**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.z7waio2t3p3l)

[Purpose 2](#_heading=h.1wulzmsp4zs)

[Scope 2](#_heading=h.kuar2k2xttcw)

[Module Three Milestone 2](#_heading=h.q7i6nfxkn3h2)

[Ten Core Security Principles 2](#_heading=h.sya1y9rkhdbz)

[C/C++ Ten Coding Standards 3](#_heading=h.fdig4vvkse6h)

[Coding Standard 1 4](#_heading=h.nl4vglgplu9l)

[Coding Standard 2 5](#_heading=h.dclub0q4sjdi)

[Coding Standard 3 6](#_heading=h.2m6562dijdak)

[Coding Standard 4 7](#_heading=h.2o39n6ahggud)

[Coding Standard 5 8](#_heading=h.j7tnnkbvzil)

[Coding Standard 6 9](#_heading=h.ba7uk91zpzqv)

[Coding Standard 7 10](#_heading=h.3dsd4tt8ca4k)

[Coding Standard 8 11](#_heading=h.l5hdykqn1n89)

[Coding Standard 9 13](#_heading=h.arerbegc4dmp)

[Coding Standard 10 14](#_heading=h.7nux1224433d)

[Defense-in-Depth Illustration 15](#_heading=h.o9yl6fieu6e9)

[Project One 15](#_heading=h.7it12y7c1z0f)

[1.](#_heading=h.59qozckc6s67) Revise the C/C++ Standards 15

[2.](#_heading=h.22mnlfcws859) Risk Assessment 15

[3.](#_heading=h.uj7u28cv3919) Automated Detection 15

[4.](#_heading=h.3qku868mt71g) Automation 15

[5.](#_heading=h.7am7ddi782iy) Summary of Risk Assessments 16

[6.](#_heading=h.l3bazo3k6p6b) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.f4e0rzna2np4) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3fa2zpxc5cz6)

[Enforcement 18](#_heading=h.s9ho861oid0p)

[Exceptions Process 18](#_heading=h.8215fz7w8wal)

[Distribution 19](#_heading=h.5uvtweo8mbtu)

[Policy Change Control 19](#_heading=h.i7xb0z1mj3uy)

[Policy Version History 19](#_heading=h.355j5jrbrsnt)

[Appendix A Lookups 19](#_heading=h.dv3p9hyl60p2)

[Approved C/C++ Language Acronyms 19](#_heading=h.qhj9ztvw6375)

## 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 | Validating input data ensures that only properly formed data is entering the system, in effect preventing any malicious inputs from causing the system to malfunction. Most inputs that are externally sourced should be considered untrusted and therefore subject to input validation, this includes command line arguments, network interfaces, environmental variables, and user controlled files. |
| 1. Heed Compiler Warnings | Compiler warnings are put in place to help detect and prevent any vulnerabilities that are present in the code. These warnings can be rectified by modifying the code to run correctly. SAST and DAST tools are great resources to use to detect these security flaws. |
| 1. Architect and Design for Security Policies | Security should be embedded in the system from the very beginning of the design process. For example, instead of having the systems handle all tasks, it would be better to divide the system into subsystems. |
| 1. Keep It Simple | Keeping the design as simple as possible helps limit the likelihood of errors during the implementation, configuration, and use of the system. Also, the more complex a system is the harder it is to assure an appropriate level of security is in place. |
| 1. Default Deny | Base access on permissions rather than exclusion, this means that by default access should be denied and only allows access based on conditions that are defined in the protection scheme. |
| 1. Adhere to the Principle of Least Privilege | Every process should only be done with the least privilege possible in order to complete the task. Any time privileged tasks need to be completed, it should be done so for the least amount of time necessary to do said task. This limits the opportunities attackers can have to execute malicious code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | All data must be sanitized when being passed to other subsystems. This includes command shells, relational databases, and commercial off-the-shelf (COTS) components. It’s possible for attackers to exploit any unused functionality in these components. sanitizing the data beforehand can help reduce the risks of security breaches. |
| 1. Practice Defense in Depth | By implementing multiple layers of security, if one layer fails or is circumvented, another layer is there to prevent or limit any exploit being successful. An example of the security layers in the development environment can be implementing secure coding practices as well as run-time security. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques such as fuzz testing, penetrations, testing, and source code audits aid in identifying and fixing security vulnerabilities. external reviewers provided an independent perspective, which allow for an unbiased review of the system and correct any possible vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Coding standards are specific requirements on how security policies will be implemented. These standards vary depending on the development language as well as the platform the system is being built on. |

### 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** | **DCL60-CPP. Obey the one-definition rule** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | This rule states that every non-inline function or variable that is used, must have only one definition across the entire program. This is the case even when a program is split into multiple source files. This requirement ensures that when a program is compiled independently and later linked together, the linker only has one unambiguous definition for the function or variable. It’s very common in C++ development to split code into multiple files for better organization and maintainability. So, when different source files have different definitions for the same thing, this results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two classes have the same name with differing definitions. even though they are functionally the same, they are not defined using the same sequence of tokens resulting in undefined behavior. |
| // a.cpp  struct S {  int a;  };    // b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| The correct solution depends on the programmer's intent. If the programmer intends for the definitions to be public in both source files, then it’s best to use a header file to introduce the object into both source files. |
| // S.h  struct S {  int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** Heed compiler warnings: Warnings like DCL60-CPP in the IDE or Static Analysis tool should never be ignored. Eliminate the warning by modifying the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Type-compatibility  definition-duplicate undefined-extern undefined-extern-pure-virtual external-file-spreading  type-file-spreading | partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL60 |  |
| CodeSonar | 9.0p0 | LAND.STRUCT.DEF.FDH  LAND.STRUCT.DEF.ODH | Function defined in header file Object defined in header file |
| Helix QAC | 2025.1 | C++1067, C++1509, C++1510 | [Insert text.] |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully implemented |
| Parasoft  C/C++test | 2024.2 | CERT\_CPP-DCL60-a | The One Definition RUle shall not be violated |
| Polyspace Bug Finder | R2024b | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered) |
| RuleChecker | 22.10 | type-compatibility  definition-duplicate  undefined-extern  undefined-extern-pure-virtual  external-file-spreading  type-file-spreading | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **EXP53-CPP. Do not read uninitialized memory** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | if local variables are not initialized, they assume unexpected values. If an indeterminate value is produced by an evaluation, the behavior is undefined. |

| **Noncompliant Code** |
| --- |
| In this example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this example, the variable is initialized before it is printed. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** Heed compiler warnings: Warnings like EXP53-CPP in the IDE or Static Analysis tool should never be ignored. Eliminate the warning by modifying the code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | uninitialized-read | Partially checked |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| CodeSonar | 9.0p0 | LANG.STRUCT.RPL LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| Helix QAC | 2025.1 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978 |  |
| Klocwork | 2025.1 | UNINIT.CTOR.MIGHT  UNINIT.CTOR.MUST  UNINIT.HEAP.MIGHT  UNINIT.HEAP.MUST  UNINIT.STACK.ARRAY.MIGHT  UNINIT.STACK.ARRAY.MUST  UNINIT.STACK.ARRAY.PARTIAL.MUST  UNINIT.STACK.MIGHT  UNINIT.STACK.MUST |  |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024b | CERT C++: EXP53-CPP | Checks for:   * Non-initialized variable * Non-initialized pointer   Rule partially covered. |
| PVS-Studio | 7.36 | V546, V573, V614, V670, V679, [V7](https://pvs-studio.com/en/docs/warnings/v730/)3[0](https://pvs-studio.com/en/docs/warnings/v730/), V788, V1007, V1050 |  |
| RuleChecker | 22.10 | uninitialized-read | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **STR-30-C. Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | If a program attempts to modify a string literal, this results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The char pointer str is being assigned to the address of a string literal, which in this example is “string literal”.  The second line of code is trying to assign the capital letter “S” to the string, replacing whatever is at index 0. This results in undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Instead of using a pointer, the compliant code utilizes an array initializer, which specifies both the specific values of characters in the array and the size of the array as well. 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):** Heed compiler warnings: Warnings like STR-30-C in the IDE or Static Analysis tool should never be ignored. Eliminate the warning by modifying the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | string-literal-modification  write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Compass/ROSE |  |  | Can detect simple violations of this rule |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| Helix QAC | 2024.4 | C0556, C0752, C0753, C0754  C++3063, C++3064, C++3605, C++3606, C++3607 |  |
| Klocwork | 2024.4 | CERT.STR.ARG.CONST\_TO\_NONCONST CERT.STR.ASSIGN.CONST\_TO\_NONCONST |  |
| LDRA tool suite | 9.7.1 | 157 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR30-a CERT\_C-STR30-b | A string literal shall not be modified Do not modify string literals |
| PC-lint Plus | 1.4 | 489, 1776 | Partially supported |
| Polyspace Bug Finder | R2024b | CERT C: Rule STR30-C | Checks for writing to const qualified object (rule fully covered) |
| PVS-Studio | 7.36 | V675 |  |
| RuleChecker | 24.04 | string-literal-modification | Partially checked |
| Splint | 3.1.1 |  |  |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified (see one compliant and one non-compliant example). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **STR02-C. Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to subsystems. |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. Below is a code example that takes inputs of an email address to a buffer and then uses the string as an argument to call to the system. The issue is that a user can input strings that can exploit vulnerabilities. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| Below is a whitelisting approach to data sanitization. This defines a list of accepted characters and removes any characters that are not accepted. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |

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

| **Principles(s):** Validate input: Any input made by the user should be sanitized and validated to prevent any unexpected/unwanted behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 9.0p0 | IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Klocwork | 2024.4 | NNTS.TAINTED SV.TAINTED.INJECTION |  |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR02-a CERT\_C-STR02-b CERT\_C-STR02-c | Protect against command injection Protect against file name injection Protect against SQL injection |
| Polyspace Bug Finder | R2024b | CERT C: Rec. STR02-C | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **MEM50-CPP. Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | 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. |

| **Noncompliant Code** |
| --- |
| S has been dereferenced after it has been deallocated. This vulnerability can be exploited to run 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 used. As can be seen below, s is allocated then the function f() is being called via the pointer s. Finally, s is deleted. |
| #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):** Use effective quality assurance techniques: Performing numerous types of testing and independent security reviews can help identify these flaws. |
| --- |

**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 |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| 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 | 9.0p0 | ALLOC.UAF | Use after free |
| Compass/ROSE |  |  |  |
| 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 |
| Helix QAC | 2025.1 | C++4303, C++4304 |  |
| Klocwork | 2025.1 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST |  |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024b | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| PVS-Studio | 7.36 | V586, V774 |  |
| Splint | 5.0 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **DCL03-C. Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. static\_assert allows incorrect assumptions to be diagnosed at compile time rather than runtime therefore there is no overhead. |

| **Noncompliant Code** |
| --- |
| This code uses assert(), which needs to be placed in a function and executed, meaning that it’s not in the same structure that it refers to. The diagnosis only occurs at runtime and only if the function 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 code uses static\_assert(), which allows for the assumptions to be diagnosed at compile time instead of runtime. If it does fail, it results in a meaningful 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):** Use effective quality assurance techniques: Performing numerous types of testing and independent security reviews can help identify these flaws.  Adopt a secure coding standard: In doing so, this allows us to know when to use static\_assert. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | P3 | 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 | 9.0p0 | (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** | **ERR51-CPP. Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching exception handler. Using the matching exception handler ensures that the stack is properly unwound and allows for the opportunity to manage the external resources before terminating the process. This adds another layer of defense as it prevents internal information from being exposed that might aid an attacker. |

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions that are thrown by throwing\_func(). Because of this, std::terminate() is called which also calls std::abort(), which abnormally terminates the process. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Architect and Design for Security Policies: Following this would dictate that if an expectation is thrown, It should be caught.  Use effective quality assurance techniques: Through thorough testing, any uncaught exceptions can be found, and protected against.  Defense in depth: adds an additional layer of security by gracefully catching errors so the application does not crash and reveal internal information. |
| --- |

**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 |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 9.0p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2025.1 | C++4035, C++4036, C++4037 |  |
| Klocwork | 2025.1 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-ERR51-a CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2024b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **INT32-C. Ensure that operations on signed integers do not result in overflow** |
| --- | --- | --- |
| Integers | [STD-008-CPP] | Signed integer overflow results in undefined behavior and are particularly important to safeguard against when operations on signed integer values are performed on untrusted data like user input. It’s important to validate user input to prevent this from happening. |

| **Noncompliant Code** |
| --- |
| This code example can result in a signed integer overflow during the addition of si\_a and si\_b. |
| void func(signed int si\_a, signed int si\_b) {  signed int sum = si\_a + si\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant code ensures that the addition operation cannot result in a signed integer overflow. |
| #include <limits.h>    void f(signed int si\_a, signed int si\_b) {  signed int sum;  if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||  ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {  /\* Handle error \*/  } else {  sum = si\_a + si\_b;  }  /\* ... \*/  } |

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

| **Principles(s):** Validate input: Without doing this, it is very easy to trigger a signed integer overflow, which can allow for arbitrary input from the user. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | integer-overflow | Fully checked |
| CodeSonar | 9.0p0 | 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 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Cppcheck Premium | 24.11.0 | premium-cert-int32-c |  |
| Helix QAC | 2024.4 | C2800, C2860  C++2800, C++2860  DF2801, DF2802, DF2803, DF2861, DF2862, DF2863 |  |
| Klocwork | 2024.4 | NUM.OVERFLOW CWARN.NOEFFECT.OUTOFRANGE NUM.OVERFLOW.DF |  |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT32-a CERT\_C-INT32-b CERT\_C-INT32-c | Avoid signed integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| Parasoft Insure++ |  |  | Runtime analysis |
| Polyspace Bug Finder | R2024b | CERT C: Rule INT32-C | Checks for:   * Integer overflow * Tainted division operand * Tainted modulo operand   Rule partially covered. |
| PVS-Studio | 7.36 | V1026, V1070, V1081, V1083, V1085, V5010 |  |
| TrustInSoft Analyzer | 1.38 | signed\_overflow | Exhaustively verified (see one compliant and one non-compliant example). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **DLC58-CPP. Do not modify the standard namespaces** |
| --- | --- | --- |
| Declarations and Initializations] | [STD-009-CPP] | Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions. One feature of namespaces is that they can be further extended, even within separate source files. However, it is undefined behavior to add new declarations to the std namespace except under special circumstances. |

| **Noncompliant Code** |
| --- |
| This noncompliant code has the declaration of x, added to the std namespace, which results in undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| This compliant solution declares x into a namespace without a reserved name. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):** Architect and design for security policies: The policies should not allow for this to happen.  Keep it simple: modifying standard namespaces instead of creating a new one is the opposite of keeping it simple. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL58 |  |
| CodeSonar | 9.0p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Helix QAC | 2025.1 | C++3180, C++3181, C++3182 |  |
| Klocwork | 2025.1 | CERT.DCL.STD\_NS\_MODIFIED |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces 'std' and 'posix' |
| Polyspace Bug Finder | R2024b | CERT C++: DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |
| PVS-Studio | 7.36 | V1061 |  |
| SonarQube C/C++ Plugin | 4.10 | S3470 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **OOP53-CPP. Write constructor member initializers in the canonical order** |
| --- | --- | --- |
| Object oriented programming | [STD-010-CPP] | Writing member initializers other than in canonical order can result in undefined behavior, such as reading uninitialized memory. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code, someVal is being initialized first and then dependOnSomeVal to a value that is dependent on someVal. Because the declaration order of the member variables does not match the member initializer order, attempting to read the value of someVal results in an unspecified value being stored into dependsOnSomeVal. |
| class C {  int dependsOnSomeVal;  int someVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| This compliant code changes the ordering of the declaration for the class member variables. |
| class C {  int someVal;  int dependsOnSomeVal;    public:  C(int val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** Keep it simple: This keeps the code clear on what it is doing for anyone to look at and understand. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | initializer-list-order | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP53 |  |
| Clang | 3.9 | -Wreorder |  |
| CodeSonar | 9.0p0 | LANG.STRUCT.INIT.OOMI | Out of Order Member Initializers |
| Helix QAC | 2025.1 | C++4053 |  |
| Klocwork | 2025.1 | CERT.OOP.CTOR.INIT\_ORDER |  |
| LDRA tool suite | 9.7.1 | 206 S | Fully implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-OOP53-a | List members in an initialization list in the order in which they are declared |
| Polyspace Bug Finder | R2024b | CERT C++: OOP53-CPP | Checks for members not initialized in canonical order (rule fully covered) |
| RuleChecker | 22.10 | initializer-list-order | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S3229 |  |

### 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.

Using automation for enforcing and maintaining secure coding standards helps in creating the most secure application possible. As can be seen from the DevSecOps diagram above, security is embedded throughout each phase of the pipeline. Based on the security policies and standards that are defined in this document, they should be implemented into the DevSecOps process with the help of automation. For instance, the Build and Verify and Test phases are the first two phases where automation can be utilized. During the continuous integration process, the code commit can be subject to unit testing, scanning dependencies, static application security testing (SAST), and penetration testing to help find vulnerabilities as early as possible.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | High | P3 | L3 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | Low | Likely | Low | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P9 | L2 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Likely | Medium | P18 | L1 |
| STD-009-CPP | High | Unlikely | Medium | P6 | L2 |
| 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 at rest | Encryption at rest is when stored data, like on a smartphone or hard drive, is in an encrypted state by using encryption algorithms. So even if the data is obtained through malicious means, it’s still unreadable unless the encryption key is also known for it to then be decrypted. This policy is important because we need to protect the user data that we collect and store. Without this policy, we would be an incredibly easy target for malicious attacks. |
| Encryption in flight | Encryption in flight refers to data being in an encrypted state as it is in transit from source to destination, typically over a network. While data is in transit, it must be encrypted otherwise if someone intercepts that data while it is in transit, they are able to gain access to the data in its natural encrypted format. |
| Encryption in use | Encryption in use is the encryption of data while the system is using it (i.e. processing, updating, reading, etc.) This policy aims to fill in the gap as when data is in flight and at rest is now encrypted, data while it is in use becomes the weak link and will naturally be targeted. Therefore it is important to mitigate any possible data exposure by encrypting data while it’s in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming a user is who they say they are. It’s essentially asking the user “Who are You?” in which the user will provide some credentials that the application already has stored in order to authenticate whether the provided credentials matches what the application has in storage. If a user is new then we need to ensure the data given to us is properly stored and ready to use for future authentication. This policy applies whenever a user is trying to sign in to an application and we need to verify the credentials given. |
| Authorization | Authorization is simply the set of attributes a user is authorized to perform. Certain users have a certain level of access to the network, system, or application. This policy applies whenever a user is trying to perform an action within the system. This helps prevent regular users from doing actions or accessing parts of a system they shouldn’t be allowed to do or access like accessing sensitive files or adding changes to the database through commands. |
| Accounting | Accounting is the recording and monitoring of users actions on the network and or application who have been authenticated and authorized. This policy is important to know what is going on within the system. For example, if someone notices unusual behavior within the system, an audit can be done in order to see anything that may have led to that unusual behavior. |

**\***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 | 5/25/2025 | Module 3 | Levi Bajuscik | Mimi Tam |
| 3.0 | 6/15/2025 | Module 6 | Levi Bajuscik | Mimi Tam |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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