completed. This can include Unit Tests, Integration, and Acceptance Testing. This is contrarily done to other models such as the Waterfall Method where testing is done after development. |
| 1. Adopt a Secure Coding Standard | A coding standard is a collection of rules, guidelines, and best practices to be taken by Software Developers when developing a program or application. Some examples of this include limiting the amount of each statement to 80-100 characters, naming conventions such as test classes starting with the name of the class they are testing and providing multi-line block comments for describing each function/class. Use of a Static Code Analyzer can ensure that your Coding Standards are being followed within the application. |

### 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** | INT-031-C | Ensure that integer conversions do not result in lost or misinterpreted data. |

| **Noncompliant Code** |
| --- |
| 1.Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  /\* ... \*/  } |
| 2. Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of a signed type to a value of an unsigned type. This noncompliant code example results in a negative number being misinterpreted as a large positive number. |
| #include <limits.h>    void func(signed int si) {  /\* Cast eliminates warning \*/  unsigned int ui = (unsigned int)si;    /\* ... \*/  }    /\* ... \*/    func(INT\_MIN); |

| **Compliant Code** |
| --- |
| 1.Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type: |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  if (u\_a <= SCHAR\_MAX) {  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  } else {  /\* Handle error \*/  }  } |
| 2. Validate ranges when converting from a signed type to an unsigned type. This compliant solution converts a value of a signed int type to a value of an unsigned int type: |
| #include <limits.h>    void func(signed int si) {  unsigned int ui;  if (si < 0) {  /\* Handle error \*/  } else {  ui = (unsigned int)si; /\* Cast eliminates warning \*/  }  /\* ... \*/  }  /\* ... \*/    func(INT\_MIN + 1); |

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

| **Principles(s):**  Principle #10 (Adopt a Secure Coding Standard): Using a Static Analyzer tool such as Cppcheck when dealing with integers can increase the likelihood of detecting truncation if it occurs. Also providing comments may help a programmer catch the mistake of an unsafe type conversion or at the very least help narrow down the mistake down the road. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity\* | 2017.07 | NEGATIVE\_RETURNS  REVERSE\_NEGATIVE  MISRA\_CAST | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instances where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of a complex expression is implicitly converted |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| Polyspace Bug Finder | R2023a | CERT C: Rule INT31-C | Checks for:  Integer conversion overflow  Call to memset with unintended value  Sign change integer conversion overflow  Tainted sign change conversion  Unsigned integer conversion overflow  Rule partially covered. |
| TrustInSoft Analyzer | 1.38 | signed\_downcast | Exhaustively verified. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT-030-C | Ensure that unsigned integer operations do not wrap |

| **Noncompliant Code** |
| --- |
| 1.This noncompliant code example can result in an unsigned integer wrap during the subtraction of the unsigned operands ui\_a and ui\_b. If this behavior is unanticipated, it may lead to an exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int udiff = ui\_a - ui\_b;  /\* ... \*/  } |
| 2. This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| 1.This compliant solution performs a precondition test of the unsigned operands of the subtraction operation to guarantee there is no possibility of unsigned wrap: |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int udiff;  if (ui\_a < ui\_b){  /\* Handle error \*/  } else {  udiff = ui\_a - ui\_b;  }  /\* ... \*/  } |
| 2. This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap: |
| #include <limits.h>    void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum;  if (UINT\_MAX - ui\_a < ui\_b) {  /\* Handle error \*/  } else {  usum = ui\_a + ui\_b;  }  /\* ... \*/  } }  /\* ... \*/  } |

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

| **Principles(s):**  Principle #9 (Use Effective Quality Assurance Techniques): Programmers should constantly be checking their work for correctness and using a variety of tests to include unit testing to ensure the program is performing as intended and is within boundary. Using a variety of pre and post condition tests can increase the confidence of wraparound not occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.04 | integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| Polyspace Bug Finder | R2023a | CERT C: Rule INT30-C | Checks for:  Unsigned integer overflow  Unsigned integer constant overflow  Rule partially covered. |
| TrustInSoft Analyzer | 1.38 | unsigned overflow | Exhaustively verified. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR-031-C | Characters and Strings- Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| 1.The following code shows an off-by-one error. Because the loop within the code copying characters from src to dest does not include the null-termination character, it will write 1 byte past the end of dest, which can cause Buffer Overflow. |
| #include <stddef.h>    void copy(size\_t n, char src[n], char dest[n]) {  size\_t i;    for (i = 0; src[i] && (i < n); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |
| 2.The gets() function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from stdin. This noncompliant code example assumes that gets() will not read more than BUFFER\_SIZE - 1 characters from stdin. This is an invalid assumption, and the resulting operation can result in a buffer overflow.  The gets() function reads characters from the stdin into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. |
| #include <stdio.h>    #define BUFFER\_SIZE 1024    void func(void) {  char buf[BUFFER\_SIZE];  if (gets(buf) == NULL) {  /\* Handle error \*/  }  } |

| **Compliant Code** |
| --- |
| 1.In this block of code, the termination for the loop is changed for consideration of the null-termination character that is added to the end of dest: |
| #include <stddef.h>    void copy(size\_t n, char src[n], char dest[n]) {  size\_t i;    for (i = 0; src[i] && (i < n - 1); ++i) {  dest[i] = src[i];  }  dest[i] = '\0';  } |
| 2. The fgets() function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from stdin to buf cannot exceed the allocated memory: |
| #include <stdio.h>  #include <string.h>    enum { BUFFERSIZE = 32 };    void func(void) {  char buf[BUFFERSIZE];  int ch;    if (fgets(buf, sizeof(buf), stdin)) {  /\* fgets() succeeded; scan for newline character \*/  char \*p = strchr(buf, '\n');  if (p) {  \*p = '\0';  } else {  /\* Newline not found; flush stdin to end of line \*/  while ((ch = getchar()) != '\n' && ch != EOF)  ;  if (ch == EOF && !feof(stdin) && !ferror(stdin)) {  /\* Character resembles EOF; handle error \*/  }  }  } else {  /\* fgets() failed; handle error \*/  }  } |

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

| **Principles(s):**  Principle #9 (Use Effective Quality Assurance Techniques): Programmers should constantly be checking their work for correctness and using a variety of tests to include unit testing to ensure the program is performing as intended and is within boundary. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| CodeSonar | 7.4p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Astrée | 23.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS-000-J | Prevent SQL injection |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a char array, the database connection is created, and then the passwords are hashed.  Unfortunately, this code example permits a SQL injection attack by incorporating the unsanitized input argument username into the SQL command, allowing an attacker to inject validuser' OR '1'='1. The password argument cannot be used to attack this program because it is passed to the hashPassword() function, which also sanitizes the input. |
| 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 compliant solution uses a parametric query with a ? character as a placeholder for the argument. This code also validates the length of the username argument, preventing an attacker from submitting an arbitrarily long user name. |
| 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):**  Principle #1 (Validate User Input): Using the correct validations for a proper situation, such as length and format checks on Strings input from the user, can be used in conjunction with other safety practices to prevent SQL injection attack from an unauthorized user. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Parasoft Jtest | 2023.1 | 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 |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-050-CPP | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| 1.In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |
| 2. In the following noncompliant code example, the dynamically allocated memory managed by the buff object is accessed after it has been implicitly deallocated by the object's destructor. |
| #include <iostream>  #include <memory>  #include <cstring>    int main(int argc, const char \*argv[]) {  const char \*s = "";  if (argc > 1) {  enum { BufferSize = 32 };  try {  std::unique\_ptr<char[]> buff(new char[BufferSize]);  std::memset(buff.get(), 0, BufferSize);  // ...  s = std::strncpy(buff.get(), argv[1], BufferSize - 1);  } catch (std::bad\_alloc &) {  // Handle error  }  }    std::cout << s << std::endl;  } |

| **Compliant Code** |
| --- |
| 1.In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |
| 2. In this compliant solution, the lifetime of the buff object extends past the point at which the memory managed by the object is accessed. |
| #include <iostream>  #include <memory>  #include <cstring>    int main(int argc, const char \*argv[]) {  std::unique\_ptr<char[]> buff;  const char \*s = "";    if (argc > 1) {  enum { BufferSize = 32 };  try {  buff.reset(new char[BufferSize]);  std::memset(buff.get(), 0, BufferSize);  // ...  s = std::strncpy(buff.get(), argv[1], BufferSize - 1);  } catch (std::bad\_alloc &) {  // Handle error  }  }    std::cout << s << std::endl;  } |

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

| **Principles(s):**  Principle #4 (Keep it Simple): Keeping your program as simple and small as possible, such as constructing memory objects at the points they are to be used and deconstructing said objects when it is ensured that the object will no longer need use, helps your program with memory management as well as vulnerability prevention. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| CodeSonar | 7.4p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Polyspace Bug Finder | R2023a | CERT C++: MEM50-CPP | Checks for:  Pointer access out of bounds  Deallocation of previously deallocated pointer  Use of previously freed pointer  Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MSC-011-C | Incorporate diagnostic tests using assertions |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors. |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char \*)malloc(len + 1);  assert(NULL != dup);    memcpy(dup, c\_str, len + 1);  return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char\*)malloc(len + 1);  /\* Detect and handle memory allocation error \*/  if (NULL == dup) {  return NULL;  }    memcpy(dup, c\_str, len + 1);  return dup;  } |

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

| **Principles(s):**  Principle #6 (Practice Defense in Depth): Using a security layer to help in detecting bugs, such as assertions, will assist in preventing denial-of-service attacks and other vulnerabilities that could have been avoided within an application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2023.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |
| PVS-Studio | 7.26 | V551, V606, V649, V779 |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-051-CPP | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| 1.In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |
| 2. In this noncompliant code example, the thread entry point function thread\_start() does not catch exceptions thrown by throwing\_func(). If the initial thread function exits because an exception is thrown, std::terminate() is called. |
| #include <thread>    void throwing\_func() noexcept(false);    void thread\_start() {  throwing\_func();  }    void f() {  std::thread t(thread\_start);  t.join();  } |

| **Compliant Code** |
| --- |
| 1.In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |
| 2. In this compliant solution, the thread\_start() handles all exceptions and does not rethrow, allowing the thread 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):**  Principle #2 (Heed Compiler Warnings): Compiler warnings are often best set at the highest level and should not be ignored when come across in code. In this example, a warning could very well alert the programmer of a potential and error and the need to use exceptions to avoid an unexpected termination of the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| CodeSonar | 7.4p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++test | 2023.1 | 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 |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Characters and Strings** | STR-002-C | Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| 1.Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() [Viega 2003]: |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |
| 2. This noncompliant code example is taken from [VU#881872], a vulnerability in the Sun Solaris TELNET daemon (in.telnetd) that allows a remote attacker to log on to the system with elevated privileges.  The vulnerability in in.telnetd invokes the login program by calling execl(). This call passes unsanitized data from an untrusted source (the USER environment variable) as an argument to the login program: |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")),  0); |

| **Compliant Code** |
| --- |
| 1.The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |
| 2. This compliant solution inserts the "--" (double dash) argument before the call to getenv("USER") in the call to execl(): |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  "--",  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")), 0); |

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

| **Principles(s):**  Principle #8 (Sanitize Data sent to other Systems): Data Sanitization involves securely and permanently deleting or destroying data from a storage device so it cannot be recovered. This should be done when data is passed to complex systems such as databases, disk drives, and command shells. Hackers may be able to use techniques such as SQL Injection Attacks if not sanitized and this can lead to consequences within your system such as invasions of data privacy. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |
| Polyspace Bug Finder | R2023a | CERT C: Rec. STR02-C | Checks for:  Execution of externally controlled command  Command executed from externally controlled path  Library loaded from externally controlled path  Rec. partially covered. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | EXP-000-J | Do not ignore values returned by methods |

| **Noncompliant Code** |
| --- |
| 1.This noncompliant code example attempts to delete a file but fails to check whether the operation has succeeded: |
| public void deleteFile(){    File someFile = new File("someFileName.txt");  // Do something with someFile  someFile.delete();    } |
| 2. This noncompliant code example ignores the return value of the String.replace() method, failing to update the original string. The String.replace() method cannot modify the state of the String (because String objects are immutable); rather, it returns a reference to a new String object containing the modified string. |
| public class Replace {  public static void main(String[] args) {  String original = "insecure";  original.replace('i', '9');  System.out.println(original);  }  } |

| **Compliant Code** |
| --- |
| 1.This compliant solution checks the Boolean value returned by the delete() method and handles any resulting errors: |
| public void deleteFile(){    File someFile = new File("someFileName.txt");  // Do something with someFile  if (!someFile.delete()) {  // Handle failure to delete the file  }    } |
| 2. This compliant solution correctly updates the String reference original with the return value from the String.replace() method: |
| public class Replace {  public static void main(String[] args) {  String original = "insecure";  original = original.replace('i', '9');  System.out.println(original);  }  } |

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

| **Principles(s):**  Principle #10 (Adopt a Secure Coding Standard): Creating rules and guidelines for verifying the success or failure of a function or method, such as using the Boolean value of false to indicate an operation within a method was unsuccessful, will allow for maintainable and adaptable code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | JAVA.NULL.RET.UNCHECKED  JAVA.FUNCS.IRV | Call Might Return Null (Java)  Ignored Return Value (Java) |
| Coverity | 7.5 | CHECKED\_RETURN | Implemented |
| Parasoft Jtest | 2023.1 | CERT.EXP00.NASSIG  CERT.EXP00.AECB | Ensure method and constructor return values are used  Avoid "try", "catch" and "finally" blocks with empty bodies |
| SonarQube | 9.9 | S2201  S899 | Return values from functions without side effects should not be ignored  Return values should not be ignored when they contain the operation status code |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Containers** | CTR-053-CPP | Use valid iterator ranges |

| **Noncompliant Code** |
| --- |
| 1.In this noncompliant code example, iterators from different containers are passed for the same iterator range. Although many STL implementations will compile this code and the program may behave as the developer expects, there is no requirement that an STL implementation treat a default-initialized iterator as a synonym for the iterator returned by end(). |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::vector<int>::const\_iterator e;  std::for\_each(c.begin(), e, [](int i) { std::cout << i; });  } |
| 2. In this noncompliant example, the two iterators that delimit the range point into the same container, but the first iterator does not precede the second. On each iteration of its internal loop, std::for\_each() compares the first iterator (after incrementing it) with the second for equality; as long as they are not equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| 1.In this compliant solution, the proper iterator generated by a call to end() is passed. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |
| 2. In this compliant solution, the iterator values passed to std::for\_each() are passed in the proper order. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):**  Principle #4 (Keep it simple): Using the already defined functions available in the C library as well as the libraries for other programming languages is preferrable over custom functions unless explicitly applicable. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| Helix QAC | 2023.1 | C++3802 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |
| Polyspace Bug Finder | R2023a | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered). |

### 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.

An automation factor can be established in the verify and test phase. We can implement powerful unit testing tools such as Junit5 or something similar of this such as QUnit. This will also allow us to incrementally develop our code using Test Driven Development and to break our application into testable sections. This will help build upon the assessment and planning stages after each round.

Furthermore, we can identify the dependencies that will be used in the application using static testing and update our list of dependency vulnerabilities to reduce potential risks when moving into the production phase.

Moving onto the production phase, we may automate our penetration tests. In specifically the domain of monitoring and detection, we can introduce log notifications and document them for future reference.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| INT-031-C | High | Probable | High | P6 | L2 |
| INT-030-C | High | Likely | High | P9 | L2 |
| STR-031-C | High | Likely | Medium | P18 | L1 |
| IDS-000-J | High | Likely | Medium | P18 | L1 |
| MEM-050-CPP | High | Likely | Medium | P18 | L1 |
| MSC-011-C | Low | Unlikely | High | P1 | L3 |
| ERR-051-CPP | Low | Probable | Medium | P4 | L3 |
| STR-002-C | High | Likely | Medium | P18 | L1 |
| EXP-000-J | Medium | Probable | Medium | P8 | L2 |
| CTR-053-CPP | High | Probable | High | P6 | 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 refers to encrypted data that is not being used. An example of this is passwords in a database or other sensitive information that may not always be in use or hard drives. With this said, consideration needs to be taken to provide encryption tools as well as disk encryption to keep data secure. |
| Encryption at flight | Encryption at flight is the process of encrypting data that is leaving or entering a specific part of the application. An example of this is sending a message that uses asymmetric encryption. It is important that the correct algorithm is chosen for each case. Encryption at flight can be between two devices within a network or the process of data being moved out of a network. |
| Encryption in use | Encryption in use is the encryption of security of data that is being used or accessed. An example of this is using a public key to decrypt a message. Sensitive data should be properly hashed using a powerful encryption method such as AES-256 so the 256 bit keeps information more secure. There should also be use of defense in depth during this stage. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of identifying the user trying to access the system. This can apply to user logins where a user supplies a username or password, and the system checks against those credentials. Companies are now gearing towards two-step verification to where you must be authenticated in two ways to access the system. An example of this is through a code sent to a user’s associated email or SMS messaging. |
| Authorization | Authorization is where the system determines the user’s level of access depending on their role within the system. An example of this is the are the available file access and privileges that an admin has on a website compared to a new user. Companies should be practicing the principle of least privilege in which each role has the least number of resources required to do their job. This will lessen the damage of a security breach and keep things uniform. |
| Accounting | Accounting is the process of keeping documentation and accountability for the system and its data. This can include changes to the database and the addition of new users. A log should be in place to keep track of data that was accessed, if it was edited in any way, and what user(s) accessed it. This also helps track unauthorized users who may or may not have infiltrated the system. |

**\***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 |