**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 | Input data validation is an important topic when it comes to c++ security. Invalid or malicious input data can lead to security breaches and vulnerabilities. For example, in some instances when OR x=x is amended to an sql request, it can cause the database to return private data or information that should not be available. Its important to always validate data input in systems that can become compromised or used for illicit purposes. |
| 1. Heed Compiler Warnings | Compiler warnings should be taken with concern and fixed as soon as possible. The compiler is an important part of a c++ program and can help the writer catch errors at compile and runtime. Fixing these issues can improve performance as well as limit injection vectors within the program. |
| 1. Architect and Design for Security Policies | A few specific rules for architect and design for security policies include designing a system with security in mind. Any given subsystem with a program should only have the authority to carry out its intended purpose. By limiting the scope of systems, security designers can protect the system from users with malicious intent and modularize the security of the system as a whole. This is important because it makes software more secure and increases the resiliency of the program. |
| 1. Keep It Simple | When thinking about security when designing software its important to make smart additions in the right place. Instead of having some sort of error handling for each element in a system, leading to writing more and messier code, one can use tests to see if a portion of code is susceptible to injection or interference and make sufficient error handling for the possible cases. Some things may not need error handling at all, but hey should still validate data they produce to eliminate the chance of injection at currently known vectors. |
| 1. Default Deny | Its important new users and roles are granted minimal security privileges, typically only for data the absolutely need or systems the need to utilize. One of the first layers of defense any good programmer knows is protecting the user from themselves. Their privileges should be clearly defined and systems should have a set of roles with varying levels of privileges. By default, the system should deny the user access if they don’t have the required clearance. |
| 1. Adhere to the Principle of Least Privilege | Users should have the least privilege possible in order to interact with the software they are using. This reduces the chances of attackers gaining access to critical systems and sensitive data by compromising low level user accounts. It can also reduce the risk of authorized users making mistakes with sensitive data. A user should only be granted necessary privileges to complete their job. This means setting a default minimum access to low level users, and providing logging an tracking for high level users to ensure no nefarious activity is going on. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems brings about multiple benefits for the security of the system as a whole. System intercommunication is one way attackers can gain unauthorized access to a system or run custom queries for data. The system sending the data should make sure there are no injection points in data sent and validate the security vulnerability potentials in sent data. There are some known issues that should be accounted for and checked in sent data to make sure the security of the system remains intact. |
| 1. Practice Defense in Depth | DiD is another important topic that relies on building multiple layers of security, sometimes redundant, so if one layer fails another can handle whatever security functionality was provided. Even if attackers were to breach one level of security, they will find they still need to break through more in order to gain access to the system. This can dissuade attackers from trying to break further into the system. These layers can also be used to identify the conditions of when one broke down, so developers can fix it and protect known areas of vulnerability. |
| 1. Use Effective Quality Assurance Techniques | Using effective quality assurance techniques is important because they can catch many errors and vulnerabilities before the system is published for official use. While some of these findings can slow down development if they are difficult to fix, they often come with the benefit of knowing areas of a system could be improved and should be in order to protect the data and security of users. Communication between the development team and QA allows both parties to complete their jobs faster and increases awareness of important vulnerabilities or bugs in the software. |
| 1. Adopt a Secure Coding Standard | Adopting a secure coding standard can help the software by making it easier to read and provide a baseline quality for all developers working on it. By adhering to these guidelines, developers will find its easier to read code written by others and having properly documented code throughout the entire program can improve a developer overall understanding of the system. Maintaining these standards will result in a program that is easily changeable and straightforward. |

### 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** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **Data Type** | STR30-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. |

| **Noncompliant Code** |
| --- |
| The char pointer \*str is initialized with a value of a string literal, attempting to modify the string will result in 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. 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'; |

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

| **Principles(s):** Architect and design for security principles, adopt a secure coding standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | string-literal-modification  write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully Implemented |
| Splint | 3.1.1 |  |  |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not pass a non-null-terminated character sequence to a library function that expects a string** |
| --- | --- | --- |
| **Data Value** | STR32-C | Many library functions expect null terminated string arguments. Passing a character sequence or wide character sequence that is not null-terminated can result in accessing memory that is outside the bounds of the object. This can lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| C\_str is not null terminated when passed as an argument to printf and may result in undefined behavior. |
| #include <stdio.h>    void func(void) {  char c\_str[3] = "abc";  printf("%s\n", c\_str);  } |

| **Compliant Code** |
| --- |
| This code block properly initializes a char array without bounds, prompting storage for the entire string as well as a null terminator |
| **void** func(**void**) {  **char** c\_str[] = "abc";  **printf**("%s\n", c\_str);  } |

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

| **Principles(s):** Sanitize data from other systems, adopt a secure coding standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | [Insert text.] | Supported |
| Coverity | 2017.07 | STRING\_NULL | Fully Implemented |
| Helix QAC | 2024.1 | DF2835,DF2836,DF2839 |  |
| PVS-Studio | 7.31 | V692 |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Range check element access** |
| --- | --- | --- |
| **String Correctness** | STR53-CPP | The std::string index operators const\_reference operator[](size\_type) const and reference operator[](size\_type) return the character stored at the specified position, pos. When pos >= size(), a reference to an object of type charT with value charT() is returned. The index operators are unchecked (no exceptions are thrown for range errors), and attempting to modify the resulting out-of-range object results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    **extern** std::**size\_t** get\_index();    **void** f() {    std::string s("01234567");    s[get\_index()] = '1';  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  extern std::size\_t get\_index();    void f() {  std::string s("01234567");  try {  s.at(get\_index()) = '1';  } catch (std::out\_of\_range &) {  // Handle error  }  } |

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

| **Principles(s):** Adopt a secure coding standard |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Assert\_failure |  |
| Helix QAC | 2024.1 | C++3162  C++3163  C++3164  C++3165 |  |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer Overrun  Buffer Underrun  Tainted Buffer access  Type overrun  Type underrun |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53a | Guarantee that container indices are within the valid range |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude User input from format strings** |
| --- | --- | --- |
| **SQL Injection** | FIO32-C | Formatted strings are sometimes used to run operations in different systems. Attackers can use these vectors to cause errors and inject their own instructions. |

| **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. |
| #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);  } |

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

| **Principles(s):** Adopt a secure coding standard, validate input data |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported via stubbing/taint analysis |
| Splint | 3.1.1 |  |  |
| PVS-Studio | 7.3.1 | V618 |  |
| LDRA tool suite | 9.7.1 | 86 D | Partially implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | Mem50-CPP | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required |
| #include <new>  struct S {  void f();  };  void g() noexcept(false) {  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):** Adopt a secure coding standard, practice defense in depth |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft Insure++ |  |  | Runtime deection |
| Splint | 5.0 |  |  |
| PVS-Studio | 7.31 | V586,V774 |  |
| Astree | 22.10 | Dangling\_pointer\_use |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Avoid side effects in arguments to unsafe macros** |
| --- | --- | --- |
| **Assertions** | PRE31-C | An unsafe function-like macro is one whose expansion results in evaluating one of its parameters more than once or not at all. Never invoke an unsafe macro with arguments containing an assignment, increment, decrement, volatile access, input/output, or other expressions with side effects (including function calls, which may cause side effects). |

| **Noncompliant Code** |
| --- |
| This noncompliant code example includes an assert() macro containing an expression (index++) that has a side effect: |
| #include <assert.h>  #include <stddef.h>    **void** process(**size\_t** index) {  **assert**(index++ > 0); /\* Side effect \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution avoids the possibility of side effects in assertions by moving the expression containing the side effect outside of the assert() macro. |
| #include <assert.h>  #include <stddef.h>    **void** process(**size\_t** index) {  **assert**(index > 0); /\* No side effect \*/    ++index;    /\* ... \*/  } |

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

| **Principles(s):** Secure coding standard, architect and design for security policies |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CC2.EXP31  CC2.PRE31 | Fully implemented |
| Klocwork | 9.7.1 | 9S, 562,572 S, D 1Q | Fully implemented |
| PC-lint Plus | 1.4 | 666, 2666 | Fully supported |
| RuleChecker | 24.04 | expanded-side-effect-multiplied  expanded-side-effect-not-evaluated  side-effect-not-expanded | Partially checked |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP | All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| This code demonstrates an exception being used but no handle created to catch the exception thrown |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| This code uses an exception and includes a hander |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

| **Principles(s):** adopt a secure coding standard, defense in layers |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all  Early catch all | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable catch |
| Parasoft C/C++Test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| RuleChecker | 22.10 | Main-function-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not access an object outside of its lifetime** |
| --- | --- | --- |
| **Access Object** | EXP54-CPP | Every object has a lifetime where it can be utilized in a definitive manner. The lifetime begins when storage has been allocated and initialization is complete. The lifetime ends when a nontrivial destructor, if any, is called for the storage of the object to be reused or released. Use of an object, or a pointer to an object outside of its lifetime can result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| S has not been initialized, meaning its lifetime has not yet started when we attempt to call a function from a struct, will result in undefined behavior. |
| struct S {  void mem\_fn();  };    void f() {  S \*s;  s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| S is initialized, beginning its lifetime when we call the function mem\_fn() it, this will not result in undefined behavior |
| struct S {  void mem\_fn();  };    void f() {  S \*s = new S;  s->mem\_fn();  delete s;  } |

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

| **Principles(s):** Keep it simple, practice defense in depth |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Return-reference-local  Dangling\_pointer\_use | Partially checked |
| CodeSonar | 8.1p0 | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Parasoft Insure++ |  |  | Runtime detection |
| RuleChecker | 22.10 | Return reference local | Partially checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not declare or define a reserved identifier** |
| --- | --- | --- |
| **Declarations** | DCL51-CPP | Do not declare or define a reserved identifier. These identifiers include override, final, alignas, carries\_deendency, deprecated, and no return. No other identifiers are reserved Declaring or defining an identifier in a context in which it is reserved results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| Using macro preprocesses that guard against additional header inclusion is a common practice, but when used with an underscore immediately followed by a capital letter means the name is reserved to the implmenetation for use in the global namespace. |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_  // contents of <my\_header.h>  #endif //\_My\_Header\_H\_ |

| **Compliant Code** |
| --- |
| The solution is to avoid using trailing underscores in the name of the header guard. |
| #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H    // Contents of <my\_header.h>    #endif // MY\_HEADER\_H |

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

| **Principles(s):** Adopt a secure coding standard, heed compiler warnings |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Reserved\_identifier | Partially checked |
| PVS-Studio | 7.31 | V1059 |  |
| RuleChecker | 22.10 | Reserved-identifier | Partially checked |
| CodeSonar | 8.1p0 | LANG.ID.NU.MK  LANG.STRUCT.DEL.RESERVED | Macro name is c Keyword  Declaration of reserved name |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Detect and handle memory allocation errors** |
| --- | --- | --- |
| Memory | MEM52-CPP | The default memory allocation operator, ::operator new(std::size\_t), throws a std::bad\_alloc exception if the allocation fails. Therefore, you need not check whether calling ::operator new(std::size\_t) results in nullptr. The nonthrowing form, ::operator new(std::size\_t, const std::nothrow\_t &), does not throw an exception if the allocation fails but instead returns nullptr. The same behaviors apply for the operator new[] versions of both allocation functions. Additionally, the default allocator object (std::allocator) uses ::operator new(std::size\_t) to perform allocations and should be treated similarly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an array of int is created using ::operator new[](std::size\_t) and the results of the allocation are not checked. The function is marked as noexcept, so the caller assumes this function does not throw any exceptions. Because ::operator new[](std::size\_t) can throw an exception if the allocation fails, it could lead to abnormal termination of the program. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| When using std::nothrow, the new operator returns either a null pointer or a pointer to the allocated space. Always test the returned pointer to ensure it is not nullptr before referencing the pointer. This compliant solution handles the error condition appropriately when the returned pointer is nullptr. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

| **Principles(s):** Defense in depth, design and architect for security principals |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  |  |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled. |
| LDRA tool suit | 9.7.1 | 45 D | Partially implemented |
| Parasoft insure++ |  |  | Runtime detection |

### 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 will be used to detect coding errors and breach of standards as well as in unit testing and security testing. We should first verify and test with vulnerability scanning and functional compliancy and security testing. It’ s important to establish a strong foundation, so making sure the libraries the system will depend on are bug free and up to date with standards Is necessary. Then we can rebuild the system and make sure it is securely created inside a trusted repository. We then can augment and design more security practices like test-driven design, OWASP, and keep to security standards. At this point the security of the system has been established, and we can assess and plan our next steps as we go into production with regards to security. Once the system is released, we can do some penetration testing and try to break the security while fixing errors or adding additional functionality. If an attack occurs, we should have the ability to block or prevent access to the systems and its services, or rollback to an earlier version. Until we have plans to release a new version, we will monitor and detect intrusion attempts and configure and deploy further penetration testing to ensure the product is as secure as possible.

### 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 |
| --- | --- | --- | --- | --- | --- |
| STR30-C | Low | Likely | Low | P9 | L2 |
| STR32-C | High | Probable | Medium | P12 | L1 |
| STR53-CPP | High | Unlikely | Medium | P6 | L2 |
| DCL51-CPP | Low | Unlikely | Low | P3 | L3 |
| FIO32-C | High | Likely | Medium | P18 | L1 |
| EXP54-CPP | High | Probable | High | P6 | L2 |
| MEM52-CPP | High | Likely | Medium | P18 | L1 |
| Mem50-CPP | High | Likely | Medium | P18 | L1 |
| ERR51-CPP | High | Likely | Medium | P18 | L1 |
| PRE31-C | Low | Unlikely | Low | P3 | L3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to encrypting data that is inactive or not moving between devices and networks. This information tends to be stored in an archive, which makes it les vulnerable than data that is moving between systems. It should be secured with security software, firewalls, and data loss prevention solutions along with encryption. |
| Encryption in flight | Encryption in flight refers to data that is moving from one location to another. Some examples include email, Microsoft teams, instant messengers like WhatsApp. This data is exposed across the internet or private network, so its less secure and prone to attack. A few ways to protect this data include identify vulnerabilities early, define the framework for security, and implement the necessary technologies for protection. |
| Encryption in use | Encryption in use refers to encrypting data that is accessed or consumed by an application. Data at this state is at its most vulnerable because it is susceptible to attack by the user. Its important to protect information like this and implement additional security measures like authentication and permissions. Some ways we can protect this data is to ensure protection software is functional and up to date. Authentication must be required by the system and strong passwords should be encouraged. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication includes checking the authenticity of credentials supplied by a user. This can be one to many different credentials or pieces of information the user should have. The system compares the credentials submitted to the list of saved encrypted credentials on the server. If one matches each necessary criteria, it will be granted authentication to send and receive requests or other functions. |
| Authorization | After the user has been authenticated and they attempt to make a request, depending on the request they will have to have sufficient authorization or permission for the system to comply. This adds another layer of protection by securing data behind select permissions that can only be granted by a system administrator. Different levels of permissions can be made and customized based on the functionality that should be available for that user. |
| Accounting | Finally, the last layer of defense added includes an accounting of all actions that take place on the server. This increases security by providing a log that security experts can check and automate notifications for suspicious behavior. This is a necessary step because it can be used to give advanced notice of security vulnerabilities and detect attacks. |

**\***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 | 6/1/2024 | Added coding standards and examples | Grant Sorenson | Grant Sorenson |
| 3.0 | 6/20/2024 | Added risk assessment and automation overview | Grant Sorenson | Grant Sorenson |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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