**Green Pace Developer: Security Policy**



# 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 | Validate input data, or input validation, is testing each untrusted input to ensure the information provided fits the criteria and is not improperly formed. In the case that input validation is not included, an attacker could maliciously input with the intention to confuse the system and gain entrance through that vulnerability. |
| 1. Heed Compiler Warnings | It is important to take notice of all complier warnings and to correct the code where the warnings were based on. In the case that these complier warnings are ignored, it gives a vulnerability to the system. Preferably use the highest warning level available. |
| 1. Architect and Design for Security Policies | When creating the architecture and design for the software, you must abide by the most up-to-date secure policies. Otherwise, there is a higher risk of vulnerabilities within the software. |
| 1. Keep It Simple | In most cases, you want to keep the software design simple. Complex designs require more effort in security as well as increases the possibility of errors. |
| 1. Default Deny | Access is always defaulted to deny entry unless they have identified permissions. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege allows only enough permissions/access to complete that intended function. While somewhat similar to default deny, this gives the user limited permissions to only access what they need. Otherwise, permissions are denied to reduce unauthorized use. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data, which is removing and/or destroying data that is saved in memory, will help prevent injection attacks when sending to other systems. |
| 1. Practice Defense in Depth | Defense in Depth, also known as DiD, is a layered defense against security attacks. For further clarification, it is putting many different security measures in place so that if a hacker gets through one layer/defense, there will be others in place to hold them back. |
| 1. Use Effective Quality Assurance Techniques | Test and debug to ensure all bugs are identified and fixed. Additionally, it may be beneficial to run a vulnerability scan and complete integration testing. For more assurance, having an outside source from the team will allow assumption errors to be caught. |
| 1. Adopt a Secure Coding Standard | Research the most up-to-date secure coding standards for your language and apply them to your development process. |

### 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.

\*All coding standards information is taken or referenced from*SEI CERT C++ Coding Standard*

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Do not cast to an out-of-range enumeration value // INT50-CPP |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were 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** |
| --- |
| This compliant solution checks that 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 for 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):** 1, 2, 9. To avoid operating on unspecified values, the arithmetic value being cast must be within the range of values the enumeration can represent. When dynamically checking for out-of-range values, checking must be performed before the cast expression. Will most likely result in data integrity violations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| CodeSonar | 6.2p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| Helix QAC | 2021.2 | C++3013 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PRQA QA-C++ | 4.4 | 3013 |  |
| PVS-Studio | 7.17 | V1016 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not rely on the value of a moved-from object // EXP63-CPP. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the integer values 0 through 9 are expected to be printed to the standard output stream from a std::string rvalue reference. However, because the object is moved and then reused under the assumption its internal state has been cleared, unexpected output may occur despite not triggering undefined behavior. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  std::string s;  for (unsigned i = 0; i < 10; ++i) {  s.append(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::string object is initialized to the expected value on each iteration of the loop. This practice ensures that the object is in a valid, specified state prior to attempting to access it in g(), resulting in the expected output. |
| #include <iostream>  #include <string>    void g(std::string v) {  std::cout << v << std::endl;  }    void f() {  for (unsigned i = 0; i < 10; ++i) {  std::string s(1, static\_cast<char>('0' + i));  g(std::move(s));  }  } |

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

| **Principles(s):** 1, 9. While usually moved-from objects are in a valid state, unspecified values produce unspecified behavior – such as termination. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2021.2 | C++4701, C++4702, C++4703 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-EXP63-a | Do not rely on the value of a moved-from object |
| Polyspace Bug Finder | R2021b | CERT C++: EXP63-CPP | Checks for read operations that reads the value of a moved-from object (rule fully covered) |
| PVS-Studio | 7.17 | V1030 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Do not attempt to create a std::string from a null pointer // STR51-CPP |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, 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 (or some other error occurs). |
| #include <cstdlib>  #include <string>    void f() {  std::string tmp(std::getenv("TMP"));  if (!tmp.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, 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):** 2, 4, 9. Passing a null pointer to this function is undefined behavior because it would result in dereferencing a null pointer. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure |  |
| Helix QAC | 2021.2 | C++4770, C++4771, C++4772, C++4773, C++4774 |  |
| Klocwork | 2021.4 | 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 | 2021.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Exclude user input from format strings // FIO30-C, also applies to CPP |

| **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 user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| #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 solution fixes the problem 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):** 1, 5, 6, 8, 9. Excluding user input from formatted strings is important. 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. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 6.2p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Compass/ROSE |  |  |  |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| GCC | 4.3.5 |  | Can detect violations of this rule when the -Wformat-security flag is used |
| Helix QAC | 2021.3 | C4916, C4917, C4918  C++4916, C++4917, C++4918 |  |
| Klocwork | 2021.4 | SV.FMTSTR.GENERIC  SV.TAINTED.FMTSTR |  |
| LDRA tool suite | 9.7.1 | 86D | Partially Implemented |
| Parasoft C/C++ | 2021.2 | 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 |
| PC-lint Plus | 1.4 | 592 | Partially supported: reports non-literal format strings |
| Polyspace Bug Finder | R2021a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |
| PRQA QA-C | 9.7 | 4916, 4917, 4918 |  |
| PRQA QA-C++ | 4.4 | 4916, 4917, 4918 |  |
| PVS-Studio | 7.17 | V618 |  |
| Splint | 3.1.1 |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Properly deallocate dynamically allocated resources // MEM51-CPP |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the local variable space is passed as the expression to the placement new operator. The resulting pointer of that call is then passed to ::operator delete(), resulting in undefined behavior due to ::operator delete() attempting to free memory that was not returned by ::operator new(). |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;  // ...  delete s1;  } |

| **Compliant Code** |
| --- |
| This compliant solution removes the call to ::operator delete(), instead explicitly calling s1's destructor. This is one of the few times when explicitly invoking a destructor is warranted. |
| #include <iostream>    struct S {  S() { std::cout << "S::S()" << std::endl; }  ~S() { std::cout << "S::~S()" << std::endl; }  };    void f() {  alignas(struct S) char space[sizeof(struct S)];  S \*s1 = new (&space) S;  // ...  s1->~S();  } |

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

| **Principles(s):** 3, 6, 9; maybe 7. Deallocating a pointer that is not allocated dynamically is undefined behavior because the pointer value was not obtained by an allocation function. Deallocating a pointer that has already been passed to a deallocation function is undefined behavior because the pointer value no longer points to memory that has been dynamically allocated. This can lead to exploitable vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | invalid\_dynamic\_memory\_allocation  dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM51 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 6.2p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM | Free non-heap variable  Double free  Type mismatch |
| Helix QAC | 2021.2 | C++2110, C++2111, C++2112, C++2113, C++2118, C++3337, C++3339, C++4262, C++4263, C++4264 |  |
| Klocwork | 2021.4 | C L.FFM.ASSIGN  CL.FFM.COPY  CL.FMM  CL.SHALLOW.ASSIGN  CL.SHALLOW.COPY  FMM.MIGHT  FMM.MUST  FNH.MIGHT  FNH.MUST  FUM.GEN.MIGHT  FUM.GEN.MUST  UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST |  |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |
| Parasoft insure++ |  |  | Runtime Detection |
| Polyspace Bug Finder | R2021b | CERT C++: MEM51-CPP | Checks for:  Invalid deletion of pointer  Invalid free of pointer  Deallocation of previously deallocated pointer  Rule partially covered. |
| PRQA-C++ | 4.4 | 2110, 2111, 2112, 2113, 2118,  3337, 3339, 4262, 4263, 4264 |  |
| PVS-Studio | 7.17 | V515, V554, V611, V701, V748, V773, V1066 |  |
| SonarQube C/C++ Plugin | 4.10 | S1232 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use a static assertion to test the value of a constant expression // DCL03-C |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| This portable compliant solution uses static\_assert. Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. An assertion can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** 2, 4, 9. Static assertion is used to find and eliminate software defects, but does not necessarily mean the code is incorrect; identifying incorrect assumptions and not for runtime error checking. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 6.2p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions thrown before main() begins executing // ERR58-CPP |

| **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** |
| --- |
| This compliant solution makes globalS into a local variable with static storage duration, allowing any exceptions thrown during object construction to be caught because the constructor for S will be executed the first time the function globalS() is called rather than at program startup. This solution does require the programmer to modify source code so that previous uses of globalS are replaced by a function call to globalS(). |
| 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):** 2, 3, 9, 10. 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. This can result in denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Potentially-throwing-static-initialization | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR58 |  |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Helix QAC | 2021.2 | C++4634, C++4636, C++4637, C++4639 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program |
| Polyspace Bug Finder | R2021b | CERT C++: ERR58-CPP | Checks for exceptions raised during program startup (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4634, 4636, 4637, 4639 |  |
| RuleChecker | 20.10 | Potentially-throwing-static-initialization | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Characters and Strings | STD-008-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator // STR50-CPP |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** 1, maybe 3. To prevent buffer overflow errors, ensure that the destination is of sufficient size to hold the data to be copied. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Helix QAC | 2021.2 | C++2835, C++2836, C++2839, C++5216 |  |
| Klocwork | 2021.4 | NNTS.MIGHT  NNTS.TAINTED  NNTS.MUST  SV.UNBOUND\_STRING\_INPUT.CIN |  |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2021b | 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  Rule partially covered. |
| SonarQube C/C++ Plugin | 4.10 | S3519 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-009-CPP | Value-returning functions must return a value from all exit paths // MSC52-CPP |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):** 2, 3, 4, 9. A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Return-implicit | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 |  |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as *function-try-blocks* |
| CodeSonar | 6.2p0 | LANG.STRUCT.MRS | Missing return statement |
| Helix QAC | 2021.2 | C++2888 |  |
| Klocwork | 2021.4 | FUNCRET.GEN  DUNCRET.IMPLICIT |  |
| LDRA tool suite | 9.7.1 | 2 D, 36 S | Fully implemented |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MSC52-a | All exit paths from a function with non-void return type shall have an explicit return statement with an expression |
| Polyspace Bug Finder | R2021b | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |
| SonarQube C/C++ Plugin | 4.10 | S935 |  |
| PRQA QA-C++ | 4.4 | 1510 |  |
| PVS-Studio | 7.17 | V591 |  |
| RuleChecker | 20.10 | Return-implicit | Fully checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | STD-010-CPP | Guarantee that library functions do not overflow // CTR52-CPP |

| **Noncompliant Code** |
| --- |
| STL containers can be subject to the same vulnerabilities as array data types. The std::copy() algorithm provides no inherent bounds checking and can lead to a buffer overflow. In this noncompliant code example, a vector of integers is copied from src to dest using std::copy(). Because std::copy() does nothing to expand the dest vector, the program will overflow the buffer on copying the first element. |
| #include <algorithm>  #include <vector>    void f(const std::vector<int> &src) {  std::vector<int> dest;  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

| **Compliant Code** |
| --- |
| The proper way to use std::copy() is to ensure the destination container can hold all the elements being copied to it. This compliant solution enlarges the capacity of the vector prior to the copy operation. |
| #include <algorithm>  #include <vector>  void f(const std::vector<int> &src) {  // Initialize dest with src.size() default-inserted elements  std::vector<int> dest(src.size());  std::copy(src.begin(), src.end(), dest.begin());  // ...  } |

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

| **Principles(s):** 1, 9; maybe 3. Copying data to a buffer that is too small to hold the data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Atree | 20.10 | Invalid-pointer-deference |  |
| CodeSonar | 6.2p0 | BADFUNC.BO.\*  LANG.MEM.BO | A collection of warning classes that report uses of library functions prone to internal buffer overflows.  Buffer Overrun |
| Helix QAC | 2021.2 | C++3526, C++3527, C++3528, C++3529, C++3530, C++3531, C++3532, C++3533, C++3534 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-CTR52-a | Do not pass empty container iterators to std algorithms as destinations |

### 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.

DevOps uses continuous integration/ continuous delivery (CI/CD) and uses infrastructure as code, while DevSecOps includes vulnerability testing and threat modeling. Additionally, most security in DevOps is addressed at the end of development and DevSecOps surveys the security throughout the entirety of development (start to finish). When organizations are looking to change to DevSecOps, there are a few best practices which include – making automation your friend, check your code for dependencies, don’t bite off more than others can chew, some tools are more useful than others, threat modeling is hard but important to do, and of course, ensure your developers understand and can implement secure coding (Vijayan, n.d.). The first step of DevSecOps is to assess and plan. First and foremost it is important to look at the requirements, determine where you believe vulnerabilities may lay within them before development, how to pre-prevent issues, and more. Following that, you will begin to design the product to fit those criteria’s. Now, the developers will put the preplanning together to securely build the program. Throughout the development process, they will complete vulnerability scanning, testing, using trusted sources, and more. After production, they will complete transition/deployment testing and monitor logs, alerts, analytics, etc.. If there were issues or successful testing/real attacks, they will block the attacker from accessing further and reevaluate. Finally, the developers will stabilize and maintain the system.

### 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 | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | 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 | Likely | Low | P9 | L2 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | Medium | Probable | Medium | P8 | 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 for data that is stored on a disk. This data will then be translated into another form, preventing unauthorized users the ability to decrypt. |
| Encryption at flight | Encryption at flight is when the data being transmitted is encrypted. Without this information being encrypted and the endpoints being authenticated, it faces vulnerabilities and possible data exposure. |
| Encryption in use | Encryption in use is that any possible sensitive data being secure and encrypted at all times. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is testing to make sure you are who you said you are, usually with user logins. |
| Authorization | Authorization will establish if you have the qualifications (level of access) to access, change, or delete something. |
| Accounting | Accounting will document all actions performed by a user. This will allow those with the correct authorization to view user information, statistics, see the users requests, and more. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 2/13/2022 | Completed Template | Madeline Neel |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**References**

Schiela, Robert. (2020, May 29). *SEI CERT C++ Coding Standard*. Retrieved from https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Vijayan, Jaikumar. (n.d.). *6 DevSecOps best practices: Automate early and often.* Retrieved from https://techbeacon.com/security/6-devsecops-best-practices-automate-early-often