**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 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 21](#_Toc52464068)

[Defense-in-Depth Illustration 23](#_Toc52464069)

[Project One 23](#_Toc52464070)

[1. Revise the C/C++ Standards 23](#_Toc52464071)

[2. Risk Assessment **Error! Bookmark not defined.**](#_Toc52464072)

[3. Automated Detection **Error! Bookmark not defined.**](#_Toc52464073)

[4. Automation 23](#_Toc52464074)

[5. Summary of Risk Assessments 24](#_Toc52464075)

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

[7. Map the Principles 25](#_Toc52464077)

[Audit Controls and Management 27](#_Toc52464078)

[Enforcement 27](#_Toc52464079)

[Exceptions Process 27](#_Toc52464080)

[Distribution 28](#_Toc52464081)

[Policy Change Control 28](#_Toc52464082)

[Policy Version History 28](#_Toc52464083)

[Appendix A Lookups 28](#_Toc52464084)

[Approved C/C++ Language Acronyms 28](#_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 external input (User input from forms, uploaded files, command prompt input) should be considered suspect. Validate all inputs to ensure they fit the size of the variable used to hold that data. Input vulnerabilities are quite common, so be aware of buffer overruns and underruns, stack overflows and underruns, command line scripts, and SQL injection attacks. |
| 1. Heed Compiler Warnings | Exceptions and errors in code give hackers an end run around your security. A compiler warning will allude to such a vulnerability so ensure all warnings are addressed. Warnings will allow a program to compile and run, so addressing the warning is important. |
| 1. Architect and Design for Security Policies | Design your system to follow secure coding principles. Where necessary handle permission separately, re use code where possible to reduce the possibility for errors and allow easy debugging to fix errors that do arise. |
| 1. Keep It Simple | Complexity means a lot of potential for errors in code. It also makes it more challenging to find and fix those errors. Since coding errors lead to attacks, keeping the code simple, breaking it into modules, makes the code easier to fix if breaches occur and easier to identify errors or reduce them in the first place. |
| 1. Default Deny | Default to denying access unless permissions are given, or authentication is achieved. |
| 1. Adhere to the Principle of Least Privilege | Each user should be given the lowest level of privilege required for the tasks this user needs to accomplish. This ensures that any user will only have basic privileges unless otherwise given by someone authorized to provide those privileges and keep hackers from performing operations basic privilege doesn’t allow. |
| 1. Sanitize Data Sent to Other Systems | Any time data will be used for input, be it from users or between systems, be sure to present the data in a way that will prevent SQL injections and other errors vie processing. Hackers can make use of unused functions if not properly sanitized. |
| 1. Practice Defense in Depth | Using layers of protection that guard against the multitude of vulnerable points for hackers, including encryption, port listening, and so on, while also ensuring you have enough redundancy in place to handle any breaches of existing protocols. |
| 1. Use Effective Quality Assurance Techniques | The best way to make sure an app is quality is to incorporate good testing plans and procedures to ensure as many issues are found and fixed before deployment and a follow up on bugs that could not be caught in development. |
| 1. Adopt a Secure Coding Standard | Consider the vulnerabilities of the language and platform you are planning to use and adopt coding standards surrounding these vulnerabilities along with known vulnerabilities that exist among all languages. Ensure programmers follow this standard by clearly documenting the implementations. |

### 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** | **Use rsize\_t or size\_t for all integer values representing the size of an object** |
| --- | --- | --- |
| **Data Type** | [STD-001-C] | The type size\_t generally covers the entire address space. The C Standard, Annex K (normative), "Bounds-checking interfaces," introduces a new type, rsize\_t, defined to be size\_t but explicitly used to hold the size of a single object [Meyers 2004]. In code that documents this purpose by using the type rsize\_t, the size of an object can be checked to verify that it is no larger than RSIZE\_MAX, the maximum size of a normal single object, which provides additional input validation for library functions. See STR07-C. Use the bounds-checking interfaces for string manipulation for additional discussion of C11 Annex K. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the dynamically allocated buffer referenced by p overflows for values of n > INT\_MAX: |
| char \*copy(size\_t n, const char \*c\_str) {  int i;  char \*p;    if (n == 0) {  /\* Handle unreasonable object size error \*/  }  p = (char \*)malloc(n);  if (p == NULL) {  return NULL; /\* Indicate malloc failure \*/  }  for ( i = 0; i < n; ++i ) {  p[i] = \*c\_str++;  }  return p;  }    /\* ... \*/    char c\_str[] = "hi there";  char \*p = copy(sizeof(c\_str), c\_str); |

| **Compliant Code** |
| --- |
| Declaring i to be of type rsize\_t eliminates the possible integer overflow condition (in this example). Also, the argument n is changed to be of type rsize\_t to document additional validation in the form of a check against RSIZE\_MAX: |
| char \*copy(rsize\_t n, const char \*c\_str) {  rsize\_t i;  char \*p;    if (n == 0 || n > RSIZE\_MAX) {  /\* Handle unreasonable object size error \*/  }  p = (char \*)malloc(n);  if (p == NULL) {  return NULL; /\* Indicate malloc failure \*/  }  for (i = 0; i < n; ++i) {  p[i] = \*c\_str++;  }  return p;  }    /\* ... \*/    char c\_str[] = "hi there";  char \*p = copy(sizeof(c\_str), c\_str); |

|  |  |
| --- | --- |
| **Principles** | |
| **1 - Validate Input Data**  **2 - Heed Compiler Warnings**  **10 - Adopt a Secure Coding Standard** | Validating input is always a must. If it can come from a user, it must be validated. Compiler warnings will often appear when the size of a variable might be an issue, so adopting secure standards to follow these rules is mandatory. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT01 | Static analysis, detection of bad code and architecture analysis |
| CodeSonar | 7.4p0 | LANG.TYPE.BASIC | Basic numeral type used |
| Compass/ROSE |  |  | Can detect violations of this recommendation. In particular, it catches comparisons and operations where one operand is of type size\_t or rsize\_t and the other is not |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Declare function parameters that are pointers to values not changed by the function as const** |
| --- | --- | --- |
| **Data Value** | [STD-002-C] | Declaring function parameters const indicates that the function promises not to change these values.  In C, function arguments are passed by value rather than by reference. Although a function may change the values passed in, these changed values are discarded once the function returns. For this reason, many programmers assume a function will not change its arguments and that declaring the function's parameters as const is unnecessary |

| **Noncompliant Code** |
| --- |
| Unlike passed-by-value arguments and pointers, pointed-to values are a concern. A function may modify a value referenced by a pointer argument, leading to a side effect that persists even after the function exits. Modification of the pointed-to value is not diagnosed by the compiler, which assumes this behavior was intended. |
| void foo(int \*x) {  if (x != NULL) {  \*x = 3; /\* Visible outside function \*/  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution addresses the const violation by not modifying the constant argument: |
| void foo(const int \* x) {  if (x != NULL) {  printf("Value is %d\n", \*x);  }  /\* ... \*/  } |

|  |  |
| --- | --- |
| **Principles** | |
| **9 - Use Effective Quality Assurance Techniques** | Input to a function should also be considered. If the value shouldn’t change, ensure that it is passed by reference or as a constant. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **parameter-missing-const** | Domain aware static analyzer |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | CertC-INT01 | Static analysis, detection of bad code and architecture analysis |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. C-style strings require a null character to indicate the end of the string, while the C++ std::basic\_string template requires no such character. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies** | Using variables of the correct size or ability to change capacity depending on input helps to prevent buffer overflow exception. If designed and built properly, threats like these are greatly reduced. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **parameter-missing-const** | Domain aware static analyzer |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | CertC-INT01 | Static analysis, detection of bad code and architecture analysis |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule STR31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulestr31c.html) | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |

#### Coding Standard 4

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

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() [Viega 2003]: |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |

|  |  |
| --- | --- |
| **Principles** | |
| **7 - Sanitize Data Sent to Other Systems** | When supplying data to other systems, sanitize the data so that it will not throw an exception, thus creating vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protect against command injection Protect against file name injection Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Clear sensitive information stored in reusable resources** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C] | Sensitive data stored in reusable resources may be inadvertently leaked to a less privileged user or attacker if not properly cleared. Examples of reusable resources include  Dynamically allocated memory  Statically allocated memory  Automatically allocated (stack) memory  Memory caches  Disk  Disk caches  The manner in which sensitive information can be properly cleared varies depending on the resource type and platform. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, sensitive information stored in the dynamically allocated memory referenced by secret is copied to the dynamically allocated buffer, new\_secret, which is processed and eventually deallocated by a call to free(). Because the memory is not cleared, it may be reallocated to another section of the program where the information stored in new\_secret may be unintentionally leaked. |
| char \*secret;  /\* Initialize secret to a null-terminated byte string,  of less than SIZE\_MAX chars \*/    size\_t size = strlen(secret);  char \*new\_secret;  new\_secret = (char \*)malloc(size+1);  if (!new\_secret) {  /\* Handle error \*/  }  strcpy(new\_secret, secret);    /\* Process new\_secret... \*/    free(new\_secret);  new\_secret = NULL; |

| **Compliant Code** |
| --- |
| To prevent information leakage, dynamic memory containing sensitive information should be sanitized before being freed. Sanitization is commonly accomplished by clearing the allocated space (that is, filling the space with '\0' characters). |
| char \*secret;  /\* Initialize secret to a null-terminated byte string,  of less than SIZE\_MAX chars \*/    size\_t size = strlen(secret);  char \*new\_secret;  /\* Use calloc() to zero-out allocated space \*/  new\_secret = (char \*)calloc(size+1, sizeof(char));  if (!new\_secret) {  /\* Handle error \*/  }  strcpy(new\_secret, secret);    /\* Process new\_secret... \*/    /\* Sanitize memory \*/  memset\_s(new\_secret, '\0', size);  free(new\_secret);  new\_secret = NULL; |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies**  **Adopt a Secure Coding Standard** | Clearing data once out of scope is just good practice. Preventing memory leaks is important for a secure system |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **(customization)** | Users can add a custom check for use of realloc(). |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-MEM03-a** | Sensitive data should be cleared before being deallocated |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Because assert() calls abort(), cleanup functions registered with atexit() are not called. If the intention of the programmer is to properly clean up in the case of a failed assertion, then runtime assertions should be replaced with static assertions where possible. (See DCL03-C. Use a static assertion to test the value of a constant expression.) When the assertion is based on runtime data, the assert should be replaced with a runtime check that implements the adopted error strategy (see ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy). |

| **Noncompliant Code** |
| --- |
| This noncompliant code example defines a function that is called before the program exits to clean up: |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }    int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }    /\* ... \*/    assert(/\* Something bad didn't happen \*/);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to assert() is replaced with an if statement that calls exit() to ensure that the proper termination routines are run: |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }    int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }    /\* ... \*/    if (/\* Something bad happened \*/) {  exit(EXIT\_FAILURE);  }    /\* ... \*/  } |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies** | Proper architecture and test design will solve the problem of the correct assert() function usage. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 44 S | Enhanced enforcement |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 586 | Fully Supported |

#### 

#### Coding Standard 7

| **Coding Standard** | **Label** | **Exception objects must be nothrow copy constructible** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | The C++ Standard allows the copy constructor to be elided when initializing the exception object to perform the initialization if a temporary is thrown. Many modern compiler implementations make use of both optimization techniques. However, the copy constructor for an exception object still must not throw an exception because compilers are not required to elide the copy constructor call in all situations, and common implementations of std::exception\_ptr will call a copy constructor even if it can be elided from a throw expression. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an exception of type S is thrown in f(). However, because S has a std::string data member, and the copy constructor for std::string is not declared noexcept, the implicitly-defined copy constructor for S is also not declared to be noexcept. In low-memory situations, the copy constructor for std::string may be unable to allocate sufficient memory to complete the copy operation, resulting in a std::bad\_alloc exception being thrown. |
| #include <exception>  #include <string>    class S : public std::exception {  std::string m;  public:  S(const char \*msg) : m(msg) {}    const char \*what() const noexcept override {  return m.c\_str();  }  };    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the type of the exception object can inherit from std::runtime\_error, or that type can be used directly. Unlike std::string, a std::runtime\_error object is required to correctly handle an arbitrary-length error message that is exception safe and guarantees the copy constructor will not throw [ ISO/IEC 14882-2014 ]. |
| #include <stdexcept>  #include <type\_traits>    struct S : std::runtime\_error {  S(const char \*msg) : std::runtime\_error(msg) {}  };    static\_assert(std::is\_nothrow\_copy\_constructible<S>::value,  "S must be nothrow copy constructible");    void g() {  // If some condition doesn't hold...  throw S("Condition did not hold");  }    void f() {  try {  g();  } catch (S &s) {  // Handle error  }  } |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies** | Proper architecture and test design will solve the problem of exception handling. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-err60-cpp | Checked by clang-tidy |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR60-a** **CERT\_CPP-ERR60-b** | Exception objects must be nothrow copy constructible An explicitly declared copy constructor for a class that inherits from 'std::exception' should have a non-throwing exception specification |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not access an object outside of its lifetime** |
| --- | --- | --- |
| **Expressions** | [STD-008-CPP] | Every object has a lifetime in which it can be used in a well-defined manner. The lifetime of an object begins when sufficient, properly aligned storage has been obtained for it and its initialization is complete. The lifetime of an object ends when a nontrivial destructor, if any, is called for the object and the storage for the object has been reused or released. Use of an object, or a pointer to an object, outside of its lifetime frequently results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a pointer to an object is used to call a non-static member function of the object prior to the beginning of the pointer's lifetime, resulting in undefined behavior. |
| struct S {  void mem\_fn();  };    void f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, storage is obtained for the pointer prior to calling S::mem\_fn(). |
| struct S {  void mem\_fn();  };    void f() {  S \*s = new S;  s->mem\_fn();  delete s;  } |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies** | Proper architecture and design will solve the problem of expressions and objects used beyond their lifetime. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **return-reference-local dangling\_pointer\_use** | Partially checked |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **return-reference-local** | Partially checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STD-009-C] | At compile time, string literals are used to create an array of static storage duration of sufficient length to contain the character sequence and a terminating null character. String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is undefined if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. (See undefined behavior 33.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| char str[] = "string literal";  str[0] = 'S'; |

|  |  |
| --- | --- |
| **Principles** | |
| **Architect and Design for Security Policies**  **Heed Compiler Warnings**  **Use Effective Quality Assurance Techniques** | Proper architecture and test design will ensure attempts to alter string literals does not occur. If a warning shows this might be an issue, it should be addressed and using a good approach to quality through static testing and unit testing will help catch errors like this in code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR30** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR30-a** **CERT\_C-STR30-b** | A string literal shall not be modified Do not modify string literals |

Coding Standard 10

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **Input / Output** | [STD-010-C] | Never call a formatted I/O function with a format string containing a tainted value . An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [Seacord 2013b]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  fprintf(stderr, msg);  free(msg);  } |

| **Compliant Code** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents: |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  fputs(msg, stderr);  free(msg);  } |

|  |  |
| --- | --- |
| **Principles** | |
| **3 - Architect and Design for Security Policies**  **1 - Validate Input Data** | When using formatted code, do not use input data. Excluding input data from users is the easiest way to avoid this issue, but in a case where input data must be used, be sure to sanitize and validate thoroughly. If possible, use another method for formatted input/output if user data is required. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **IO.INJ.FMT MISC.FMT** | Format string injection Format string |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **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 |

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.

### 

### Automation



**Planning and Assessment**

Analyze and familiarize the team with any current trends in security as well as currently known methods for breaches. Look at how much defense in depth will be needed to provide enough redundancy for threat levels. Analyze threat levels and create a mitigation plan to incorporate into the code.

**Design**

When designing the application along with a testing plan, create an automation plan for static testing to ensure any potential related threats are covered at testing time. Create a policy for handling the security testing

**Build**

Follow the policy document when building the application, ensuring that best practices are used for known vulnerabilities.

**Verify and Test**

During testing do static analysis using automation. Choose an automation application for the scope of code and follow all warnings and vulnerability notes to ensure these are fixed in code.

**Transition and Health Check**

Set up test to simulate hacker attacks to do penetration testing to ensure the measures in place are working as expected and identify any improvements that need to be made before deployment.

**Monitor and Detect**

Because no software is 100% free of vulnerabilities, it will be necessary to implement measures to monitor the system and detect any breaches in real time as much as possible. Where it isn’t possible to do real time detection, mitigate this by creating a schedule to review logs to this end.

**Respond**

Take appropriate actions to correct any code that has been compromised or cracked. Fix any vulnerabilities and update the policy for any new methodologies. Plan to report any breaches to those concerned for transparency.

**Maintain and Stabilize**

Reset security baseline and keep code free of any known vulnerabilities, taking care to fix any breaches and stabilize code to be protected from any successful attacks.

### 

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-C | Medium | Probable | Medium | P8 | L2 |
| [STD-002-C] | Low | Unlikely | Low |  | P3 |
| [STD-003-CPP] | High | Likely | Medium | P18 |  |
| [STD-004-C] | High | Likely | Medium | P18 | L1 |
| [STD-005-C] | Medium | Unlikely | High | P2 | L3 |
| [STD-006-C] | Medium | Unlikely | Medium | P4 | L3 |
| [STD-007-CPP] | Low | Probable | Medium | P4 | L3 |
| [STD-008-CPP] | High | Probable | High |  | P6 |
| [STD-009-C] | Low | Likely | Low |  | P9 |
| [STD-010-C] | High | Likely | Medium | P18 |  |
| [STD-002-C] | Medium | Probable | Medium | P8 | L2 |
| [STD-003-CPP] | Low | Unlikely | Low |  | P3 |
| [STD-004-C] | High | Likely | Medium | P18 |  |
| [STD-005-C] | High | Likely | Medium | P18 | L1 |

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encrypt data on the hard drive because no approach to security is 100% effective. If hackers to get int to the system, encrypting data locally will be a final level of protection. |
| Encryption at flight | Encrypt all data when it is being sent over the network. If bad actors intercept packets of data during transfer that is not encrypted they will have access to personal data. |
| Encryption in use | Encrypt any data used internally and sent over https or other protocols. Encrypt prior to transport so that any |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is any sort of authorization where a user is required to provide unique credentials. Requiring authentication will allow tracking of activity to one individual. Ensuring authentication will help prevent attacks. |
| Authorization | Authorization refers to keeping the permissions to the least possible permissions to keep the users in a data space that their role requires, but no more to prevent security breaches. |
| Accounting | When a user is authenticated, this will allow tracking in the system. Having a trail of the user activities allow tracing to where data breaches happen and allow investigations to be more thorough in the event of a data security issue. This also helps to keep users accountable and therefore less likely to take part in any suspicious practices. |

## 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 | 08/05/2023 | Initial Template | Shayne Rushton |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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