**Joseph Veneski: Security Policy Guide**



# 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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 12](#_Toc52464063)

[Coding Standard 6 14](#_Toc52464064)

[Coding Standard 7 16](#_Toc52464065)

[Coding Standard 8 18](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 22](#_Toc52464068)

[Defense-in-Depth Illustration 24](#_Toc52464069)

[Project One 24](#_Toc52464070)

[1. Revise the C/C++ Standards 24](#_Toc52464071)

[2. Risk Assessment 24](#_Toc52464072)

[3. Automated Detection 24](#_Toc52464073)

[4. Automation 24](#_Toc52464074)

[5. Summary of Risk Assessments 25](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 25](#_Toc52464076)

[7. Map the Principles 27](#_Toc52464077)

[Audit Controls and Management 28](#_Toc52464078)

[Enforcement 28](#_Toc52464079)

[Exceptions Process 28](#_Toc52464080)

[Distribution 29](#_Toc52464081)

[Policy Change Control 29](#_Toc52464082)

[Policy Version History 29](#_Toc52464083)

[Appendix A Lookups 29](#_Toc52464084)

[Approved C/C++ Language Acronyms 29](#_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 | Any user input to a system must be thoroughly checked and sanitized before any interactions. Bad input data can lead to buffer overflows, SQL injection, or cross-site scripting (XSS) attacks. By ensuring all input to the system meets defined criteria and rejecting bad input can significantly enhance system security. |
| 1. Heed Compiler Warnings | Compiler warnings should not be ignored and can provide insight into potential vulnerabilities in code. As the cost of rectifying issues increases later in the development cycle, it is in the company’s best interest to eliminate potential vulnerabilities as early as possible. |
| 1. Architect and Design for Security Policies | Security should be an integral part of the system design and not an afterthought. Incorporating security from the outset can mitigate risks as well as avoiding needing to redesign later to meet security compliance standards. Avoid creating technical debt and follow secure coding standards from the start. |
| 1. Keep It Simple | Do not overengineer or create needlessly complex designs. Code should be designed to be efficient, functional, and readable. Use variable names that make sense and are descriptive, comment code thoroughly, ensure functions are single-purpose. |
| 1. Default Deny | The security principle of Default Deny is an approach that blocks access to all requests except explicitly listed requests. This policy can help to minimize potential vectors through which attackers can infiltrate a system and increase overall security. In conjunction with validating all input data, this can help mitigate threats significantly. |
| 1. Adhere to the Principle of Least Privilege | Only give the minimum access required to perform a task or function. If users only need read access to a database, they should not be able to accidentally overwrite or delete data. This can also prevent intruders who gain access from wreaking havoc by limiting the amount of damage they can do. |
| 1. Sanitize Data Sent to Other Systems | All data leaving the system should be validated and checked. It is important to ensure data leaving the system does not contain malicious code that can be propagated, and any information is secured. Compliance and regulation standards require that any personal data needs to be encrypted and secured such as personal identifiable information or financial information. |
| 1. Practice Defense in Depth | Defense in Depth is a multilayered strategy to enhance security. This could include implementation of authorization and authentication measures, limited access controls such as following the principle of least privilege, and a thorough security policy. Through a multi-tiered approach including security training and awareness for staff to security by design, systems can greatly improve their security stance and mitigate many potential threats. |
| 1. Use Effective Quality Assurance Techniques | Code should meet predefined quality standards and be thoroughly tested. Automated testing tools should be used for analysis and developers should participate in frequent code reviews. Ensure appropriate testing coverage of code and employ the skills of security professionals such as penetration testers when able to find potential vulnerabilities. |
| 1. Adopt a Secure Coding Standard | For designing any larger projects which require coordination of a team, adopting a secure coding standard can help bring all stakeholders within alignment. Defined standards for variable definition and use, data sanitization, and code testing coverage can increase security as well as helping to ensure everything meets compliance and regulation standards. This can also make code easier to maintain and review in the future. |

### 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** | [DAT-001-CPP] | Verify Data Types match expected types to prevent type-related errors or vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this example a function is to add two void pointers together assuming they are both integers without explicitly verifying. This results in undefined behavior if a floating type is passed instead of an integer as shown below. Without specifying the exact types, the compiler is unable to enforce type safety resulting in runtime errors. |
| #include <iostream>  void unsafeAddition(void\* a, void\* b) {  int result = \*(static\_cast<int\*>(a)) + \*(static\_cast<int\*>(b));  std::cout << “Result: “ << result << endl;  }  int main() {  int x = 2;  double y = 3.5; // this is will cause an issue  unsafeAddition(&x, &y);  return 0;  } |

| **Compliant Code** |
| --- |
| In this example, parameters are specified by exact type ensuring type safety. Whenever possible, utilize C++’s strong typing features over ambiguous types like void pointers. |
| #include <iostream>  void safeAddition(int a, int b) {  int result = a + b;  std::cout << “Result: “ << result << endl;  }  int main() {  int x = 5;  int y = 20;  // now we will get a compiler error if trying to add incorrect types  safeAddition(x, y);  return 0;  } |

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

| **Principles(s):** 1) Validate Input Data: All input data should be verified that it matches expected parameters and causes no issues. Type checks, bounds checking, and other input handling such as parameterized queries significantly reduce risk of vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3 | ALLOC.TM | API07-C Enforce type safety |
| Coverity | 2023.6 | MISRA C 2012 Rule 8.2 | EXP37-C Call functions with the correct number and type of arguments |
| GCC | 13.2 | -Wstrict-prototypes | EXP37-C Call functions with the correct number and type of arguments |
| PC-lint Plus | 2.1 | 9074 | DCL07-C Include the appropriate type information in function declarators |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [INT-002-CPP] | Validate data is within expected range to prevent potential overflow or underflows. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, two integers are added without bounds checking resulting in an overflow and unexpected behavior. |
| #include <iostream>  #include <limits>  void addIntegers(int a, int b) {  int result = a + b;  std::cout << “Result: “ << result << std::endl;  }  int main() {  int x = std::numeric\_limits<int>::max(); // largest integer value  int y = 1; // add 1 to cause an overflow  addIntegers(x, y); // there is no verification of data range so we get an overflow  return 0;  } |

| **Compliant Code** |
| --- |
| This fixed version of the code checks whether or not the addition will result in a overflow, and informs the user instead of performing the potential vulnerable code. |
| #include <iostream>  #include <limits>  bool willOverflow(int a, int b) {  // check for potential overflow  if(b > 0 && a > std::numeric\_limits<int>::max() – b) {  return true; // potential overflow  }    if(b < 0 && a < std::numeric\_limits<int>max() – b) {  return true; // potential underflow  }  return false;  }  void safeAdd(int a, int b) {  if(willOverflow(a, b)) {  std::cout << “Potential overflow/underflow” << std::endl;  } else {  int result = a + b;  std::cout << “Result: “ << result << endl;  }  }  int main() {  int x = std::numeric\_limits<int>::max();  int y = 1;    safeAdd(x, y); // will inform of overflow  return 0;  } |

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

| **Principles(s):** 1) Validate Input Data: All input data should be verified that it matches expected parameters and causes no issues. Type checks, bounds checking, and other input handling such as parameterized queries significantly reduce risk of vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely-Very Likely | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.10 | Integer-overflow | INT08-C Verify that all integer values are in range |
| CodeSonar | 7.3 | ALLOC.SIZE.IOFLOW | INT08-C Verify that all integer values are in range |
| LDRA | 9.8.1 | 488 S | INT08-C Verify that all integer values are in range |
| RuleChecker | 23.10 | stdlib-limits | FLP32-C Prevent or detect domain and range errors in math functions |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-003-CPP] | Utilize safe String functions to avoid potential buffer overflows. Using strncpy over strcpy and snprintf over sprintf allows restricting the number of bytes to prevent bounds issue. |

| **Noncompliant Code** |
| --- |
| In this example strcpy() attempts to copy too many characters overflowing the buffer of the destination string. |
| #include <stdio.h>  #include <string.h>  int main() {  char sourceString[20] = “This is a long text”;  char destString[10];  // this copy will cause a buffer overflow!  strcpy(destString, sourceString);  printf(“Destination: %s\n”, destString);  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code example utilizes strncpy() to specify the number of characters to copy so as not to cause a buffer overflow. Must ensure string ends in null terminator if truncation may occur. |
| #include <stdio.h>  #include <string.h>  int main() {  char sourceString[20] = "This is a long text";  char destString[10];  // strncpy will only copy (sizeof(destination – 1)) characters  strncpy(destString, sourceString, sizeof(destString) - 1);  destString[sizeof(destString) - 1] = '\0'; //ensure null terminator  printf("Destination: %s\n", destString);  return 0;  } |

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

| **Principles(s):** 2) Heed Compiler Warnings, 3) Architect and Design for Security Policies: Do not ignore compiler warnings on potential vulnerabilities and deprecations. Understand and be aware of most current best practices and utilize safer more secure versions of functions whenever available. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.10 | Astrée supported | STR07-C Use the bounds-checking interfaces for string manipulation |
| Parasoft | 1.0.4 | CERT\_C-STR07-a | STR07-C Use the bounds-checking interfaces for string manipulation |
| PC-lint Plus | 2.1 | 586 | STR07-C Use the bounds-checking interfaces for string manipulation |
| SonarQube C/C++ Plugin | 9.9.4 | S1081 | STR07-C Use the bounds-checking interfaces for string manipulation |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STR-004-CPP] | Use prepared statements and parameterized queries to avoid potential SQL injection attacks. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example user input is directly appended to a query which creates a potential for malicious SQL injection. |
| string queryShell = “SELECT \* FROM users WHERE name = ‘”;  string input;  cin >> input;  /\* directly concatenating strings into final query susceptible to injection! \*/  string finalQuery = queryShell + input + “’”;  executeSQLQuery(finalQuery); // potential vulnerable execution! |

| **Compliant Code** |
| --- |
| In this revised version of above, the query is prepared and user input is safely bounded as a parameter preventing it from executing malicious SQL code. |
| string sqlPrepared= “SELECT \* FROM users WHERE name = ?”;  string input;  cin >> input;  // user input treated as data not part of SQL command  executeSQLQuery(sqlPrepared, input); |

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

| **Principles(s):** 1) Validate Input Data, 8) Practice Defense In Depth: SQL injection attacks are always a risk and should be defended against through layered defense strategy such as input validation, parameterized SQL queries, and heuristic filter for potential attack patterns. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Very Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3 | IO.INJ.SQL | STR02-C Sanitize data passed to complex subsystems |
| Klocwork | 2024.1 | SV.TAINTED.INJECTION | STR02-C Sanitize data passed to complex subsystems |
| LDRA | 9.8.1 | 108 D | STR02-C Sanitize data passed to complex subsystems |
| Parasoft | 1.0.4 | CERT\_C-STR02-a (b/c) | STR02-C Sanitize data passed to complex subsystems |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-005-CPP] | Make use of smart pointers over raw pointers to ensure proper deallocation of memory. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example memory is manually deallocated within processFunction() which potentially never gets executed if an error is thrown within that function. This results in a potential memory leak. |
| #include <iostream>  class SomeResource {  public:  SomeResource() {std:: cout << “Acquired.\n”;}  ~SomeResource() {std::cout << “Released.\n”;}  };  void processFunction() {  SomeResource\* resource = new SomeResource();  // if exception occurs here and an early return results  // resources will not be deallocated  delete resource;  }  int main() {  processFunction();  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant example makes use of std::unique\_ptr which automatically deallocates when out of scope preventing memory leaks. |
| #include <iostream>  #include <memory>  Class SomeResource {  public:  SomeResource() { std::cout << “Acquired.\n”; }  ~SomeResource() { std::cout << “Released.\n”; }  };  void processFunction() {  std::unique\_ptr<SomeResource> resource = std::make\_unique<SomeResource>();  // even if exception occurs, resource will be automatically deleted when unique\_ptr goes out of scope  }  int main() {  processFunction();  return 0;  } |

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

| **Principles(s):** 4) Keep It Simple, 6) Adhere to Principle of Least Privilege, 9) Use Effective Quality Assurance Techniques: Use of Smart pointers simplifies some complexity of memory management and leads to less error-prone code. Whenever memory is accessed, it should be released as soon as operations are completed, not holding anything longer than necessary which is automatically handled with smart pointers. Ensure code meets modern best practices and standards through quality assurance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft | 1.0.4 | Runtime Analysis | EXP34-C Do not dereference null pointers |
| CppCheck | 2.13.0 | nullPointer | EXP34-C Do not dereference null pointers |
| ECLAIR | 3.12.0 | CC2.EXP36 | EXP36-C Do not cast pointers into more strictly aligned pointer types |
| RuleChecker | 23.10 | restrict | EXP43-C Avoid undefined behavior when using restrict-qualified pointers |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [MSC-006-CPP] | Use Assertions to verify the internal system is valid, not to handle issues. |

| **Noncompliant Code** |
| --- |
| In this example, an Assertion is used for input validation which is incorrect. Assertions may be removed from production code, input validation should be handled separately with error handling and boundary checks. |
| #include <cassert>  #include <iostream>  void setAge(int age) {  // incorrect usage of assert  assert(age >= 0 && age <= 100 && “Age must be between 0 and 100”);  // proceeds to set invalid value for age  std::cout << “Age set to: “ << age << std::endl;  }  int main() {  int userAge;  std::cout << “Enter your age: “;  std::cin >> userAge;  setAge(userAge);  return 0;  } |

| **Compliant Code** |
| --- |
| In this example, an assertion is used to make sure the sum of vector elements is not zero which should never occur. Even were this assertion removed from code for production, it would not change the logic. The assertion message can be useful for future maintenance and debugging as well. |
| #include <cassert>  #include <vector>  void normalizeVector(std::vector<int>& vect) {  int sum = 0;  for (int value : vect) {  sum += value;  }  assert(sum != 0 && “Sum of vector elements must not be zero before normalization”);  for (int& value : vect) {  value /= sum; // normalize each element by sum  }  }  int main() {  std::vector<int> myVec = {1, 2, 3};  normalizeVector(myVec);  return 0;  } |

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

| **Principles(s):** 3) Architect and Design for Security Policies, 9) Use Effective Quality Assurance Techniques: Use of Assertions can help identify issues early in development and help in testing. Assertions aid in quality assurance testing as well as providing clarity code functionality. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.10 | static-assert | MSC40-C Do not violate constraints |
| Clang | 18.1.3 | misc-static-assert | DCL03-C Use a static assertion to test the value of a constant expression |
| CodeSonar | 7.3 | Customization | DCL03-C Use a static assertion to test the value of a constant expression |
| Parasoft | 1.0.4 | CERT\_C-ERR06-a | ERR06-C Understand the termination behavior of assert() and abort() |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [EXC-007-CPP] | All exceptions must be handled. In C++ when no matching exception handler is found, the default call is to std::terminate() which calls std::abort() resulting in an abnormal process termination and the stack not being unwound. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, an exception is thrown but not properly handled. This results in std::terminate() being called which does not check if the stack has been unwound or not. Since neither the main function, nor the ‘func’ function properly handle the exception thrown in failedThrowFunc, std::terminate() is called. |
| void failedThrowFunc() noexception(false);  void func() {  failedThrowFunc();  }  int main() {  func();  } |

| **Compliant Code** |
| --- |
| In this example, the noncompliant code above has been rectified by including exception handling in the main method. |
| void throwFunc() noexception(false);  void func() {  throwFunc();  }  int main() {  try {  func();  } catch (…) {  // handle the exception here  }  } |

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

| **Principles(s):** 3) Architect and Design for Security Policies, 4) Keep it Simple, 9) Use of Effective Quality Assurance Techniques: Ensuring all possible exception events are handled in some manner and the system is able to recover gracefully mitigates vulnerabilities from unaccounted fail states. Even basic exception handling can help provide more stability and predictability to a system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium/High | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA | 9.8.1 | 122 S | ENV32-C All exit handlers must return normally |
| CodeSonar | 7.3 | BADFUNC.ABORT | ENV32-C All exit handlers must return normally |
| Helix QAC | 2024.1 | DF4856-7-8 | ENV32-C All exit handlers must return normally |
| TrustInSoft Analyzer | 1.47 | Pointer arithmetic | ERR33-C Detect and handle standard library errors |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Management | [FLM-008-CPP] | Ensure all files are properly closed when no longer needed. Files should be closed within the same scope that they are opened immediately after their intended use to preserve data integrity and efficient resource management. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, a file is opened for reading but never closed after data is read. Explicitly closing the file after use not only ensures no resource leaks, but also informs other developers that the file is no longer needed in the rest of the code. |
| #include <fstream>  #include <iostream>  void readFileContents(const std::string& filePath) {  std::ifstream file(filePath);  if (!file.is\_open()) {  std::cerr << “Failed to open file: “ << filePath << std::endl;  return;  }  std::string line;  while (getline(file, line)) {  std::cout << line << std::endl;  }  // the file is never explicitly closed in this function  // violating the coding standard  }  int main() {  readFileContents(“example.txt”);  return 0;  } |

| **Compliant Code** |
| --- |
| In this version of the code from above, the file is immediately closed after data is read from it and it is no longer needed. This ensures proper resource management and prevents potential resource leaks. |
| #include <fstream>  #include <iostream>  void readFileContents(const std::string& filePath) {  std::ifstream file(filePath);  if (!file.is\_open()) {  std::cerr << “Failed to open file: “ << filePath << std::endl;  return;  }  std::string line;  while (getline(file, line)) {  std::cout << line << std::endl;  }  file.close(); // explicitly close the file after use  // potential other operations that do not require the file being open  }  int main() {  readFileContents(“example.txt”);  return 0;  } |

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

| **Principles(s):** 2) Heed Compiler Warnings, 3) Architect and Design for Security Policies, 9) Use of Effective Quality Assurance Techniques: File manipulation is generally flagged by the compiler when risk of resource leaks or unclosed files may occur. Ensure proper file handling techniques are in place and resources are managed effectively to prevent potential vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.10 | Supported, but no explicit checker | FIO42-C Close files when they are no longer needed |
| CodeSonar | 7.3 | ALLOC.LEAK | FIO42-C Close files when they are no longer needed |
| Coverity | 2023.6 | USE\_AFTER\_FREE | FIO46-C Do not access a closed file |
| Klocwork | 2024.1 | RH.LEAK | FIO4(2)2-C Close files when no longer needed and before spawning processes |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Value-returning functions must return value from all exit paths | [EXP-009-CPP] | Functions that return a value must do so for every potential logic path. Not returning a value from the function will result in undefined behavior. Ensure this covers exception handling paths as well. |

| **Noncompliant Code** |
| --- |
| In the following example, an exception occurs but does not specify a return value resulting in undefined behavior. All logic paths must return something for value-returning functions. |
| #include <vector>  std::size\_t func(std::vector<int> &vect, std::size\_t s) try {  vect.resize(s);  return s;  } catch (…) {  // no specified return value here  } |

| **Compliant Code** |
| --- |
| In this corrected version of the above example, even when an exception occurs a value is still returned. SEI Cert C++ Coding standards specify that flowing off the end of the main() function is the same as a return 0 preventing undefined behavior. |
| #include <vector>  std:: size\_t func(std::vector<int> &vect, std::size\_t s) try {  vect.resize(s);  return s;  } catch (…) {  return 0; // specify return value from all logic paths  } |

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

| **Principles(s):** 2) Heed Compiler Warnings, 9) Use Effective Quality Assurance Techniques: Compilers typically will throw warnings for functions that are missing a return value that was specified. Static testing techniques of code base to ensure proper code coverage decreases risk of vulnerabilities and leads to more maintainable code. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 23.10 | function-return-type | MSC40-C Do not violate constraints – return values if specified |
| CodeSonar | 7.3 | LANG.FUNCS.IRV | EXP12-C Do not ignore values returned by functions |
| Coverity | 2023.6 | MISSING\_RETURN | MSC37-C Ensure that control never reaches the end of a non-void function |
| RuleChecker | 23.10 | Function-return-type | DCL31-C Declare identifiers before using them |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Ensure Iterator Range Safety | [ARR-010-CPP] | Prevent out-of-range errors and undefined behavior by ensuring all iterators point to valid elements within their respective containers. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example elements are erased from a vector while iterating over it. The operation leads to the iterator being invalidated and subsequent undefined behavior. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  for(auto i = numbers.begin(); i != numbers.end(); ++i) {  if (\*i % 2 == 0) {  numbers.erase(i); // this causes undefined behavior on the next loop  }  }  for (int num : numbers) {  std::cout << num << “ “;  }  std::cout << std:: endl;  return 0;  } |

| **Compliant Code** |
| --- |
| This corrected version of the above code ensures that the iterator is only incremented when an element is erased. This way the iterator pointer only references valid elements and undefined behavior is prevented. |
| #include <iostream>  #include <vector>  int main() {  std::vector<int> numbers = {1, 2, 3, 4, 5};  for(auto i = numbers.begin(); i != numbers.end(); /\*not incrementing here \*/ ) {  if (\*i % 2 == 0) {  i = numbers.erase(i);  } else {  ++i; // only update if element has been erased  }  for(int num : numbers) {  std::cout << num << “ “;  }  std::cout << std::endl;  return 0;  } |

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

| **Principles(s):** 1) Validate Input Data, 9) Use Effective Quality Assurance Techniques: Iterators should be treated as input to functions and as such should be subject to the same rigor of bounds testing to ensure no invalid memory accesses. Code analysis should identify potential issues with iterators and should generally be checked for common vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2024.1 | ABV.ITERATOR | EXP08-C Ensure pointer arithmetic is used correctly |
| Astrée | 23.10 | array-index-range | ARR30-C Do not form or use out-of-bounds pointers or array subscripts |
| TrustInSoft Analyzer | 1.47 | Index\_in\_address | ARR30-C Do not form or use out-of-bounds pointers or array subscripts |
| ÉCLAIR | 3.12.0 | CC2.FLP30 | FLP30-C Do not use floating-point variables as loop counters |

### 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 with 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.

To automate enforcement of the standards defined in this policy, security will be implemented at various stages throughout the existing pipeline. The Assess and plan stage can be supplemented with automated tracking of threats and regulatory changes to stay on top of the latest developments. Design and Build phases should include initial setup of automated testing with static code analysis and unit testing, threat modeling, and software composition analysis tools for verifying third party dependencies. Verify and test phase should include dynamic runtime analysis tools to verify code behavior in a live environment with possible automated penetration testing to simulate attacks. Lastly, use of containers can increase isolation and allow for further automated security checks enhancing security measures.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DAT-001-CPP | High | Unlikely | Medium | High | 2 |
| INT-002-CPP | High | Likely-Very Likely | Medium | High | 3 |
| STR-003-CPP | High | Likely | Low | High | 3 |
| STR-004-CPP | High | Very Likely | Medium | High | 5 |
| MEM-005-CPP | Medium | Likely | Low | High | 2 |
| MSC-006-CPP | Low | Unlikely | Low | Low | 1 |
| EXC-007-CPP | High | Likely | Medium-High | Medium | 4 |
| FLM-008-CPP | Medium | Likely | Low | Medium | 3 |
| EXP-009-CPP | High | Likely | Low | Medium | 1 |
| ARR-010-CPP | High | Medium | Low | High | 3 |

### 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 rests protects data that is stored such as on a server or hard drive. With proper encryption, even if the storage medium falls into unauthorized hands, without the key it is difficult or impossible to access the data. Protecting data at rest significantly reduces risk and impact of potential breaches. |
| Encryption in flight | Encryption in flight secures data as it moves across networks by encrypting it before transmission and decrypting it only upon arrival at the intended recipient. This process is essential for protecting against vulnerabilities, such as man-in-the-middle attacks and is crucial for maintaining the integrity and confidentiality of sensitive data. |
| Encryption in use | Encryption in use protects data that is actively being accessed or processed, particularly critical in environments like cloud computing where data might be exposed in multi-tenant or less controlled settings. This encryption method can help prevent data breaches and exposures by bad actors and is crucial to maintaining confidentiality. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a process used to verify the identity of users or entities before granting access to resources or a network. Methods of authentication include User Logins, passwords, pins, security tokens, and multifactor authentication involving combinations of methods. Strong Authentication measures can act as a strong part of a defense but often goes hand in hand with Authorization and Accounting. |
| Authorization | Authorization encompasses what access authenticated users, systems, or entities have to resources. This can cover things such as access levels for administrators and users, read/write access to the database, or creating new users. The Principle of Least Privilege ensures that users and processes only have access to the resources required to complete their tasks and nothing else. This can help limit the damage attackers could do if they compromise some accounts with limited access levels. |
| Accounting | Accounting covers logging of events and monitoring for anomalous system behavior that could indicate potential bad actors. Many regulatory requirements include supplying audit trails of user activities for review in case of breaches. Accounting also covers resource monitoring and verifying system health which can help reduce maintenance costs. All events within the system should be accounted for including user activity, database modifications, and file accesses by users. Many if not all accounting can be assisted through automation tools. |

**\***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 | 03/24/2024 | Security Principles & Coding Standards | Joseph Veneski |  |
| 1.2 | 04/14/2024 | Risk Assessment, Automated Detection, Automation, Summary of Risk Assessments, Encryption Policies, Mapping the Principles | Joseph Veneski |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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