**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 the process of analyzing user inputs and not allowing an input that could potentially cause harm to the network. This is generally used on any website or application that allows user input. Input validation reduces the chances of buffer overflow attacks, SQL injections, and cross-cite scripting. |
| 1. Heed Compiler Warnings | Heed compiler warnings by using static and dynamic analysis tools to identify any security vulnerabilities. By modifying code based on the results of the detection tools, the programmer can eliminate threats to the network or application. |
| 1. Architect and Design for Security Policies | Architect and design for security policies provide a set of rules for the organization’s cyber security, roles, and responsibilities in maintaining information and technology. This is done by assessing the environment to determine what risks and vulnerabilities there are and what countermeasures can be taken for every person who interacts with the system or network. |
| 1. Keep It Simple | The code should be kept simple and small as possible to reduce the number of components used. This minimizes attack surface and reduces any entry point for an attacker to get into the program. Keeping the program simple can be done by disabling unused functionalities and using minimal installs. |
| 1. Default Deny | Deny by default denies users and entities access unless specifically authorized. All inbound and outbound traffic is blocked unless permitted by the firewall. This protects from attacks that target unused ports in the firewall. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege is that users, programs, and processes should have the bare minimum privileges to complete their tasks. Allowing only enough access to perform the specific task reduces the chance of attackers gaining access. This protects critical systems and data from low-level user accounts, devices, and applications where an attacker can get in from. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization ensures that data conforms to the subsystem it is passed to by making sure that the security requirements for the data are met to reduce chances of exposure of sensitive data when data is sent across a trust boundary. |
| 1. Practice Defense in Depth | Defense-in-Depth (DiD) is a strategy which uses multiple layers of security measures to protect an organization’s assets. When using multiple security measures, it decreases any chances of attacks against the network or even hardware on the physical layer. Since it utilizes multiple layers of defense, if one defensive measure is compromised, there are other layers to protect it. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques apply a layer of security by making sure that the code meets quality and security checks. Quality assurance helps identify potential vulnerabilities within the system. The programmer can then make changes to protect against attacks to the program. |
| 1. Adopt a Secure Coding Standard | Secure coding standards provide rules and guidelines for coding that prevent, detect, and eliminate security vulnerabilities. Using coding standards helps make sure that all vulnerabilities are known, and the programmer can adjust code accordingly. |

### 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] | Ensure that integer conversions do not result in lost or misinterpreted data. |

| **Noncompliant Code** |
| --- |
| Loss of data and loss of sign errors can occur when converting 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 \*/  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed 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 \*/  }  } |

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

| **Principles(s):** Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.CAST.PC.AV  LANG.CAST.PC.CONST2PTR  LANG.CAST.PC.INT  LANG.CAST.COERCE  LANG.CAST.VALUE  ALLOC.SIZE.TRUNC  MISC.MEM.SIZE.TRUNC  LANG.MEM.TBA | Cast: arithmetic type/void pointer  Conversion: integer constant to pointer  Conversion: pointer/integer  Coercion alters value  Cast alters value  Truncation of allocation size  Truncation of size  Tainted buffer access |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| LDRA tool suite | 9.7.1 | 93 S, 433 S, 434 S | Partially implemented |
| 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. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that division and remainder operations do not result in divide-by-zero errors. |

| **Noncompliant Code** |
| --- |
| Prevents a signed integer overflow but fails to prevent a divide-by-zero error during the division of the signed operands. |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_a == LONG\_MIN) && (s\_b == -1)) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Tests the division operation to guarantee there is no possibility of divide-by-zero errors or signed overflow. |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

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

| **Principles(s):** The result of the / operator is the quotient from the division of the first arithmetic operand by the second arithmetic operand. Division operations are susceptible to divide-by-zero errors. Overflow can also occur during two's complement signed integer division when the dividend is equal to the minimum (most negative) value for the signed integer type and the divisor is equal to −1. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.ARITH.DIVZERO  LANG.ARITH.FDIVZERO | Division by zero  Float Division By Zero |
| Coverity | 2017.07 | DIVIDE\_BY\_ZERO | Fully implemented |
| Cppcheck | 1.66 | zerodiv  zerodivcond | Context sensitive analysis of division by zero  Not detected for division by struct member / array element / pointer data that is 0  Detected when there is unsafe division by variable before/after test if variable is zero |
| Parasoft C/C++ test | 2023.1 | CERT\_C-INT33-a | Avoid division by zero |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to modify string literals. |

| **Noncompliant Code** |
| --- |
| The char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| char str[] = "string literal";  str[0] = 'S'; |

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

| **Principles(s):** At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| LDRA tool suite | 9.7.1 | 157 S | Partially implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified  Do not modify string literals |
| RuleChecker | 23.04 | string-literal-modfication | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Exclude user input from format strings. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by the user. |
| #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** |
| --- |
| By replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr 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);  } |

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

| **Principles(s):** Never call a formatted I/O function with a format string containing a tainted value. An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| LDRA tool suite | 9.7.1 | 86 D | Partially Implemented |
| Parasoft C/C++ test | 2023.1 | 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 |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Allocate sufficient memory for an object. |

| **Noncompliant Code** |
| --- |
| Inadequate space is allocated for a struct tm object because of the size of the pointer is being used to determine the size of the pointed-to object. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

| **Compliant Code** |
| --- |
| The correct amount of memory is allocated for the struct tm object. When allocating space for a single object, passing the pointer type to the sizeof operator is a simple way to allocate sufficient memory. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(\*tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

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

| **Principles(s):** The types of integer expressions used as size arguments to malloc(), calloc(), realloc(), or aligned\_alloc() must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  ALLOC.SIZE.TRUNC  IO.TAINT.SIZE  MISC.MEM.SIZE.BAD  LANG.MEM.BO  LANG.MEM.BU  LANG.STRUCT.PARITH  LANG.STRUCT.PBB  LANG.STRUCT.PPE  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Addition overflow of allocation size  Addition overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Truncation of allocation size  Tainted allocation size  Unreasonable size argument  Buffer Overrun  Buffer Underrun  Pointer Arithmetic  Pointer Before Beginning of Object  Pointer Past End of Object  Tainted Buffer Access  Type Overrun  Type Underrun |
| Coverity | 2017.07 | BAD\_ALLOC\_STRLEN  SIZECHECK (deprecated) | Partially implemented  Can find instances where string length is miscalculated (length calculated may be one less than intended) for memory allocation purposes. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary  Finds memory allocations that are assigned to a pointer that reference objects larger than the allocated block |
| LDRA tool suite | 9.7.1 | 400 S, 487 S, 115 D | Enhanced enforcement |
| Parasoft C/C++ test | 2023.1 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Detect errors when converting a string to a number. |

| **Noncompliant Code** |
| --- |
| Multiple numeric values are converted from the standard input stream. If the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected. |
| #include <iostream>    void f() {  int i, j;  std::cin >> i >> j;  // ...  } |

| **Compliant Code** |
| --- |
| Exceptions are enabled so that any conversion failure results in an exception being thrown. This approach cannot distinguish between which values are valid and which are invalid and must assume that all values are invalid. Both the badbit and failbit flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions. |
| #include <iostream>    void f() {  int i, j;    std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try {  std::cin >> i >> j;  // ...  } catch (std::istream::failure &E) {  // Handle error  }  } |

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

| **Principles(s):** The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than INT\_MAX). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a C Standard Library function. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | BADFUNC.ATOF  BADFUNC.ATOI  BADFUNC.ATOL  BADFUNC.ATOLL  (customization) | Use of atof  Use of atoi  Use of atol  Use of atoll  Users can add custom checks for uses of other undesirable conversion functions. |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-ERR34-a | The library functions atof, atoi and atol from library stdlib.h shall not be used |
| PC-lint Plus | 1.4 | 586 | Assistance provided |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions thrown before main() begins executing. |

| **Noncompliant Code** |
| --- |
| The constructor for S may throw an exception that is not caught when globalS is constructed during program startup. |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| Making globalS into a local variable with static storage duration, allows exceptions thrown during object construction to be caught because of the constructor for S will be executed for the first time the function globalS() is called rather than at the program startup. |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** When declaring an object with static or thread storage duration, and that object is not declared within a function block scope, the type's constructor must be declared noexcept and must comply with ERR55-CPP. Honor exception specifications. Additionally, the initializer for such a declaration, if any, must not throw an uncaught exception (including from any implicitly constructed objects that are created as a part of the initialization). If an uncaught exception is thrown before main() is executed, or if an uncaught exception is thrown after main() has finished executing, there are no further opportunities to handle the exception and it results in implementation-defined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | cert-err58-cpp | Checked by clang-tidy |
| CodeSonar | 7.4p0 | LANG.STRUCT.EXCP.THROW | Use of throw |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program |
| Polyspace Bug Finder | R2023a | CERT C++: ERR58-CPP | Checks for exceptions raised during program startup (rule fully covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | [STD-008-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| 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. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| 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;  } |

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

| **Principles(s):** Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| 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 |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Correctness | [STD-009-CPP] | Do not attempt to create a std::string from a null pointer. |

| **Noncompliant Code** |
| --- |
| A std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist. |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| The results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    void f() {  const char \*tmpPtrVal = std::getenv("TMP");  std::string tmp(tmpPtrVal ? tmpPtrVal : "");  if (!tmp.empty()) {  // ...  }  } |

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

| **Principles(s):** The std::basic\_string type uses the traits design pattern to handle implementation details of the various string types, resulting in a series of string-like classes with a common, underlying implementation. Specifically, the std::basic\_string class is paired with std::char\_traits to create the std::string, std::wstring, std::u16string, and std::u32string classes. The std::char\_traits class is explicitly specialized to provide policy-based implementation details to the std::basic\_string type. One such implementation detail is the std::char\_traits::length() function, which is frequently used to determine the number of characters in a null-terminated string. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.4p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Klocwork | 2023.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2023a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Value | [STD-010-CPP] | Do not cast to an out-of-range enumeration value. |

| **Noncompliant Code** |
| --- |
| This example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting the enumeration type, which may not be able to represent the given integer value. If a value outside of the range are passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| Checking the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principles(s):** Enumerations in C++ come in two forms: scoped enumerations in which the underlying type is fixed and unscoped enumerations in which the underlying type may or may not be fixed. The range of values that can be represented by either form of enumeration may include enumerator values not specified by the enumeration itself. The range of valid enumeration values for an enumeration type is defined by the C++ Standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | cast-integer-to-enum | Partially checked |
| CodeSonar | 7.4p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation can be done in pre-production and production phases. The process to automate enforcement of the standards can be done in the design, build, verify and test phases of the pre-production. This will ensure that the standards are being enforced throughout the development of the project. Automation tools can be used during development. In the production phase, they can be used to maintain and stabilize. This will help detect vulnerabilities.

### 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 | Probable | High | P6 | L2 |
| STD-002-CPP | Low | Likely | Medium | P6 | L2 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Probable | High | P6 | L2 |
| STD-006-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | High | Likely | Medium | P18 | L1 |
| STD-010-CPP | Medium | Unlikely | Medium | P4 | L3 |

### 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 provides protection for stored data. It prevents an attacker from accessing unencrypted data by making sure it is encrypted when it is stored. It should be used as an extra layer of protection. If someone were to obtain physical access to the data, they would not be able to access the data itself as easily. |
| Encryption at flight | Encryption at flight is the encryption of data that moves over a network. This protects the data while it is being transmitted. It should be used so the data is protected in case it gets intercepted. |
| Encryption in use | Encryption in use encrypts data that is in use when accessed by a user or application. This helps protect against cyber attacks since data that is in use is the most vulnerable form of data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is when a user can prove who they are providing information. This information can be a username and password, keycard, or biometrics. This is to make sure only authorized users are allowed certain levels of access to protect against attacks. |
| Authorization | Authorization is what determines what the user has access to. For example, there would be user privileges or administrator privileges. The administrator would have a higher level of authorization. This helps protect against attacks but can also prevent negligence. |
| Accounting | Accounting keeps track of user activity while they are logged in. Accounting may be used to analyze user trends and audit user activity. This policy applies because if a change was made to the network or program there would be a way to see which user was making what change at what time. |

**\***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 |