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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 1](#_Toc1076988286)

[Contents 1](#_Toc2113712612)

[Overview 2](#_Toc1898889132)

[Purpose 3](#_Toc1936575261)

[Scope 3](#_Toc18371628)

[Module Three Milestone 3](#_Toc1737179960)

[Ten Core Security Principles 3](#_Toc1286725407)

[C/C++ Ten Coding Standards 4](#_Toc1919870841)

[Coding Standard 1 4](#_Toc448857775)

[Coding Standard 2 6](#_Toc642276314)

[Coding Standard 3 8](#_Toc991224614)

[Coding Standard 4 10](#_Toc1959848577)

[Coding Standard 5 12](#_Toc867630490)

[Coding Standard 6 14](#_Toc957882383)

[Coding Standard 7 16](#_Toc1169291724)

[Coding Standard 8 18](#_Toc1841975345)

[Coding Standard 9 20](#_Toc1737928118)

[Coding Standard 10 21](#_Toc1803057194)

[Defense-in-Depth Illustration 23](#_Toc394748003)

[Project One 24](#_Toc809630562)

[Revise the C/C++ Standards 24](#_Toc263836711)

[Risk Assessment 24](#_Toc968227767)

[Automated Detection 24](#_Toc1934985731)

[Automation 24](#_Toc358936327)

[Summary of Risk Assessments 25](#_Toc838426196)

[Create Policies for Encryption and Triple A 26](#_Toc62008821)

[Map the Principles 28](#_Toc2079199781)

[Audit Controls and Management 29](#_Toc1693222084)

[Enforcement 29](#_Toc1475412949)

[Exceptions Process 29](#_Toc756049201)

[Distribution 29](#_Toc1068248433)

[Policy Change Control 30](#_Toc1103073772)

[Policy Version History 30](#_Toc1161867759)

[Appendix A Lookups 30](#_Toc1856228642)

[Approved C/C++ Language Acronyms 30](#_Toc1193961820)

## 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 | Individual components in a larger system will interact with several other components through the exchange of data. Components may also receive data from outside the system. Any data not generated solely by the component itself should be treated as untrusted. This untrusted data must be validated. This means that the input should be checked to confirm that it conforms to the expected data type and numeric range that the component is expected to handle. Data that does not conform to requirements should be rejected. |
| 1. Heed Compiler Warnings | Code should be compiled at the highest warning level. This means that all available compiler warnings will be displayed at compilation. Everything that the compiler identifies as suspicious or not in-line with best practices will be flagged. These compiler warnings should not be dismissed. The source code should be modified so that it compiles cleanly without any warnings. |
| 1. Architect and Design for Security Policies | The architecture of the software should be created in a manner that allows for the enforcement of security policies. This means that each subsystem should operate only at minimum required permission level. Components of the architecture which require the strictest adherence to the policy can be given special attention if they are separate by design. |
| 1. Keep It Simple | The system and software design should be kept as simple as possible when implementing the required features. More complex systems become more difficult to implement. This creates more opportunities for errors to be made which can introduce security vulnerabilities. |
| 1. Default Deny | Access to the system and the system’s components should be done via explicit individual permissions. This is opposed to specific exclusions which identity connections that are not allowed instead of only those who are. |
| 1. Adhere to the Principle of Least Privilege | The permission level for individual system components and the system as a whole should be kept to the lowest level possible to complete system tasks. When a higher level of permission is needed, the lower level permissions should be reinstated once the higher level tasks are completed. |
| 1. Sanitize Data Sent to Other Systems | Data that is passed from one subsystem to another should be sanitized by the calling subsystem. This should identify and remove potential injection attacks that could be passed to command shells and relational databases. |
| 1. Practice Defense in Depth | No single defense strategy should be used to eliminate a given risk. The application of multiple defense options create layers of mitigation so that if one defense strategy proves ineffective at stopping a given threat, another layer will defeat it. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques should be used to confirm that the system is secure. This can include penetration testing, automated testing which includes invalid inputs, and audits of source code. |
| 1. Adopt a Secure Coding Standard | A standard should be developed and adhered to through development and deployment. This document serves to establish such a standard and show not be deviated from. |

### 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** | **Choose Appropriate Data Types** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | All data types used in programming should be understood. Specifically, appropriate data types should be chosen for size and function. Only appropriate assumptions about proper data types should be made. Any assumptions should be confirmed via static assertions or run time error handling. |

| **Noncompliant Code** |
| --- |
| The choice of the char data type is obviously too small for the defined function. The function retrieves a bank checking account balance. The char data type has a max value of 255 which is way too low for most checking accounts. Additionally, the char data type does not include decimals which means information could be lost. |
| int main() {  char balance;  int userId;  // ...  balance = getCheckingBalance(userId);  } |

| **Compliant Code** |
| --- |
| The choice of long double is more appropriate to store a bank balance. It provides enough range and allows for the storing of a decimal value. |
| int main() {  long double balance;  int userId;  // ...  balance = getCheckingBalance(userId);  } |

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

| **Principles(s):**   1. Validate user input: user input should be validated to confirm data type expectations.   9. Use Effective Quality Assurance Techniques: Data types should be tested for conformance and  proper usage confirmed via auditing.  10. Adopt a Secure Coding Standard: Data type usage should be consistent throughout the system, and  proper data types should be determined ahead of time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.1p0 | BADFUNC.BO.\*  LANG.MEM.BO  LANG.MEM.TBA | A collection of warning classes that report uses of library functions prone to internal buffer overflows. |
| Polyspace Bug Finder | R2024b | CERT C++: CTR52-CPP | Checks for library functions overflowing sequence container (rule partially covered). |
| Security Reviewer – Static Reviewer | 6.02 | C01  C04 | Fully implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Avoid Calculations that Result in Overflows** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Data values should be confirmed to be of appropriate size prior to being stored in a variable. Calculations should be confirmed to not result in buffer overflow or to cause integer wraparound. These errors could result in undefined program behavior which introduces vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This function takes in outside parameters that are used to perform repetitive addition. There are no checks in place to confirm that integer overflow does not occur. This could result in wraparound and the returning of the incorrect value. |
| int add\_numbers(int const& start, int const& increment, unsigned long int const& steps) {  int result = start;  for (unsigned long int i = 0; i < steps; ++i) {  result += increment;  }  return result;  } |

| **Compliant Code** |
| --- |
| This function checks for integer overflow before performing addition. If the calculation would result in overflow, the starting value is returned which indicates to the calling function that an error was encountered. This effectively prevents integer overflow. |
| int add\_numbers(int const& start, int const& increment, unsigned long int const& steps) {  int result = start;  int typeMax = std::numeric\_limits<int>::max();  for (unsigned long int i = 0; i < steps; ++i) {  if (result > typeMax – increment) {  return start;  }  result += increment;  }  return result;  } |

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

| **Principles(s):**  1. Validate user input: user input should be validated to confirm data type expectations.  9. Use Effective Quality Assurance Techniques: Data types should be tested for conformance and  proper usage confirmed via auditing.  10. Adopt a Secure Coding Standard: Data type usage should be consistent throughout the system, and  proper data types should be determined ahead of time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | integer-overflow | Fully checked |
| Axivion | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 9.1p0 | 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 | Checks for:  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-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoids wraparounds when performing arithmetic integer operations, integer overflow or underflow in constant expression in ‘+’, `-`, `\*` operator, and integer overflow in constant expression in `<<` operator. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Avoid String Overflow and Truncation** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | While manipulating strings, it is possible for data loss to occur if they are not handled properly. C++ defined String data type should be used for dynamically allocated memory when possible. If a character array is used, it should be of significant length to avoid overflow or truncation. |

| **Noncompliant Code** |
| --- |
| This code example copies the character data from the location of pointer string\_data into the character array a. If the string\_data does not contain a null character in the first 16 characters, then a non-null terminating string will be copied into a. Any string data stored at the string\_data pointer may be truncated when copying. |
| int main() {  char \*string\_data;  char a[16];  // Some code, potentially manipulating strings  strncpy(a, string\_data, sizeof(a));  } |

| **Compliant Code** |
| --- |
| This function correctly addresses the possibility of a null pointer to character data. If the pointer is null, the string is never copied. The length of the string data is also confirmed before copying. If the string is longer than the character array a, the string data is never copied. Finally, if no errors were detected, the string is copied. |
| int main() {  char \*string\_data = NULL;  char a[16];  // Some code, potentially manipulating strings  if (string\_data == NULL) {  // Null pointer error to handle  }  else if (strlen(string\_data) >= sizeof(a) {  // Handle error where string data is too long  }  else {  strcpy(a, string\_data);  }  } |

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

| **Principles(s):**  4. Keep it simple: consistent use of C++ defined string data type simplifies data type usage  9. Use Effective Quality Assurance Techniques: Dynamically allocated data helps to ensure expected program behavior.  10. Adopt a Secure Coding Standard: Consistent data type usage is part of a complete Secure Coding Standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Security Reviewer – Static Reviewer | 6.02 | C24 | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Use Prepared Statements When Querying** |
| --- | --- | --- |
| **SQL Injection** | STD-0004-CPP | Prepared Statements will be used when completing SQL queries. This handles SQL queries in a manner that prevents injection. SQL injection vulnerabilities allow an outside source to construct the SQL query through string manipulation. This can have undesired effects such as returning sensitive user information to an attacker. |

| **Noncompliant Code** |
| --- |
| This code example uses string concatenation to create a SQL query string using user input. This is a bad practice as it allows the user to directly alter the SQL string. The user could enter `validuser ‘ or ‘1’ = ‘1’’` creating a tautology that can bypass authentication. |
| public void completeQuery(String username, char[] password) {  // Create database connection  try {  String sqlString = “SELECT \* FROM db\_user WHERE username = ‘”  + username +  “ ‘ AND password = ‘” + password + “`”;  Statement stmt = connection.createStatement();  ResultSet rs = stmt.executeQuery(sqlString);  // Process returned query  }  } |

| **Compliant Code** |
| --- |
| This code example uses a parametric query with argument placeholders. When the setString method is used, strong type checking is enforced which mitigates possible SQL injection. |
| public void completeQuery(String username, char[] password) {  // Create database connection  String sqlString = “select \* from db\_user where username=? and  password=?”;  PreparedStatement = stmt = connection.preparedStatement(sqlString);  stmt.setString(1, username);  stmt.setString(2, password);  ResultSet rs = stmt.executeQuery();  // Process returned query  } |

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

| **Principles(s):**  1. Validate input data: User input should be validated to not contain malicious SQL injections.  3. Architect and Design for Security Policies: Subsystems should not have SQL permissions beyond what is required to complete subsystem functionality.  5. Default Deny: SQL queries should only be allowed for authenticated users.  6. Adhere to the Principle of Least Privilege: Subsystems should not have SQL permissions beyond what is required to complete subsystem functionality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented – Confirms properly formatted prepared statements for SQL queries |
| SonarQube | 9.9 | S2077  S3649 | Confirms that executed SQL queries are security-sensitive and that SQL queries are not vulnerable to injection attacks. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Only Access Properly Allocated Memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Secure practices must be followed when accessing memory. This includes not accessing memory outside of defined boundaries, proper memory allocation, freeing of allocated memory once it is no longer needed, freeing memory only once, and not accessing memory once it has been freed. Failure to follow this standard may result in attackers rewriting portions of memory, potentially leading to arbitrary code execution. |

| **Noncompliant Code** |
| --- |
| In the code example below, a pointer is made to a variable of type S. The variable is used for some purpose and deleted to free memory. A member function of that variable is then accessed. This can have undefined behavior which creates a system vulnerability. |
| int main() {  S \*s = new S;  // Pointer to instance of data type S is used  delete s;  s->f(); //Access method of variable s after it has been freed.  } |

| **Compliant Code** |
| --- |
| In this example, the call to the member function of the object of type S is made before the object was deleted. The object is only deleted once it is no longer needed. |
| int main() {  S \*s = new S;  s->f();  delete s;  } |

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

| **Principles(s):**  4. Keep it simple: Abstract datatypes created for the system should be consistent throughout.  9. Use Effective Quality Assurance Techniques: Code should be reviewed for secure memory practices.  10. Adopt a Secure Coding Standard: A secure coding standard specific to the system should list standardized datatypes that will be used along with methods for allocating and freeing data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | dangling\_pointer\_use | Astree reports all accesses to freed allocated memory. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM31 | Can detect dynamically allocated resources that are not freed |
| Coverity | 2017.07 | BAD\_FREE | Identifies calls to free() where the argument is a pointer to a pointer to a function or an array. It also detects the cases where free() is used on an address-of expression, which can never be heap allocated. |
| Polyspace Bug Finder | R2024b | CERT C: Rule MEM35-C | Checks for pointer access out of bounds and memory allocation with tainted size. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Guarantee Expected Program State** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions should be used to confirm the status of assumptions made by the developer. Assertions should be used to confirm the expected state of the system at a given point when the alternative would lead to undefined behavior. Assertions should be treated as a guarantee that a programmer makes that the system state will be correct at the time of assertion. The use of assertions mitigates the risk of programmer error introducing vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The assertion in the example below obviously evaluates to true, as it checks the value of an integer who whose value was assigned on the previous line. Such obvious truths do not need to be asserted. Static assertions of this type add noise to the code which impacts readability, and similar runtime assertions add unneeded overhead. |
| int main() {  int x = 5;  assert(x == 5);  } |

| **Compliant Code** |
| --- |
| The assertion in this code example confirms that a memory address is valid for manipulation. The programmer would expect this to be the case at this point in operation and verifies this through the assertion. |
| int main() {  \_ASSERTE(\_CrtIsValidPointer( address, size, TRUE );  } |

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

| **Principles(s):**  1. Validate Input Data: The state of the program, including input data, can be confirmed via assertions.  8. Practice Defense in Depth: Assertions represent an additional line of defense to confirm the expected program state.  9. Use Effective Quality Assurance Techniques: Static assertions can catch errors at compile time. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Respect Function’s Possible Exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Many functions defined in C++ have an exception specification which defines the exceptions that it may throw. All potential exceptions should be defined for a function should not be overlooked and should be handled appropriately. This confirms that a system is functioning as intended. |

| **Noncompliant Code** |
| --- |
| In the code example below, the function f is defined as not throwing exceptions. However, it makes use of the vector.resize() method which may throw an error. This fails to respect potential exceptions. |
| void f(std::vector<int> &v, size\_t s) noexcept(true) {  v.resize(s);  } |

| **Compliant Code** |
| --- |
| In this code example, the function now allows for the exceptions that may be thrown by the resize method. |
| void f(std::vector<int> &v, size\_t s){  v.resize(s);  } |

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

| **Principles(s):**  1. Validate Input Data: Exceptions may be able to catch invalid user input.  3. Architect and Design for Security Policies: Exceptions may be thrown when advanced permissions are sought. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++Test | 2024.2 | CERT\_CPP-ERR55-a | Where a function’s declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated types. |
| Polyspace Bug Finder | R2024b | CERT C++:ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Handle Memory to Prevent Leaks** |
| --- | --- | --- |
| Avoid Memory Leaks | STD-008-CPP | Memory leaks occur when data is allocated but never freed. This may result in memory exhaustion and makes a system vulnerable to denial of service attacks. |

| **Noncompliant Code** |
| --- |
| The function allocates memory for text\_buffer; however, that memory is not freed nor is it returned. Each time this function runs, it results in a memory leak. |
| void f(void){  char \*text\_buffer = (char \*)malloc(BUFSIZ);  if (text\_buffer == NULL) {  return –1;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This function properly frees the memory allocated for text\_buffer before returning. |
| int f(void){  char \*text\_buffer = (char \*)malloc(BUFSIZ);  if (text\_buffer == NULL) {  return –1;  }  free(text\_buffer);  return 0;  } |

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

| **Principles(s):**  9. Use Effective Quality Assurance Techniques: Proper code reviews should be completed to identify possible memory leaks.  10. Adopt a Secure Coding Standard: Consistent procedures should be used for allocating and freeing memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024b | CERT C: Rule MEM31-C | Checks for memory leak |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do Not Modify String Literals** |
| --- | --- | --- |
| String Literals | STD-009-CPP | String literals are allocated to static storage when a program is compiled. Modifying string literals results in undefined behavior because this is static memory. |

| **Noncompliant Code** |
| --- |
| This function assigns a pointer to the a string literal. The pointer is then used to directly modify the string literal. This results in undefined behavior and introduces a vulnerability. |
| void f(void){  char \*str = “string literal”;  str[0] = ‘S’;  } |

| **Compliant Code** |
| --- |
| This function avoids the above issue by copying the string literal into a string object. The string object is then modified which is defined behavior. |
| void f(void){  std::string str = “string literal”;  str[0] = ‘S’;  } |

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

| **Principles(s):**  2. Heed Compiler Warnings: Compilers will likely warn the developer when string literals are altered. These warning should be addressed so that the program compiles cleanly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2024b | CERT C: Rule STR30-C | Checks for writing to const qualified objects. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Avoid Using User Input With Format Strings** |
| --- | --- | --- |
| Format Strings | STD-010-CPP | Allowing a user to modify a format string can result in arbitrary code execution. A format string is interpreted as a set of instructions. Failing to follow the standard allows the use to control this execution. |

| **Noncompliant Code** |
| --- |
| This function uses user input in a format string. This can allow the user to create an overflow or even write unsuspected values to memory. |
| void f(const char \*input){  char sample\_string[1024];  snprintf(sample\_string, 1024, “String: %s\n”, input);  printf(sample\_string);  } |

| **Compliant Code** |
| --- |
| This example uses prepared strings instead of relying on user input for the format string. This is preferrable as possible as it removes the vulnerability. |
| void f(void){  char sample\_string[1024];  char prepared\_msg[] = “Prepared Message”;  snprintf(sample\_string, 1024, “String: %s\n”, prepared\_msg);  printf(sample\_string);  } |

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

| **Principles(s):**  10. Adopt a Secure Coding Standard: Secure coding policies should prevent developers from unsafely using format strings by using unsanitized user input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2024.2 | CERT\_C-FIO30-a  CERT\_C-FIO30-b  CERT\_C-FIO30-c | Checks for the following practices: Avoid calling functions printf/wprintf with only one argument other than string constant.  Avoid using functions fprintf/wprintf with only two parameters, when second parameter is a variable.  Never use unfiltered data from an untrusted user as the format parameter. |
| Polyspace Bug Finder | R2024b | CERT C: Rule FIO30-C | Checks for tainted string format. |

### 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.

Automation of development steps ensure that these steps will be completed correctly and efficiently. This helps to prevent developer errors and can identify security vulnerabilities that may otherwise be overlooked. While the DevSecOps pipeline is a continuous cycle, we will start out consideration at the Access and plan stage of development. After this planning phase, we the Design phase should heavily consider automation driven development. The test driven design purported will require test cases to be written. As the design of the system grows, these test cases should be written. Test cases should include both positive and negative test cases. These test cases should consider best security practices such as OWASP. When we move to the build phase, we code the system so that the carefully crafted automated test suite will pass for all tests. During the verify and test phase the automated security test software discussed for each secure coding standard should be applied. The application of these automated tools will help to ensure that developers have properly followed the defined standards. Once the system has moved to production, the an automated alert system should be used to identify any security concerns. Examples of concerns that should trigger alerts are invalid login attempts, attempts to access forbidden resources, and any system failures.

### 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 | Likely | Medium | P18 | L1 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Probable | High | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | High | P9 | L2 |
| STD-006-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-007-CPP | Low | Likely | Medium | P6 | L2 |
| STD-008-CPP | Medium | High | High | P6 | L2 |
| STD-009-CPP | Low | Likely | Medium | P6 | L2 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Data at rest is data that is in long term, non-volatile storage. First and foremost, no sensitive data will be hard coded into programs. Sensitive data at rest will not be stored as plain text data. Data that will eventually be needed as plain text data will be encrypted using the AES-256 encryption algorithm, and encryption keys will be stored separate from encrypted data. Data that will not be needed as again as plain text data, such as user passwords, will be salted and hashed using the SHA-256 hashing algorithm. The encryption and hashing of data at rest ensures that data breeches will not result in malicious access to data that is readable by the malicious party. Encryption keys will be stored separate from the encrypted data so that the data cannot be easily decrypted when accessed illegitimately. User passwords will be hashed. When logging in, the user’s text entry will be salted and hashed and compared to the stored hashed values. Non-default encryption and hashing options will be chosen to obscure the methods of encryption. |
| Encryption in flight | Data in flight is data that is traveling over a network. Data in flight is vulnerable to man in the middle attacks in which malicious parties read or intercept the data being sent which may result in data alteration. To prevent this, data will only be sent via HTTPS connections which encrypt data in flight so that only the intended user will have access to the contents of the request or response. |
| Encryption in use | When working with sensitive data in memory. Careful steps should be taken to avoid accidental data exposure. Sensitive data should only be kept in memory for as long as it is required. As soon as the data has served its purpose, it should be cleared from memory, not only freed. Memory dumps are often enabled by default and provided when a system crashes. When sensitive data is used, these memory dumps should be disabled to prevent unintentional sharing of data. Different in use encryption methods should be considered. For the most sensitive data, Homomorphic Encryption should be used as able. This encryption algorithm allows the manipulation of encrypted data without decryption which makes the system less exploitable if a malicious user is able to exploit a vulnerability to read system data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of confirming that a user is who they claim to be. In order to interact with the system, all users will be required to login. This will be done through a username and password. The username will be saved in a system database, and the password will be hashed with the resulting hashed password saved in the database. If the system is determined to contain sensitive data, multi-factor authentication should be considered. This may be done via a confirmation code sent to an email or phone or via biometric data. When a user logs in, their entered credentials will be compared to the user information stored in the system database. |
| Authorization | Authorization refers to establishing what resources an individual is permitted to use. Once a user has been authenticated, the system should determine activities the user can perform based on user attributes. System authorization will follow the principle of least privilege; that is, the users will only be allowed to perform the activities that are required to complete their role in the system. For example, a read level access user will only be able to read data from system databases and will be assigned to individuals that are not required to make database updates in their roles. This level can be restricted further to limit the data the user can view. Write access roles should also be defined. These are individuals that are required to update or create new records in a system database. Like the read access level, this role will likely be limited by the types of records that it can create. Finally, an administrator role will have full access to the system. They will be able to edit and create user accounts, create database tables, and shutdown the system along with other high access level tasks. If a user with an administrator account does not need to perform administrative tasks during a session a lower level access account should be used to not unnecessarily expose system data and services. System access levels should be assigned to the role the individual fills at a business and should inform the access given to new user accounts. |
| Accounting | Accounting refers to tracking what resources are accessed, when they were accessed, who accessed them, and what tasks were performed with them. There are several reasons to log this information. Accounting helps to determine necessary authorization control policies, trend analysis, and capacity planning. Additionally, logging the user of system resources can help to determine when someone attempt to access resources that should not be accessed by that user and may help to identify attempted exploitations. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 08/14/2025 | Implemented coding standards | Tyler Glover |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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