**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 | Ensure that all input data is validated to prevent malicious input, such as SQL injection or intentional buffer overflow. This helps protect systems from unexpected behavior or security breaches caused by poorly validated data. |
| 1. Heed Compiler Warnings | Pay attention to compiler warnings and resolve them as they can highlight vulnerabilities or coding mistakes. Ignoring them may lead to security issues or instability in the program. |
| 1. Architect and Design for Security Policies | Design systems with security in mind from the start by incorporating security policies into the architecture. This approach ensures that vulnerabilities are minimized during the development phase. Catching the bugs early can be a huge time save for the project as a whole. |
| 1. Keep It Simple | Avoid unnecessary complexity in program code and system design. Simple systems are easier to develop, debug, and maintain, leading to less hidden vulnerabilities. |
| 1. Default Deny | Set all permission systems in the program to deny access by default. In doing this, each user will have restricted access unless explicitly granted access. |
| 1. Adhere to the Principle of Least Privilege | Grant users and systems the minimum access necessary to perform their tasks. This limits the potential damage from accidental misuse or malicious activities. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data before sending it to other systems to prevent injection attacks and ensure compatibility. Proper sanitization protects against exploiting vulnerabilities in the target system. |
| 1. Practice Defense in Depth | Implement multiple layers of security so that even if one layer fails, others will still protect the system. There is a balance that can be struck with this practice. Implement enough defenses to ensure safety without hurting user experience or program performance. |
| 1. Use Effective Quality Assurance Techniques | Incorporate multiple layers of testing, like code reviews, static analysis, and penetration testing, to help identify and fix vulnerabilities early in the development phase. This principle helps remove bugs and exploits before the program is released to production. |
| 1. Adopt a Secure Coding Standard | Follow secure coding standards to ensure consistency and reduce the risk of vulnerabilities being present in the program. This helps contribute to the building of secure and bug-free software. |

### 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] | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.” (Pincar, 2023d) |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the object is initialized prior to printing its value.” (Pincar, 2023d) |
| #include <iostream>    void f() {    int i = 0;    std::cout << i;  } |

| **Principles(s):**   1. **Heed Compiler Warnings**: Compilers will often give warnings about using uninitialized memory 2. **Keep It Simple**: Less likely to lose track of uninitialized variables with simpler code 3. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch uninitialized memory issues early 4. **Adopt a Secure Coding Standard**: Secure coding standards state that variables should always be initialized |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 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 | 8.3p0 | LANG.STRUCT.RPL, LANG.MEM.UVAR | Return pointer to local, Uninitialized variable |
| Helix QAC | 2024.4 | DF726, DF2727, DF2728, DF2961, DF2962, DF2963, DF2966, DF2967, DF2968, DF2971, DF2972, DF2973, DF2976, DF2977, DF978 | - |
| Klocwork | 2024.4 | 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 | R2024a | CERT C++: EXP53-CPP | Checks for: Non-initialized variable, Non-initialized pointer. Rule partially covered. |
| PVS-Studio | 7.35 | V546, V573, V614, V670, V679, V730, V788, V1007, V1050 | - |
| RuleChecker | 22.10 | uninitialized-read | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior.” (Saks, 2023) |
| #include <cstdarg>    int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior.” (Saks, 2023) |
| #include <type\_traits>    template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }    template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {  return f + add(rest...);  } |

| **Principles(s):**   1. **Keep It Simple**: Variadic functions increase complexity which can lead to hard-to-find errors 2. **Adopt a Secure Coding Standard**: Secure coding standards often discourage C-style variadic functions |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | function-ellipsis | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL50 | - |
| Clang | 3.9 | cert-dcl50-cpp | Checked by clang-tidy. |
| CodeSonar | 8.3p0 | LANG.STRUCT.ELLIPSIS | Ellipsis |
| Helix QAC | 2024.4 | C++2012, C++2625 | - |
| Klocwork | 2024.4 | MISRA.FUNC.VARARG | - |
| LDRA tool suite | 9.7.1 | 41 S | Fully Implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL50-a | Functions shall not be defined with a variable number of arguments |
| Polyspace Bug Finder | R2024a | CERT C++: DCL50-CPP | Checks for function definition with ellipsis notation (rule fully covered) |
| RuleChecker | 22.10 | function-ellipsis | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | FunctionEllipsis | - |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| “Because the input is unbounded, the following code could lead to a buffer overflow.” (Pincar, 2023a) |
| #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.” (Pincar, 2023a) |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

| **Principles(s):**   1. **Validate User Input**: Ensure proper string storage to prevent buffer overflow 2. **Heed Compiler Warnings**: Compilers will often give warnings about potential buffer overflows 3. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch string storage issues early 4. **Adopt a Secure Coding Standard**: Secure coding standards state that strings should be properly sized and terminated |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 8.3p0 | MISC.MEM.NTERM, LANG.MEM.BO, LANG.MEM.TO | No space for null terminator, Buffer overrun, Type overrun |
| Helix QAC | 2024.4 | C++5216, DF2835, DF2836, DF2839 | - |
| Klocwork | 2024.4 | 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 | 2024.2 | 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 | R2024a | CERT C++: STR50-CPP | Checks for: Use of dangerous standard function, Missing null in string array, Buffer overflow from incorrect string format specifier, Destination buffer overflow in string manipulation, Insufficient destination buffer size. Rule partially covered. |
| RuleChecker | 22.10 | stream-input-char-array | Partially checked |
| SonarQube C/C++ Plugin | 4.10 | S3519 | - |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-JAV] | Normalize strings before validating them |

| **Noncompliant Code** |
| --- |
| “The Normalizer.normalize() method transforms Unicode text into the standard normalization forms described in Unicode Standard Annex #15 Unicode Normalization Forms. Frequently, the most suitable normalization form for performing input validation on arbitrarily encoded strings is KC (NFKC) .  This noncompliant code example attempts to validate the String before performing normalization.” (Mohindra, 2024) |
| // String s may be user controllable  // \uFE64 is normalized to < and \uFE65 is normalized to > using the NFKC normalization form  String s = "\uFE64" + "script" + "\uFE65";    // Validate  Pattern pattern = Pattern.compile("[<>]"); // Check for angle brackets  Matcher matcher = pattern.matcher(s);  if (matcher.find()) {  // Found black listed tag  throw new IllegalStateException();  } else {  // ...  }    // Normalize  s = Normalizer.normalize(s, Form.NFKC); |

| **Compliant Code** |
| --- |
| “This compliant solution normalizes the string before validating it. Alternative representations of the string are normalized to the canonical angle brackets. Consequently, input validation correctly detects the malicious input and throws an IllegalStateException.” (Mohindra, 2024) |
| String s = "\uFE64" + "script" + "\uFE65";    // Normalize  s = Normalizer.normalize(s, Form.NFKC);    // Validate  Pattern pattern = Pattern.compile("[<>]");  Matcher matcher = pattern.matcher(s);  if (matcher.find()) {  // Found blacklisted tag  throw new IllegalStateException();  } else {  // ...  } |

| **Principles(s):**   1. **Validate User Input**: Normalizing strings helps ensure that validation is accurate and prevents attacks like SQLi 2. **Sanitize Data Sent to Other Systems**: Normalizing strings is important to sanitize data before it is sent to a database 3. **Adopt a Secure Coding Standard**: Normalizing input is a common secure coding practice and is part of many secure coding standards |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors (see Chapter 8) |
| CodeSonar | 8.1p0 | JAVA.IO.INJ.SQL | SQL injection |
| Coverity | 7.5 | SQLI, FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_, FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Fortify | 1.0 | HTTP\_Response\_Splitting, SQL\_Injection\_\_Persistence, SQL\_Injection | Implemented |
| Klocwork | 2024.4 | SV.DATA.DB, SV.SQL, SV.SQL.DBSOURCE | Implemented |
| Parasoft Jtest | 2024.2 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SonarQube | 9.9 | S2077, S3649 | Executing SQL queries is security-sensitive, SQL queries should not be vulnerable to injection attacks |
| SpotBugs | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE, SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |

#### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems.” (Pincar, 2023f) |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required.” (Pincar, 2023f) |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Principles(s):**   1. **Heed Compiler Warnings**: Compilers will often give warnings about use-after-free or dangling pointer issues 2. **Keep It Simple**: Keeping a program simpler will lead to less pointers and memory issues 3. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch use-after-free or dangling pointer issues early 4. **Adopt a Secure Coding Standard**: Secure coding standards emphasize the importance of secure memory management |
| --- |

**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 | 8.3p0 | 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 |
| Helix QAC | 2024.4 | C++4303, C++4304 | - |
| Klocwork | 2024.4 | 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 | R2024a | 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 | 7.35 | V586, V774 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Incorporate diagnostic tests using assertions |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack.” (Seacord, 2018) |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char \*)malloc(len + 1);  assert(NULL != dup);    memcpy(dup, c\_str, len + 1);  return dup;  } |

| **Compliant Code** |
| --- |
| “This compliant solution demonstrates how to detect and handle possible memory exhaustion:” (Seacord, 2018) |
| char \*dupstring(const char \*c\_str) {  size\_t len;  char \*dup;    len = strlen(c\_str);  dup = (char\*)malloc(len + 1);  /\* Detect and handle memory allocation error \*/  if (NULL == dup) {  return NULL;  }    memcpy(dup, c\_str, len + 1);  return dup;  } |

| **Principles(s):**   1. **Keep It Simple**: Assertions help simplify the debugging process 2. **Use Effective Quality Assurance Technique**: Assertions qualify as an effective quality assurance technique 3. **Adopt a Secure Coding Standard**: Secure coding standards often recommend using assertions to confirm program correctness and prevent undefined behavior |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.3p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called.” (Ballman, 2023) |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources.” (Ballman, 2023) |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

| **Principles(s):**   1. **Keep It Simple**: Handling exceptions appropriately helps prevent complex errors 2. **Practice Defense in Depth**: Exception handling adds a layer of defense for the Defense in Depth practice 3. **Use Effective Quality Assurance Technique**: Exception handling is often tested to ensure edge cases are acknowledged and managed. 4. **Adopt a Secure Coding Standard**: Secure coding standards often recommend proper exception handling |
| --- |

**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 | 8.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2024.4 | C++4035, C++4036, C++4037 | - |
| Klocwork | 2024.4 | 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 | R2024a | 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** | **Name of Standard** |
| --- | --- | --- |
| Container Usage | [STD-008-CPP] | Guarantee that container indices and iterators are within the valid range |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table.” (Pincar, 2023b) |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments.” (Pincar, 2023b) |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

| **Principles(s):**   1. **Validate Input Data**: Checking that indices and iterators are within range is a form of validating input to prevent out-of-bounds access 2. **Heed Compiler Warnings**: Compilers often give warnings for potential out-of-bounds access 3. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch invalid range accesses early 4. **Adopt a Secure Coding Standard**: Secure coding standards emphasize the importance of bounds-checking practices |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference | - |
| CodeSonar | 8.3p0 | LANG.MEM.BO, LANG.MEM.BU, LANG.MEM.TO, LANG.MEM.TU, LANG.MEM.TBA, LANG.STRUCT.PBB, LANG.STRUCT.PPE, LANG.STRUCT.PARITH | Buffer overrun, Buffer underrun, Type overrun, Type underrun, Tainted buffer access, Pointer before beginning of object, Pointer past end of object, Pointer Arithmetic |
| Helix QAC | 2024.4 | C++3139, C++3140, DF2891 | - |
| Klocwork | 2024.4 | ABV.ANY\_SIZE\_ARRAY, ABV.GENERAL, ABV.GENERAL.MULTIDIMENSION, ABV.STACK, ABV.TAINTED, SV.TAINTED.ALLOC\_SIZE, SV.TAINTED.CALL.INDEX\_ACCESS, SV.TAINTED.CALL.LOOP\_BOUND, SV.TAINTED.INDEX\_ACCESS | - |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2024a | CERT C++: CTR50-CPP | Checks for: Array access out of bounds, Array access with tainted index, Pointer dereference with tainted offset. Rule partially covered. |
| PVS-Studio | 7.35 | V781 | - |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expression Evaluation | [STD-009-CPP] | Do not depend on the order of evaluation for side effects |

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, i is evaluated more than once in an unsequenced manner, so the behavior of the expression is undefined.” (Pincar, 2023c) |
| void f(int i, const int \*b) {  int a = i + b[++i];  // ...  } |

| **Compliant Code** |
| --- |
| “This example is independent of the order of evaluation of the operands and can each be interpreted in only one way.” (Pincar, 2023c) |
| void f(int i, const int \*b) {  ++i;  int a = i + b[i];  // ...  } |

| **Principles(s):**   1. **Heed Compiler Warnings**: Compilers may warn about ambiguous evaluation 2. **Keep It Simple**: Avoiding side effects in evaluation order simplifies the code 3. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch evaluation order bugs 4. **Adopt a Secure Coding Standard**: Secure coding standards often discourage reliance on evaluation order and side effects |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-EXP50 |  |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required |
| CodeSonar | 8.3p0 | LANG.STRUCT.SE.DEC, LANG.STRUCT.SE.INC | Side Effects in Expression with Decrement, Side Effects in Expression with Increment |
| Compass/ROSE |  |  | Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator |
| Coverity | v7.5.0 | EVALUATION\_ORDER | Can detect the specific instance where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently |
| ECLAIR | 1.2 | CC2.EXP30 | Fully implemented |
| GCC | 4.9 |  | Can detect violations of this rule when the -Wsequence-point flag is used |
| Helix QAC | 2024.4 | C++3220, C++3221, C++3222, C++3223, C++3228 | - |
| Klocwork | 2024.4 | PORTING.VAR.EFFECTS, CERT.EXPR.PARENS, MISRA.EXPR.PARENS.INSUFFICIENT, MISRA.INCR\_DECR.OTHER | - |
| LDRA tool suite | 9.7.1 | 35 D, 1 Q, 9 S, 134 S, 67 D, 72 D | Partially implemented |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-EXP50-a, CERT\_CPP-EXP50-b, CERT\_CPP-EXP50-c, CERT\_CPP-EXP50-d, CERT\_CPP-EXP50-e, CERT\_CPP-EXP50-f | The value of an expression shall be the same under any order of evaluation that the standard permits. Don't write code that depends on the order of evaluation of function arguments. Don't write code that depends on the order of evaluation of function designator and function arguments. Don't write code that depends on the order of evaluation of expression that involves a function call. Between sequence points an object shall have its stored value modified at most once by the evaluation of an expression. Don't write code that depends on the order of evaluation of function calls |
| Polyspace Bug Finder | R2024a | CERT C++: EXP50-CPP | Checks for situations where expression value depends on order of evaluation (rule fully covered). |
| PVS-Studio | 7.35 | V521, V708 | - |
| SonarQube C/C++ Plugin | 4.10 | IncAndDecMixedWithOtherOperators | Partially implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Iinput/Output | [STD-010-CPP] | Do not alternately input and output from a file stream without an intervening positioning call |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is undefined.” (Pincar, 2023e) |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data";  std::string str;  file >> str;  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the std::basic\_istream<T>::seekg() function is called between the output and input, eliminating the undefined behavior.” (Pincar, 2023e) |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    file << "Output some data";    std::string str;  file.seekg(0, std::ios::beg);  file >> str;  } |

| **Principles(s):**   1. **Heed Compiler Warnings**: Compilers may warn about issues with file handling and stream positioning 2. **Use Effective Quality Assurance Technique**: Good testing, like static analysis tools or runtime checks, can catch improper file stream usage 3. **Adopt a Secure Coding Standard**: Secure coding standards often specify proper file handling practices |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-FIO50 | - |
| CodeSonar | 8.3p0 | IO.IOWOP, IO.OIWOP | Input After Output Without Positioning, Output After Input Without Positioning |
| Helix QAC | 2024.4 | DF4711, DF4712, DF4713 |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-FIO50-a | Do not alternately input and output from a stream without an intervening flush or positioning call |
| Polyspace Bug Finder | R2024a | CERT C++: FIO50-CPP | Checks for alternating input and output from a stream without flush or positioning call (rule fully covered) |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



Automation is important for integrating security into every phase of the software development lifecycle. Tools such as IriusRisk for threat modeling, OWASP dependency-check for vulnerability scanning, and others like JBroFuzz and OWASP ZAP during testing help us maintain DevSecOps security standards.

By automating these processes, we reduce the possibility of human error and ensure continuous monitoring and rapid response to security threats. This proactive approach speeds up the development cycle and enhances overall program security by consistently validating and improving our defenses.

In the planning phase, a tool like IriusRisk can be used to help collaborate with teammates. IriusRisk helps a team collaboratively build threat modeling for a project.

Next, in the build phase, code is consistently scanned for any potential vulnerabilities and exploits with a tool like OWASP dependency-check. This tool will help automate the vulnerability scanning process by looking for utilized libraries with public vulnerabilities.

During the testing phase, a build artifact is created (a functioning early version of the program). It is tested using tools like JBroFuzz, OWASP ZAP, and Arachi which help detect issues with user authentication, authorization, SQL injection, and API-related endpoints.

Lastly, once the program is deployed, tools like Osquery, Falco, and Tripwire will help determine whether an application is functioning as intended while live in a production environment. One way they do this is by purposely messing up one part of the program to make sure other parts remain functional. This is an important aspect of a program to test because programs can face many types of unexpected issues once deployed to a production environment.

### 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 | Medium | **P12** | **L1** |
| STD-002-CPP | High | Probable | Medium | **P12** | **L1** |
| STD-003-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-004-JAV | High | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-006-C | Low | Unlikely | High | **P1** | **L3** |
| STD-007-CPP | Low | Probable | Medium | **P4** | **L3** |
| STD-008-CPP | High | Likely | High | **P9** | **L2** |
| STD-009-CPP | Medium | Probable | Medium | **P8** | **L2** |
| STD-010-CPP | Low | Likely | Medium | **P6** | **L2** |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the encryption of information stored on physical media like hard drives, databases, or cloud storage (imperva, n.d.). To adhere to this security principle, all data on company computers will be encrypted when saved to the computer. This is something that will be done automatically by the operating system and will not require manual intervention from the user.  Encrypting data saved on computer hard drives helps prevent unauthorized access into the computer system’s files, increasing overall security. |
| Encryption in flight | Encryption in flight refers to the encryption of information that is moving from one location to another on a local network or through the internet (imperva, n.d.). To adhere to this security, all data moving throughout the local network and over the internet will need to be encrypted using TLS 1.2 or higher. This is something that will be done automatically by the web browser with preconfigured settings and will not require manual intervention from the user.  Any emails containing sensitive info must also be encrypted to ensure the data is not intercepted while in transit to the recipient.  Encrypting network traffic and sensitive information ensures that users outside of the local network will not have the ability to read sensitive information being sent. |
| Encryption in use | Encryption in use refers to the active encryption of information that is being read/modified by a program or computer user. Data is typically vulnerable in this state because it must be decrypted for a computer user to read or modify (imperva, n.d.). To adhere to this security principle, sensitive documents should only be decrypted when necessary and should be immediately saved and encrypted when finished. This practice will help minimize the amount of time that a document spends unencrypted when editing or reading it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the process where a user provides information about who they are. The data presented is typically information about the user or information that only the user knows (Fortinet, 2023).  At Green Pace, all users will be required to have a strong password of at least 12 characters that contain an uppercase, lowercase, and symbol. This policy will help increase the complexity of passwords and reduce the chance of accounts being compromised. |
| Authorization | Authorization refers to the ability to grant users access to specific areas of a network or system. These permissions should have the ability to be changed at any point in time (Fortinet, 2023).  At Green Pace, a system administrator will have the ability to modify user permissions. Per company policy, only the HR manager’s account will have the ability to add new users to the system. Aside from this account, all other accounts will come with a preset list of authorized areas in the network. This policy helps prevent accidental unauthorized usage into the system and adheres to the “default deny” security principle mentioned earlier in this document. |
| Accounting | Accounting refers to the tracking of a specific user’s access across a system or network. This tracking information should have the ability to be viewed by administrators or power users on the system or network (Fortinet, 2023).  At Green Pace, all logins will create an entry in the auditing database that tracks details like the IP address, location, and computer being used to access the account. This policy will help track unauthorized access into user accounts. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 1/25/2025 | Module 3 Milestone | Christopher Bull |  |
| 2.0 | 2/15/2025 | Final submission | Christopher Bull |  |

### Approved C/C++ Language Acronyms

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

**References**

Ballman, A. (2023, April 20). *ERR51-CPP. Handle all exceptions - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

Fortinet. (2023). *What Is AAA Security?* Fortinet. https://www.fortinet.com/resources/cyberglossary/aaa-security

imperva. (n.d.). *What is Data at Rest | Security & Encryption Explained | Imperva*. Learning Center. https://www.imperva.com/learn/data-security/data-at-rest/

Mohindra, D. (2024, June 11). *IDS00-J. Prevent SQL injection - SEI CERT Oracle Coding Standard for Java - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection

Pincar, J. (2023a, January 19). *STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Pincar, J. (2023b, April 20). *CTR50-CPP. Guarantee that container indices and iterators are within the valid range - SEI CERT C++ Coding Standard - Confluence*. Cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR50-CPP.+Guarantee+that+container+indices+and+iterators+are+within+the+valid+range

Pincar, J. (2023c, April 20). *EXP50-CPP. Do not depend on the order of evaluation for side effects - SEI CERT C++ Coding Standard - Confluence*. Cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP50-CPP.+Do+not+depend+on+the+order+of+evaluation+for+side+effects

Pincar, J. (2023d, April 20). *EXP53-CPP. Do not read uninitialized memory - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory

Pincar, J. (2023e, April 20). *FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call - SEI CERT C++ Coding Standard - Confluence*. Cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO50-CPP.+Do+not+alternately+input+and+output+from+a+file+stream+without+an+intervening+positioning+call

Pincar, J. (2023f, April 20). *MEM50-CPP. Do not access freed memory - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Saks, D. (2023, April 20). *DCL50-CPP. Do not define a C-style variadic function - SEI CERT C++ Coding Standard - Confluence*. Cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL50-CPP.+Do+not+define+a+C-style+variadic+function

Seacord, R. (2018, August 6). *MSC11-C. Incorporate diagnostic tests using assertions - SEI CERT C Coding Standard - Confluence*. Cmu.edu. https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions