**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 | It is important to validate any input that comes from all sources of data. User input is often the source of many different types of attacks, such as SQL Injection and Buffer Overflows. Validations can be created to check for specific input types, such as integers or chars, lengths of these inputs, or input patterns such as “OR Value = Value”, greatly reducing the number of vulnerabilities that exist in a program. |
| 1. Heed Compiler Warnings | Warnings from the compiler can be important to determine if there are issues in the code that can be exploited as vulnerabilities by hackers. These warnings can include issues with the code itself, or if we are using a program such as cppcheck, can include feedback regarding code that is easily exploited or missing key security checks. |
| 1. Architect and Design for Security Policies | The key to creating a secure program is to start the design with security in mind, rather than leaving this until after the program is already developed. Building security features into the core code itself will make the entire program more secure. You can also create different functions within a system that require different security permissions, restricting who can access the program and make changes and increasing the overall security of the design. |
| 1. Keep It Simple | There is often no need for overly complex code to build a program, and simpler code is easier to debug. As code gets more and more complex, there is a greater chance that errors will be made in the code, leaving places where vulnerabilities can be found and exploited. |
| 1. Default Deny | It is often better to deny everyone access to a program by default, and then force them to gain access through specific means rather than to allow anyone complete access from the beginning. Users should have to use secure methods to access different parts of the program, such as logging in with a username and password. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege goes hand in hand with Default Deny. Users should only have access to what they need in order to use a program, and nothing more. For example, a teacher might only have access to their online Grade Book, but not the ability to make changes to the Grading Scales or Report Card Templates. Providing the least amount of access helps keep a program secure by preventing accidental security breaches or malicious code from being entered in these more secure components of the program. |
| 1. Sanitize Data Sent to Other Systems | Any data that is stored or sent to another system that connects to a program should be checked for errors or malicious code, or encrypted so that it cannot be manipulated during transmission. This prevents hackers from getting access to sensitive information that may be stored outside of the program itself, such as within a database or on a secure server. If this data is not properly checked or secured, it could become corrupted or exposed, making it vulnerable to attacks like injection, spoofing, or data breaches. |
| 1. Practice Defense in Depth | Defense in Depth involves implementing multiple layers of security throughout a system, with each layer designed to address specific threats or vulnerabilities. By using overlapping protections, this approach creates a more complete security system that can defend against a wide range of potential attacks. Even if one layer is compromised, the others continue to provide protection. Examples of these layers include AES encryption, completing regular software updates, using a firewall, and restricting access by requiring complex passwords for logins. |
| 1. Use Effective Quality Assurance Techniques | Testing code is an important way to be sure it meets the necessary standards to prevent security breaches. Techniques such as code reviews or other static testing methods to examine missing or vulnerable coding sequences, or dynamic testing techniques such as unit tests can help identify possible issues when the code is running. This can help find issues that may occur when incorrect user input is entered, preventing SQL injections. Having other developers review the code as well, including those that did not work on the program or project is another technique that can be used to identify flaws in the program that can lead to bugs in the future. |
| 1. Adopt a Secure Coding Standard | Using known security standards that exist for your coding language, such as CERT or OWASP can help developers build code that utilizes industry standard security best practices. Since these standards also identify key coding errors, it is helpful when checking code for potential known issues that may expose the program to vulnerabilities. |

### 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] | Call functions with the correct number and type of arguments |

| **Noncompliant Code** |
| --- |
| In this example, the user input type is not checked, and we are just assuming it is an integer. If another type such as a char or string is entered, it could crash or corrupt the program. |
| void checkInput(std::any userInput) {  int number = std::any\_cast<int>(userInput);    std::cout << "Value is: " << number << std::endl;  } |

| **Compliant Code** |
| --- |
| Here, we are checking to see if the userInput is an integer, and if not, we throw an error. This prevents any values other than integers from being used, which can herlp prevent the addition of malicious data or a crash if other types such as floating points are used. |
| void checkInput(std::any userInput) {  if (userInput.type() != typeid(int)) {  throw std::invalid\_argument("Error: Expected an int argument.");  }  int number = std::any\_cast<int>(userInput);    std::cout << "Value is: " << number << std::endl;  } |

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

| **Principle(s):** ValidateInput Data – this principle calls for the use of checks to ensure that input data is valid. We should validate all function calls to be sure we are using the correct type of data and correct number of arguments using if/else statements and throwing errors/exceptions if this data does not match the expected input. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium/High | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13.3 | typeError | Identifies type mismatches in function calls and variable use. |
| Clang-Tidy | 17.0.6 | cppcoreguidlines-pro-type-cstyle-case | Flags unsafe casting and recommends safer alternatives |
| SonarQube | 10.3 | C:S2095 | Detects unchecked input conversions and improper data handling |
| Coverity | 2024.06 | FORWARD\_NILL, WRONG\_TYPE\_ARG | Detects type issues and argument errors |

#### STD-002-cppCoding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Never hard code sensitive information |

| **Noncompliant Code** |
| --- |
| In this example we are checking a password that has been entered into the program as a string by directly comparing this to the user’s input. |
| void passwordFunction(){  std::string password = "password123";  std::string userInput;  std::cout << "Enter password: ";  std::cin >> userInput;  if (userInput == password) {  std::cout << "Access granted.\n";  } else {  std::cout << "Access denied.\n";  }  return 0;  } |

| **Compliant Code** |
| --- |
| Here, we can store passwords in an encrypted file so it is not accessible in the program’s code. |
| void passwordFunction() {  std::ifstream passwordFile("password.txt");  std::string password;  if (!passwordFile) {  std::cerr << "Error: Could not open password file.\n";  return;  }  std::getline(passwordFile, password);  passwordFile.close();  std::string userInput;  std::cout << "Enter password: ";  std::cin >> userInput;  if (userInput == password) {  std::cout << "Access granted.\n";  } else {  std::cout << "Access denied.\n";  }  } |

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

| **Principle(s):** Adopt a Secure Coding Standard: This principle reflects best practices from CERT and OWASP, which discourages hard coding any sensitive information into code, such as passwords and encryption keys. Data that needs to be stored should be properly hashed and encrypted to prevent access by hackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | **HARDCODED.AUTH HARDCODED.DNS HARDCODED.KEY HARDCODED.SALT HARDCODED.SEED** | Hardcoded Authentication Hardcoded DNS Name Hardcoded Crypto Key Hardcoded Crypto Salt Hardcoded Seed in PRNG |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC41-a | Do not hard code string literals |
| PC-lint Plus | 1.4 | 2460 | Assistance provided: reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate |
| Polyspace Bug Finder | R2024b | CERT C: Rule MSC41-C | Checks for hard coded sensitive data (rule partially covered) |

#### Coding Standard 3

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

| **Noncompliant Code** |
| --- |
| In this example, a pointer is attached to a newly created string literal and then the first position of this is modified. This can cause undefined behavior and an access violation, since the values being pointed to should not be changing position. |
| void modifyStringLiteral(){  char \*str = "string literal";  str[0] = 'A';  } |

| **Compliant Code** |
| --- |
| In this code, the string literal is initialized into an array rather than having a pointer, allowing changes to be made to the string without causing undefined behavior. |
| void modifyStringLiteral(){  char str[] = "string literal";  str[0] = 'A';  } |

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

| **Principle(s):** **Validate Input Data:** Prevents invalid or malicious input that might lead to attempts to improperly manipulate data like string literals.  **Keep it Simple**: Avoid complex code that tries to modify string literals directly; instead, use simple, safe approaches like copying literals into mutable arrays.  **Adhere to the Principle of Least Privilege**: Limit what parts of the code/functions can modify data, ensuring that immutable data remains untouched, and reducing the changes that it can be modified by an unauthorized user or other function. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to “char\*” |
| Parasoft C/C++test | 2024.2 | CERT\_C-STR30-a  CERT-C-STR30-b | A string literal shall not be modified  Do not modify string literals |
| Astrée | 24.04 | string-literal-modification  write-to-string-literal | Fully checked |
| TrustinSoft Analyzer | 1.38 | Mem-access | Exhaustively verified, reports undefined behavior |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use parameterized queries to authenticate user input |

| **Noncompliant Code** |
| --- |
| In this example, the user input is added directly to the query, making it easy for SQL injection to take place by adding values to the input that can manipulate the query |
| std::string username;  std::cout << "Enter username: ";  std::cin >> username;  std::string query = "SELECT \* FROM users WHERE username = '" + username + "'";  db.execute(query); // Sends raw query to the database |

| **Compliant Code** |
| --- |
| This compliant code relies on a Boolean that determines if the information passed into the username field is checked before it can be executed in the query. This specific regex statement checks for “OR 1 = 1” injections. |
| bool run\_query(const std::string& sql) {    std::regex sql\_injection(R"(or\s\*(['\"]?\w+['\"]?)\s\*=\s\*\1)", std::regex::icase);  if (std::regex\_search(sql, sql\_injection)) {  std::cout << "SQL Injection attempt detected and blocked.\n";  std::cout << "Blocked query: " << sql << std::endl;  return false;  }  std::cout << "Query executed: " << sql << std::endl;  return true;  }  int main() {  std::string username;  std::cout << "Enter username: ";  std::cin >> username;  std::string sql = "SELECT \* FROM users WHERE username = '" + username  + "'";  run\_query (sql);  } |

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

| **Principle(s):** **Validate Input Data** – Validating input is critical in protecting against SQL injection. In this case, a regular expression checks for patterns commonly used in injection attacks. This helps ensure that only safe, expected input is passed into the SQL query.  **Sanitize Data Sent to Other Systems** – Any input sent to external systems like a database needs to be sanitized or checked. By inspecting the query before it runs, we help protect the system from malicious input that could expose or corrupt data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium/Low | P2 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | IO.INJ.FMT  MISC.FMY | Format String Injection  Formal String |
| Polyspace Bug Finder | R2024b | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |
| Coverity | 2017.17 | TAINTED\_STRING | Tracks user input |
| Cppcheck Premium | 24.11.0 | Premium-cert-fio32-c | Checks files for input/output injection issues |

#### Coding Standard 5

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

| **Noncompliant Code** |
| --- |
| This is an issue because we are using the size of the integer to allocate the space for p, when p is classified as a long. If the value entered requires more space than allocated by an integer type, it will cause a buffer overflow. |
| void function(size\_t len) {  long \*p;  if (len == 0 || len > SIZE\_MAX / sizeof(long)) {  /\* Handle overflow \*/  }  p = (long \*)malloc(len \* sizeof(int));  if (p == NULL) {  /\* Handle error \*/  }  free(p);  } |

| **Compliant Code** |
| --- |
| The correct code fix would be to use the size of long rather than into to allocate the size for p, since p is classified as a long type, and this will ensure we have enough spaces to store these values. |
| void function(size\_t len) {  long \*p;  if (len == 0 || len > SIZE\_MAX / sizeof(long)) {  /\* Handle overflow \*/  }  p = (long \*)malloc(len \* sizeof(long));  if (p == NULL) {  /\* Handle error \*/  }  free(p);  } |

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

| **Principle(s):** **Use Effective Quality Assurance Techniques:** This principle supports the standard by emphasizing the importance of rigorous testing and static/dynamic analysis to detect memory allocation errors early. Proper validation of memory size prevents buffer overflows, a common source of vulnerabilities and crashes.  **Keep It Simple:** Ensuring that memory is allocated with correct sizes follows the simplicity principle by avoiding complicated or error-prone calculations, thus reducing the chance of programmer mistakes leading to security flaws. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.17 | BAD\_ALLOC\_STRLEN  SIZECHECK (depreciated) | Partially implemented  Can find instances where string length is miscalculated (length calculated may be one less than intended) for memory allocation purposes. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary  Finds memory allocations that are assigned to a pointer that reference objects larger than the allocated block |
| Klocwork | 2024.4 | INCORRECT.ACCLOC\_SIZE  SV.TAINTED.ALLOC\_SIZE | Checks for incorrect size allocations Invalidated user input used to determine size allocation |
| Astrée. | 24.04 | Malloc-size-insufficient | Partially checked  Besides firect rule violations, all undefined behaviour resulting from invalid memory accesses is reported by Astrée. |
| Parasoft C/C++test | 2024.2 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to detect and handle unexpected runtime conditions during development. |

| **Noncompliant Code** |
| --- |
| Here, we are dereferencing a pointer but not checking to see if the pointer is null. If so, the program can crash since this is undefined behavior. |
| void process(int\* p) {  \*p = 10;  } |

| **Compliant Code** |
| --- |
| In this code, we first use an assertion to check if the pointer is null. If it is, the program stops and the pointer does not get dereferenced, preventing a crash. |
| void process(int\* p) {  assert(p != nullptr);  \*p = 10;  } |

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

| **Principle(s):** **Heed Compiler Warnings** – Assertions can include messages or stop a program from running, especially for examples such as the null pointer one above.  **Use Effective Quality Assurance Techniques** – Assertions are a great addition to dynamic testing, and can be used to check how well a program runs before it is released. They can help catch errors early and if a function or condition fails, the assertion stops the program immediately, making it easier to trace the issue back to its source  Default Deny: This principle assumes things will go wrong unless explicitly verified otherwise. By asserting that a pointer is not null before dereferencing it, we deny unsafe operations by default. Assertions act as gatekeepers, ensuring that only valid, safe states are allowed to proceed, which helps enforce strict runtime safety boundaries. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.17 | **CHECKED\_RETURN**  **NULL\_RETURNS**  **REVERSE\_INULL**  **FORWARD\_NULL** | Finds instances where a pointer is checked against NULL and then later dereferenced  Identifies functions that can return a null pointer but are not checked  Identifies code that dereferences a pointer and then checks the pointer against NULL |
| Cppcheck Premium | 24.11.0 | **nullPointer,**  **nullPointerDefaultArg, nullPointerRedundantCheck** | Checks for null pointers |
| Polyspace Bug Finder | R2024b | CERT C: Rule EXP34-C | Checks for user of null pointers (rule partially covered) |
| TrustinSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Detect and handle standard library errors |

| **Noncompliant Code** |
| --- |
| In this example, we use a pointer to resize memory, but if this action fails we are left with a null pointer, losing the connection to the original memory block, creating a memory leak. |
| void \*p;  void func(size\_t new\_size) {  if (new\_size == 0) {  /\* Handle error \*/  }    p = realloc(p, new\_size);  if (p == NULL) {  /\* Handle error \*/  }  } |

| **Compliant Code** |
| --- |
| To fix this issue, we can first assign the reallocation to a new pointer q to be sure this is not a null pointer, than it is reassigned back to pointer p. |
| void \*p;  void func(size\_t new\_size) {  void \*q;  if (new\_size == 0) {  /\* Handle error \*/  }    q = realloc(p, new\_size);  if (q == NULL) {  /\* Handle error \*/  } else {  p = q;  }  } |

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

| **Principles(s):** Heed Compiler Warnings: This principle emphasizes the importance of carefully reviewing and addressing compiler warnings and errors. Warnings often point out issues such as failing to verify the result of memory allocation functions like realloc, which can lead to memory leaks or crashes if we do not handle errors properly. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | P27 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | **LANG.FUNCS.IRV LANG.ERRCODE.NOTEST LANG.ERRCODE.NZ** | Ignored return value  Missing Test of Error Code  Non-zero Error Code |
| Cppcheck Premium | 24.11.0 | premium-cert-err33-c | Checks ignored/improperly handled error functions |
| Parasoft C/C++test | 2024.2 | **CERT\_C-ERR33-a CERT\_C-ERR33-d CERT\_C-ERR33-e** | The value returned by a standard library function that may return an error should be used Always check the returned value of non-void function Provide error handling for file opening errors right next to the call to fopen |
| RuleChecker | 24.04 | Error-information-unused | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Buffer Overflow | [STD-008-CPP] | Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| Here we are multiplying to integers, however if this value requires more space than an integer has available, a signed buffer overflow occurs |
| void singedIntMultiplication(signed int A, signed int B) {  signed int result = A \* B;  return result;    } |

| **Compliant Code** |
| --- |
| This code creates a temporary object to store the results of the multiplication and uses long long instead, reducing the possibility of signed overflow. If the value is still too high or low, an error is thrown before the overflow occurs. |
| void signedIntMultiplication(signed int A, signed int B) {  signed int result;  signed long long temp;  temp = (signed long long)A \* (signed long long)B;  // If temp is larger than signed int, throw an error    if (temp > INT\_MAX || temp < INT\_MIN) {  throw std::overflow\_error("Integer overflow detected");  } else {  result = (int)temp;  }  } |

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

| **Principles(s):** Validate Input Data: This principle emphasizes the importance of validating all input, such as checking that operations such as multiplication and addition do not exceed expected bounds, causing an overflow.  Architect and Design for Security Policies: Preventing buffer overflows begin with writing a system of checks into our code such as validating input data, and also throwing errors or exceptions if this data does not comply with these checks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Integer-overflow | Fully checked |
| CodeSonar | 9.0p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Parasoft C/C++test | 2024.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid signed integer overflows  Integer overglow or underflow in '+', '-', '\*' operator  Integer overflow or underflow in '<<' operator |
| TrustinSoft Analyzer | 1.38 | signed\_overflow | Exhaustively tested |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [STD-009-CPP] | Exclude user input from format strings |

| **Noncompliant Code** |
| --- |
| In this example, the print function takes the user input directly as a format string. This can allow hackers to overwrite or add malicious code to the print function, which can create a memory leak or add malicious code to the program. |
| void formatStringExample (const char \*userInput) {  printf(userInput)  ; |

| **Compliant Code** |
| --- |
| Here, we are taking the user input and creating a buffer, then printing this with a fixed format string so the user input is an argument and cannot be manipulated by malicious users. |
| void formatStringExample(const char \*userInput) {    char buffer[1024];    snprintf(buffer, sizeof(buffer), "User said: %s", userInput);    printf("%s\n", buffer);  } |
|  |

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

| **Principles(s):** **Practice Defense in Depth** — This principle applies here by layering protection against format string attacks. By excluding user input from the format string, we reduce the risk of memory corruption and code injection. Using fixed format strings prevents users from adding malicious data into this strings or leaking sensitive data. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | P9 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck Premium | 24.11.0 | premium-cert-fio30-c | Analyze file input/output, checks for file operation issues and buffer overflow risks |
| GCC | 4.3.5 |  | Can detect violations when the -Wformat-security flag is used. |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented, detects format string vulnerabilities |
| Parasoft C/C++test | 2024.2 | **CERT\_C-FIO30-a** **CERT\_C-FIO30-b** **CERT\_C-FIO30-c** | Avoid calling functions printf/wprintf with only one argument other than string constant Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable Never use unfiltered data from an untrusted user as the format parameter |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-010-CPP] | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| In this example we are have a pointer that has been initialized, but the memory it points to has not, learning to an error. This can also leave the system vulnerable, causing a dangling pointer. |
| void newInteger() {  int \*i = newInt;  std::cout << newInt << std::endl;  } |

| **Compliant Code** |
| --- |
| Here, we can use a smart pointer to reallocate memory instead to prevent a memory leak, and avoid dangling pointers. |
| void newInteger() {  std::unique\_ptr<int> i = std::make\_unique<int>(5);  std::cout << \*i << std::endl;  } |

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

| **Principles(s):** **Keep It Simple** - Using C++ features like smart pointers in this example helps avoid complex manual memory management, which is often a source of uninitialized memory errors and dangling pointers.  Heed Compiler Warnings: Compiler warnings can help find unsafe memory usage, such as uninitialized pointers or potential leaks. In this example, paying attention to warnings about uninitialized memory or improper pointer usage can help catch errors early before the program runs, preventing vulnerabilities like dangling pointers or crashes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.MEM.UVAR | Uninitialized variable |
| Cppcheck | 2.15 | unintvar  unintdata  unintstring  unintMemberVar  unintStructMember | Detects uninitialized variables, data, strings |
| Parasoft C/C++test | 2024.2 | CERT\_C-EXP33-a | Avoid use before initialization |
| RuleChecker | 24.04 | Uninitialized-local-read | Partially checked |

### Defense-in-Depth Illustration

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



## Project One

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

### Revise the C/C++ Standards

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

### Risk Assessment

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

### Automated Detection

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

### Automation

Provide a written explanation using the image provided.



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

In order to create an even more robustly secure product and automate enforcement of the standards described above, we should look into making changes in the Verify and Test area in the pre-production phase of the DevSecOps cycle, as well as the Monitor and Detect area once we move into the production phase. In our pre-production phase, we should focus on the use of automated testing tools including Cppcheck Premium and CodeSonar, which can be used to detect issues that may be missed by our compiler. Additionally, assert statements should be enforced during code reviews and verified through automated unit tests as part of the **Verify** stage. These assertions help catch incorrect assumptions and logic errors early in development.

In the **Monitor** and **Detect** stages, even after deployment, logs and automated reports generated by static analysis scans such as those created in Cppcheck should be routinely reviewed to identify recurring issues and ensure continued compliance with the standards listed above.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Probable | Medium/High | P4 | L3 |
| STD-002-CPP | High | Probable | Medium | P9 | L2 |
| STD-003-CPP | Low | Likely | Medium/High | P6 | L2 |
| STD-004-CPP | High | Likely | Medium/Low | P2 | L1 |
| STD-005-CPP | High | Probable | Medium | P6 | L2 |
| STD-006-CPP | High | Likely | Low | P12 | L1 |
| STD-007-CPP | High | Likely | Low | P27 | L1 |
| STD-008-CPP | High | Likely | Low | P18 | L1 |
| STD-009-CPP | Medium | Possible | Low | P9 | L1 |
| STD-010-CPP | High | Probable | Low | P12 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects stored data by converting it into an unreadable format using cryptographic algorithms, so even if someone gains access to the physical storage or database, they cannot read the information without the decryption key. Examples of this can be AES-256 or FireVault for macOS. This type of encryption applies to passwords, user profiles, backup files, or any sensitive information sitting in storage, protecting our company from attackers looking to steal this sensitive user data. |
| Encryption in flight | Encryption in flight protects data as it moves across networks by encrypting it before transmission and decrypting it at the receiving end. This is typically done using protocols like TLS or HTTPS when users are logging in, submitting forms, or transferring files. This matters because attackers could intercept the data in transit, especially over public or insecure networks, and use it maliciously. Encrypting data in flight prevents exposure during these vulnerable points, keeping our company’s data secure across the web. |
| Encryption in use | Encryption data **in use** protects data while it is actively being processed in memory or during computations. Unlike encryption at rest (stored data) or in transit (data moving across networks), data in use is most vulnerable because it must be decrypted for applications to read or manipulate it. Many times this data is overlooked, making it vulnerable to attacks. We want to be sure to use techniques such as enforcing product updates, strong passwords and restrict which users have access to certain data through restricted user tool rights. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies a user’s identity, usually through login credentials like a username and password, but sometimes with more secure methods like multi-factor authentication (MFA) or biometrics. This policy ensures only valid users can access the system by blocking unauthorized access, keeping the system and its data protected from hackers or other malicious users. |
| Authorization | Authorization determines what actions or areas of the system a user is allowed to access once they’re authenticated. This could include restricting read, write, or delete privileges depending on their role. The policy matters because it enforces boundaries, following the Principle of Least Privilege and only providing users access to what they need to work effectively, and not allowing access to the entire system. This limits damage if an account is compromised or if a user makes a mistake and removes or changes important data in the system. |
| Accounting | Accounting tracks what users do while they’re logged in, including logging access times, changes made to the database, files accessed, and other activity. This can help detect unauthorized or unusual login attempts and send a notification alerting the user or the company of this activity. This policy is necessary because it creates a record that can be used to investigate suspicious behavior, track changes to the system, and ensure accountability. |

**\***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 | 05/25/2025 | Template Updated | Cheryl Buzicky | Ramsey Kraya |
| 1.2 | 06/14/2025 | Final Template | Cheryl Buzicky |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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