**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 input data should always require some form of validation to prevent exploits or other potential vulnerabilities such as buffer overflow or even SQL injection using malformed string input. |
| 1. Heed Compiler Warnings | Code compiler warnings will highlight any potential problems with your code, so the highest warning level available for your compiler should be used so you can remedy any issues in your code. |
| 1. Architect and Design for Security Policies | Software architecture and design should be done with security in mind at the start of development, it should be developed in ways that implements and enforce common security policies. This can be done through dividing a system into separate yet inter-communicating subsystems each with their own set of privileges. |
| 1. Keep It Simple | Design your code to be simple and concise, as more complex code design is more likely to be error prone or something would be overlooked during development. |
| 1. Default Deny | Set it so that, at a default, access to things is denied. To gain access to a system one must be given permission first. This means that instead of the possibility of overlooking a certain permission was denied or not, you can rest assured knowing that only those that have been given exclusive permission to something will have it. |
| 1. Adhere to the Principle of Least Privilege | Either a function, program, process in code, or even a user in a system should be set up to have the least set of privileges possible that would be needed to do their respective functions and no more. Limiting what something has access to that is not needed can reduce the chance an unexpected error can occur, or the damage that could be done because of it. |
| 1. Sanitize Data Sent to Other Systems | Any data being passed to other systems or subsystems should be sanitized. This is especially the case when dealing with potential SQL injection within data input being passed to as a parameter to a different subsystem as the subsystem being invoked would not have the context about the call being made. |
| 1. Practice Defense in Depth | Defense in Depth is about adding multiple different layers of security to a system. By adding multiple layers to a system, in the event one layer either is breached or is inadequate, another layer of defense can cover any potential security flaws or vulnerabilities. |
| 1. Use Effective Quality Assurance Techniques | Using good quality assurance techniques for finding and removing potential vulnerabilities. Examples such as penetration testing, and source code audits to create a more effective quality assurance program. Alternatively using security reviews, especially external or third-party reviewers to bring an independent perspective. |
| 1. Adopt a Secure Coding Standard | A secure coding standard should be developed or applied to the specific development language being used for the project. Different programming languages may have different vulnerabilities or coding standards to consider. |

*Secure coding practices: Digital security guide: Safeonline.ng : Digital Security Guide*. Digital Security Guide | Safeonline.ng. (2018, March 12). Retrieved November 10, 2022, from https://safeonline.ng/web-developers/secure-coding-practices/

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | When choosing a data type for your integers for example, it should be considered if you should use Unsigned Int, or a Signed Int. Unsigned integer means it cannot result in negative values, while Signed integers can become negative. The correct choice should be made when choosing signed vs unsigned as to prevent wrapping.  Additionally, you should ensure that operations on signed integers do not result in overflow (or underflow). In one example to prevent overflow, it can be done by comparing the difference between something like **size\_t** vs using **unsigned int**, as **size\_t** will be able to store more numbers than **int** |

| **Noncompliant Code** |
| --- |
| Noncompliant code example can result in a signed integer overflow during multiplication of the signed operand si\_a and si\_b |
| Void func(signed int si\_a, signed int si\_b) {  Signed int result = si\_a \* si\_b;  /\* . . . \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates signed overflow on systems where **long long** is at least twice the precision of int |
| extern size\_t popcount(uintmax\_t);  #define PRECISION(umax\_value) popcount(umax\_value)    void func(signed int si\_a, signed int si\_b) {    signed int result;    signed long long tmp;    assert(PRECISION(ULLONG\_MAX) >= 2 \* PRECISION(UINT\_MAX));    tmp = (signed long long)si\_a \* (signed long long)si\_b;      /\*     \* If the product cannot be represented as a 32-bit integer,     \* handle as an error condition.     \*/    if ((tmp > INT\_MAX) || (tmp < INT\_MIN)) {      /\* Handle error \*/    } else {      result = (int)tmp;    }    /\* ... \*/  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow**

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

| **Principles(s):**  1 - Validate Input Data: Use the correct data type when handling input. (Avoid overflows)  2 - Heed Compiler Warnings: Compiler warnings can alert of when a data type declaration is incorrect.  3 – Architect and Design for Security Policies: Understanding and designing with security in mind can prevent most of these issues from occurring in the first place. (Comes down to experience and knowledge)  9 – Use Effective Quality Assurance Techniques: Check for errors by testing the code for any potential vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | Integer-overflow | Fully Checked |
| CodeSonar | 7.1p0 | ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-INT30-a CERT\_C-INT30-b CERT\_C-INT30-c | Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | As mentioned within Data Types, using proper ranging of integer data types should be used to prevent wrapping. Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrapping during an addition operation. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution preforms a precondition test to ensure there is no possibility of an unsigned integer wrapping |
| 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;    }    /\* ... \*/  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap**

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

| **Principles(s):**  1 - Validate Input Data: Ensure the input is within possible range to prevent out of bound or wrapping.  2 - Heed Compiler Warnings: Compiler warnings should show if the data value is out of bounds. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Parasoft C/C++ test | 2022.1 | **CERT\_C-INT30-a CERT\_C-INT30-b CERT\_C-INT30-c** | Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2022b | CERT C: Rule INT30-C | Checks for:   * Unsigned integer overflow * Unsigned integer constant overflow   Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Ensure that storage for strings has sufficient space for character data, as copying data to a buffer that is not large enough to hold that data will result in a buffer overflow |

| **Noncompliant Code** |
| --- |
| This noncompliant code results in a buffer overflow if the input is longer than 20 characters |
| #include <iostream>  Int main() {  char user\_input[20];  std::cin >> user\_input;  } |

| **Compliant Code** |
| --- |
| A simple solution to this problem checks if the data input is within the bounds of the character data type by using std::cin.getline() |
| #include <iostream>  int main() {  char user\_input[20];  if (std::cin.getline(user\_input, 20)) {  /\* . . . \*/  }  else {  std::cout << "Character length limit exceeded" << std::endl;  abort();  }  } |

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

| **Principles(s):**  1 – Validate Input Data: Do not allow user input to exceed string length  2 – Heed Compiler Warnings: Compiler warnings are likely to show if a string value may result in a buffer overflow.  7 – Sanitize Data Sent to Other Systems: String data should be sanitized when passed as parameters to other systems to prevent SQL injection.  8 – Practice Defense in Depth: Verifying the input is within the bounds to prevent SQL injection provides some level of protection for defense in depth. |
| --- |

**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 | 7.1p0 | MISC.MEM.NTERM  LANG.MEM.BO LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR50-b CERT\_CPP-STR50-c CERT\_CPP-STR50-e CERT\_CPP-STR50-f CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2022b | 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. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CLG] | SQL injection can occur that exploits, for example, a password or username field field to generate a Boolean expression which evaluates the expression to true. This can and should be mitigated against by using “prepared statements” for querying |

| **Noncompliant Code** |
| --- |
| Through this noncompliant code, the user passing in input can directly preform an SQL injection by including a ‘name = 1’ or 1 = 1. Thus, resulting in a SQL injection that makes a Boolean expression to true |
| name = getRequestString(“username”);  pass = getRequestString(“password”);  sql = “SELECT \* FROM Users WHERE Name = “ + name + “ AND Pass = “ + pass + “ |

| **Compliant Code** |
| --- |
| You can use prepared statements to mitigate against potential SQL injection |
| PreparedStatement pStatement = PreparedStatement();  std::cin >> username;  std::cin >> password;  sql = (“SELECT \* FROM Users WHERE Name = %s AND Pass = %s:”, username, password) |

Information found at: **https://www.geeksforgeeks.org/mitigation-sql-injection-attack-using-prepared-statements-parameterized-queries/**

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

| **Principles(s):**  1 – Validate Input Data: Validating input to prevent the user from performing an SQL Injection.  3 – Architect and Design for Security Policies: Prepared statements can prevent against SQL injection.  7 – Sanitize Data Sent to Other Systems: A prepared statement would sanitize the data to prevent or mitigate against SQL injection attempts.  8 – Practice Defense in Depth: Prepared statements will add a layer of security for defense in depth. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | 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 |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully Implemented |
| Parasoft C/C++ test | 2022.1 | **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 |
| Polyspace Bug Finder | R2022b | CERT C: Rule STR-31-C | 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. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Avoid accessing freed memory, as pointers to memory that has been deallocated are called dangling pointers and accessing these can result in exploitable vulnerabilities.  Here are some additional rules related to memory:  MEM50-CPP. Do not access freed memory MEM51-CPP. Properly deallocate dynamically allocated resources  MEM31-C. Free dynamically allocated memory when no longer needed |

| **Noncompliant Code** |
| --- |
| In this noncompliant code **s** is dereferenced after is has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant code 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;  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory**

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

| **Principles(s):**  9 – Use Effective Quality Assurance Techniques: Testing should be done to ensure there are no freed memory issues like dangling pointers which may be overlooked (There are no compiler warnings for this issue) |
| --- |

**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 | N/A |
| CodeSonar | 7.1p0 | ALLOC.UAF | Use after free |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Polyspace Bug Finder | R2022b | 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** | [STD-006-CPP] | Assertions should be used by programmers to test assumptions like if a pointer return is NULL or not. Then, if the expression evaluates to false, the **abort()** function is called. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the **assert()** macro to verify that the memory allocation succeeded. |
| 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 shows 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;  } |

Information found at: **https://www.geeksforgeeks.org/assertions-cc/**

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions**

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

| **Principles(s):**  9 – Use Effective Quality Assurance Techniques: Using assertions, you can test your code, like functions, to ensure they are working correctly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | 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 | 2022.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching expectation handler. Exceptions should be used to check for errors that are either made by the programmer or errors due to wrong input.  Additional rules related to exceptions:  ERR51-CPP. Handle all exceptions  ERR55-CPP. Honor exception specifications |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither **f()** nor **main()** catch exceptions thrown by **throwing\_func().** Due to no matching handler being found for the exception thrown. This leads to **std::terminate()** being called |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant code solution, the main entry point handles all exceptions. This also incorporates a try/catch code block |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

Information found at: **https://www.geeksforgeeks.org/exception-handling-c/?ref=gcse**

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions**

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

| **Principles(s):**  3 – Architect and Design for Security Policies: Implement into your code try/catch blocks to handle errors  9 – Use Effective Quality Assurance Techniques: Your code should be tested using exception throws incase your code is unable to do what it is supposed to like behaving in unexpected ways. |
| --- |

**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 | 7.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-ERR51-a CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly through in code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2022b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input / Output | [STD-008-CCP] | Never call a formatted I/O function without a sanitized format string, as attackers who can fully or either partially control the contents of a format string will be able to crash a vulnerable process, view the contents of a stack, or write an arbitrary memory location. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, the **incorrect\_password()** function is called during identification and authentication to display an error message if the user is not found or password is incorrect. This function accepts the name of the user as a string referenced by users and is an example of untrusted data. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fprintf(stderr, msg);    free(msg);  } |

| **Compliant Code** |
| --- |
| This compliant code solution fixes the problem by replacing the **fprintf()** call with a **fputs()** call, which outputs msg directly without evaluating its contents. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fputs(msg, stderr);    free(msg);  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings**

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

| **Principles(s):**  7 – Sanitize Data Sent to Other Systems: Do not accept unsanitized data for function calls, this also relates to validating user input by not allowing the users to pass in data that would be considered unsafe or unsanitized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | IO.INJ.FMT  MISC.FMT | Format String Injection  Format String |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-FIO30-a CERT\_C-FIO30-b CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable Never use unfiltered data from an untrusted user as the format parameter |
| Polyspace Bug Finder | R2022b | CERT C: Rule FI030-C | Checks for tainted string format (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Ensure that array references within the bounds of an array, and that integer indexes are within the bounds of the vector  Ensure the use of valid references, pointer, and iterators to reference elements of a container. As storing an iterator, reference, or pointer to an element within a container for any length of time comes with a risk of that container being modified which makes these iterators, pointers, or references to become invalid.  Here are the associated rules: CTR50-CPP. Guarantee that container indices and iterators are within valid range CTR50-CPP. Use valid references, pointers, and iterators to reference elements of a container. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows the function **insert\_in\_table()** that has two int parameters (**pos**, and **value**) both of which can be influenced by data originating from untrusted sources. This function preforms a range check on the upper bound but does not check the lower bound of the array. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the parameter **pos** is declared as **size**\_t which prevents the passing of negative arguments |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range**

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

| **Principles(s):**  3 – Architect and Design for Security Policies: Ensure that when coding, you do not create code that would go beyond valid range and cause an overflow |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference | N/A |
| CodeSonar | 7.1p0 | LANG.MEM.BO LANG.MEM.BU LANG.MEM.TO LANG.MEM.TU LANG.MEM.TBA LANG.STRUCT.PBB LANG.STRUCT.PPE LANG.STRUCT.PARITH | Buffer overrun Buffer underrun Type overrun Type underrun Tainted buffer access Pointer before beginning of object Pointer past end of object Pointer Arithmetic |
| Paraosft C/C++ test | 2022.1 | CERT\_CPP-CTR50-a | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory (Allocation) | [STD-010-CPP] | (Expanded standard from Memory Protection)  It is important to check which return value is being passed by either “**::operator new(std::size\_t)**”, which throws a **std::bad\_alloc** exception, or the nonthrowing form “**::operator new(std::size\_t, const std::nonthrow\_t &)**” which returns a **nullptr.**  When using the nonthrowing form, it is imperative to check that the return value is not **nullptr** before accessing the resulting pointer. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, an array of int is created using the **::operator new[](std::size\_t)** and the results of the allocation are not checked. The function is marked as **noexcept**, so the caller assumes this function does not throw any exceptions. |
| #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** |
| --- |
| In this compliant code, when using **std::nothrow** the new operator returns either a null pointer or a pointer to the allocated space. This solution handles the error condition appropriately when returned pointer is **nullptr**. |
| #include <cstring>  #include <new>    void f(const int \*array, std::size\_t size) noexcept {    int \*copy = new (std::nothrow) int[size];    if (!copy) {      // Handle error      return;    }    std::memcpy(copy, array, size \* sizeof(\*copy));    // ...    delete [] copy;  } |

Solution found at: **https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors**

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

| **Principles(s):**  9 – Use Effective Quality Assurance Techniques: Tests should be performed to ensure functions are properly set up to handle exceptions, instead of incorrectly labeling no exceptions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function’s parameters is undefined. |
| Polyspace Bug Finder | R2022b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (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.

During the “build” and especially “Verify and test” phase of pre-production, automatic static code analysis tools can be implemented to test code for bugs or other known vulnerabilities before the software reaches production stage.

As for when in production, automation can be implemented during the “Monitor and detect” phase, and the “Maintain and stabilize” phase, during Monitoring you can use automation to test for any newer threats that may arise like before unknown bugs and vulnerabilities. And once in the Maintain and the potential threats have ben resolved, before exiting the production process you can ensure all other security flaws and other vulnerabilities are truly accounted for.

### 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 | Likely | High | P9 | L2 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CLG | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### 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 when you are encrypting data that is being stored (potentially archived and no longer actively being used). This is done to protect the data while it is being stored either physically or within the cloud. |
| Encryption at flight | Encryption at flight is when you are actively transferring data from one point to the next, sometimes to other recipients. Encrypting the data while it is undergoing this moment protects it so it is not intercepted or stolen, even if just to see the contents of such data |
| Encryption in use | Encryption in use is when you have data that is actively being used, either the data is being edited, erased, viewed, or created. This is the sort of data that should be encrypted as it is at a higher risk for and may be a target of data breaches. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is what verifies a user’s identity, and that a specific user is who they claim to be. This can be done using username/password for verification of users and expanded to have something like a 2-factor authentication to those user accounts. |
| Authorization | Authorization is what controls what a user is capable doing, or what resources they have access to. An example of this is a normal user may have only read access to files, while a higher-ranking user (possibly administrator) would have full read/write access. |
| Accounting | Accounting, or auditing, is a record or log of data that would show what recourses were accessed, at what time, by which user and so on. Every action done by a user has a trace which is logged. This is done to essentially keep track of what is always going on, and by who. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.2 | 12/1/2022 | Finished Template | Andrew McPherson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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