**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 validation is when you take a user’s input to verify its authenticity to prevent unexpected attacks on your system such as SQL injection. Specifically, you want to make sure the data “is data that is anticipated by the design and implementation of the system and therefore will not result in the system entering an indeterminate state” (Seacord, R. C. 2013). |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws (Seacord, 2018). |
| 1. Architect and Design for Security Policies | Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set (Seacord, 2018). |
| 1. Keep It Simple | Keep the design as simple and small as possible (Saltzer, 1974; Saltzer & Schroeder, 1975). Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex (Seacord, 2018). |
| 1. Default Deny | Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted (Saltzer, 1974; Saltzer & Schroeder, 1975). |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges (Saltzer, 1974; Saltzer & Schroeder, 1975). |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem (Seacord, 2018). |
| 1. Practice Defense in Depth | Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment (Seacord, R. C. 2013). |
| 1. Use Effective Quality Assurance Techniques | Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions (Seacord, R. C. 2013). |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language and platform (Seacord, 2018). |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | INT30-C | [**Ensure that unsigned integer operations do not wrap**](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)  One must not allow integers to wrap due a possible vulnerability in exposing memory that a user should not have access to. |
|  |  |  |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap. |
| #include <limits.h>    void func(unsigned int ui\_a, unsigned int ui\_b) {    unsigned int usum;    if (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } else {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** **1)** Validate Input Data to prevent an unsigned int from occurring via the UNIT\_MAX. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| INT30-C | Likely | high | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | integer-overflow | Fully Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | implemented |
| Parasoft C/C++test | 7.1p | **CERT\_C-INT30-a**  **CERT\_C-INT30-b**  **CERT\_C-INT30-c** | * Avoid integer overflows * Integer overflow or underflow in constant expression in '+', '-', '\*' operator * Integer overflow or underflow in constant expression in '<<' operator |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STR52-CPP | [Use valid references, pointers, and iterators to reference elements of a basic\_string](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR52-CPP.+Use+valid+references%2C+pointers%2C+and+iterators+to+reference+elements+of+a+basic_string)  If one were to use an invalid reference, pointer, or iterator it will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example copies input into a std::string, replacing semicolon (;) characters with spaces. This example is noncompliant because the iterator loc is invalidated after the first call to insert(). The behavior of subsequent calls to insert() is undefined. |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the value of the iterator loc is updated as a result of each call to insert() so that the invalidated iterator is never accessed. The updated iterator is then incremented at the end of the loop. |
| #include <string>    void f(const std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();    for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

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

| **Principles(s):** **9.** Use Effective Quality Assurance Techniques applies because we want to make sure we aren’t releasing code that would allow for it to crash. This can be assisted via third party teams to get a fresh perspective. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | **ALLOC.UAF** | Use after free |
| Parasoft C/C++ Test | 2022.1 | **CERT\_CPP-STR52-a** | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2022b | **CERT C++: STR52-CPP** | Checks for use of invalid string iterator (rule partially covered). |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR53-CPP | [Range check element access](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR53-CPP.+Range+check+element+access)  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):** **1.** Validate input data applys here to make sure we do not pass something that cant be accessed properly. **4.** Keep it simple by making use of simple try catch statements to handle errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | **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 | 2022.1 | **CERT\_CPP-STR53-a** | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2022b | **CERT C++: STR53-CPP** | Checks for: Array access out of bounds Array access with tainted index Pointer dereference with tainted offset Rule partially covered. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-001-CPP] | If you do not sanitize your parameters you can allow malicious SQL code to enter in places that could breach your system. |

| **Noncompliant Code** |
| --- |
| The problem with this approach is that it requires the programmer to identify all dangerous characters and character combinations. This may be difficult or impossible without having a detailed understanding of the program, process, library, or component being called. Additionally, depending on the program environment, there could be many ways of encoding or escaping input that may be interpreted with malicious effect after successfully bypassing blacklist checking (Seacord, 2013). |
| int main(int argc, char \*argv[]) {      static char bad\_chars[] = "/ ;[]<>&\t";      char \* user\_data;      char \* cp; /\* cursor into example string \*/      user\_data = getenv("QUERY\_STRING");      for (cp = user\_data; \*(cp += strcspn(cp, bad\_chars));)        \*cp = '\_';      exit(0);  } |

| **Compliant Code** |
| --- |
| Whitelisting is recommended over blacklisting because, instead of having to trap all unacceptable characters, the programmer only needs to ensure that acceptable characters are identified. As a result, the programmer can be less concerned about which characters an attacker may try in an attempt to bypass security checks (Seacord, 2013). |
| int main(void) {      static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz\      ABCDEFGHIJKLMNOPQRSTUVWXYZ\      1234567890\_-.@";      char \*user\_data; /\* ptr to the environment string \*/      char \*cp; /\* cursor into example string \*/      user\_data = getenv("QUERY\_STRING");      printf("%s\n", user\_data);      for (cp = user\_data; \*(cp += strspn(cp, ok\_chars)); )        \*cp = '\_';      printf("%s\n", user\_data);      exit(0);  } |

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

| **Principles(s):** **1.** Validating Input Data to prevent malicious attacks by users. **6.** Adhere to the principle of least privilege to outright block access to those who have no business having it so if an injection made it through it would be less or not impactful. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | **Tainted String** | Fully implemented |
| Parasoft C/C++ test | 2022.1 | **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** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM51-CPP | [Properly deallocate dynamically allocated resources](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM51-CPP.+Properly+deallocate+dynamically+allocated+resources)  Deallocating a pointer that is not allocated dynamically (including non-dynamic pointers returned from calls to placement new()) is undefined behavior because the pointer value was not obtained by an allocation function. Deallocating a pointer that has already been passed to a deallocation function is undefined behavior because the pointer value no longer points to memory that has been dynamically allocated. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two allocations are attempted within the same try block, and if either fails, the catch handler attempts to free resources that have been allocated. However, because the pointer variables have not been initialized to a known value, a failure to allocate memory for i1 may result in passing ::operator delete() a value (in i2) that was not previously returned by a call to ::operator new(), resulting in undefined behavior. |
| #include <new>    void f() {    int \*i1, \*i2;    try {      i1 = new int;      i2 = new int;    } catch (std::bad\_alloc &) {      delete i1;      delete i2;    }  } |

| **Compliant Code** |
| --- |
| This compliant solution initializes both pointer values to nullptr, which is a valid value to pass to ::operator delete(). |
| #include <new>    void f() {    int \*i1 = nullptr, \*i2 = nullptr;    try {      i1 = new int;      i2 = new int;    } catch (std::bad\_alloc &) {      delete i1;      delete i2;    }  } |

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

| **Principles(s):** **2.** Heed compiler warnings because this is something simple that a compiler will catch and then one can correct without a lot of digging. **9.** Use effective quality assurance techniques to to help eliminate possibilities of undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| Polyspace Bug Finder | R2022b | CERT C++: MEM51-CPP | Checks for: Invalid deletion of pointer Invalid free of pointer Deallocation of previously deallocated pointer Rule partially covered. |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free Always provide empty brackets ([]) for delete when deallocating arrays Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor Properly deallocate dynamically allocated resources |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MSC11-C | [Incorporate diagnostic tests using assertions](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions) |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors. |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char \*)malloc(len + 1);    assert(NULL != dup);      memcpy(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion: |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char\*)malloc(len + 1);    /\* Detect and handle memory allocation error \*/    if (NULL == dup) {        return NULL;    }      memcpy(dup, c\_str, len + 1);    return dup;  } |

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

| **Principles(s):** **8.** Practice defense in depth applies here because you want to make sure you are looking at your systems from all possible ways to before deployment. **10.** Adopting a secure coding standard applies here because this can be a universal solution to handling memory exhaustion across the code base. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++ Test | 2022.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR51-CPP | [Handle all exceptions](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| 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):** **10.** Adopt a secure coding standard to ensure you handle external resources correctly and avoid early termination. **9.** Use effective quality assurance techniques like code audits and coverage to make the code is covered and handled properly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++ Test | 2022.1 | 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 |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Declarations and Initialization | DCL58-CPP | [Do not modify the standard namespaces](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL58-CPP.+Do+not+modify+the+standard+namespaces)  Namespaces introduce new declarative regions for declarations, reducing the likelihood of conflicting identifiers with other declarative regions |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the declaration of x is added to the namespace std, resulting in undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes the intention of the programmer was to place the declaration of x into a namespace to prevent collisions with other global identifiers. Instead of placing the declaration into the namespace std, the declaration is placed into a namespace without a reserved name. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):** **10.** Adopt a secure coding standard to avoid undefined behavior and prevent collisions. **4.** Keep it simple by adhering to a simple rule of place declaration into a namespace without a reserved name. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’ |
| Polyspace Bug Finder | R2022b | CERT C++: DCL-58-CPP | Checks for modification of standard namespaces (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | FIO51-CPP | [Close files when they are no longer needed](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed)  A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

| **Principles(s):** **10.** Adopt a secure coding standard by always closing files properly when no longer in use or needed. **4.** Keep it simple in that you close your files before terminating them. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace bug finder | R2022b | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Characters and Strings | STR50-CPP | [Guarantee that storage for strings has sufficient space for character data and the null terminator](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator)  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** |
| --- |
| This 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;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Code Sonar | 7.1p0 | misc.mem.nterm  lang.mem.bo  lang.mem.to/ | No space for null terminator  Buffer overrun  Type overrun |
| Parasoft C/C++ test | 2022.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2022b | CERT C++: STR50-CPP | 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. |

### 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.

As stated, Green Pace as a well-established DevOps process which makes adapting to DevSecOps a simpler task. One important piece to add will be to establish security as in important from the very beginning in the design plan. This needs to be looked over before code is written to prevent major refactoring of releases before pre-production even. As the code is developed it will then need to have steps to verify that security measures that were designed and planned were implemented correctly. Staging the software in a pre-production environment and completing compliance and security testing will Green Pace achieve this. Following this will be a release into production where we continue layers of defense with defense in depth. Utilizing logging tools, penetration testing, and event alerting the live product will be continually watched over and secure. If something were to occur then a response is warranted to adapt and overcome the security flaw. It is important to then follow this up quickly with future predictions and stabilization to keep everything active and secure.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL58-CPP | High | Unlikely | Medium | P6 | L2 |
| ERR51-CPP | Low | Probable | Medium | P4 | L3 |
| FIO51-CPP | Medium | Unlikely | Medium | P4 | L3 |
| INT30-C | High | Likely | High | P9 | L2 |
| MEM51-CPP | High | Likely | Medium | P18 | L1 |
| MSC11-C | Low | Unlikely | High | P1 | L3 |
| STD-001-CPP | High | Probable | Medium | P18 | L1 |
| STR50-CPP | High | Likely | Medium | P18 | L1 |
| STR52-CPP | High | Probable | High | P6 | L2 |
| STR53-CPP | High | Unlikely | Medium | P6 | L2 |

### 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 in rest | Encryption at rest is designed to prevent the attacker from accessing the unencrypted data by ensuring the data is encrypted when on disk. If an attacker obtains a hard drive with encrypted data but not the encryption keys, the attacker must defeat the encryption to read the data (Azure Data 2022). This ties in with our security policy by how we need to think defense in depth. We must take into consideration that even though some data is not being used in the moment it should be encrypted and well protected. We can also look at this from a perspective of the principle of least privilege to prevent unneeded access. |
| Encryption at flight | Encryption in flight is encrypting data that is moving between systems such as servers (*Encryption for data-in-transit* 2022). We will need to consider when moving data between any of our systems that is properly secured in case it were to be intercepted. This will follow along the Sanitize Data Sent to Other Systems standard to make sure we do this correctly by sending data through secure channels as well. |
| Encryption in use | Encryption in use is to protect data that may be in use by the user such as account details. We would want to utilizing hashing to protect and verify the password. This will add another defense layer and will incorporate when they are stored in a database somewhere else and hit that inflight moment as well. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This would be the first step of authenticating the user via the login info they provide. This allows you a first layer of security that makes sure proper authorization to an access is granted (Mylonas, 2018). Authentication is apart of the policy because it allows us to control who the person attempting to access the system is. For instance, security principles Default Deny (5) and Adhere to the Principle of Least Privilege (6) show how maintain who access our systems is a fundamental in maintaining a secure system. |
| Authorization | Authenticating correctly and then storing the passwords salted with no process for decrypting, while allowing for resetting of passwords but never retrieval (Mylonas, 2018). Authorization follows up on authentication in that we want to make if someone is authenticated and can access they are only given access to what they need, based on this authorization. This authorization process is something we do not want to do in any sort of open-source file so encrypting appropriately keeps us in sync with adhering to secure coding standards. |
| Accounting | This mean to keep track what a user did. These can be what resources were accessed, at what time, by who, and what commands were issued (Mylonas, 2018). This allows audits of the system to see did what in case of any bad actors. It is important to remember that security does not stop once a system hits production, in a way, it has just begun. A major part of the DevSecOps cycle is monitoring the network to check for intrusions, but it is as equally as important to monitor what happens right in front of you in case something goes wrong internally. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**References**

Azure Data Encryption-at-rest - azure security. Azure Data Encryption-at-Rest - Azure Security | Microsoft Learn. (2022, August 18). Retrieved from <https://learn.microsoft.com/en-us/azure/security/fundamentals/encryption-atrest>

Encryption for data-in-transit - microsoft service assurance. Microsoft Service Assurance | Microsoft Learn. (2022, September 22). Retrieved from https://learn.microsoft.com/en-us/compliance/assurance/assurance-encryption-in-transit

Mylonas, L. (2018, November 27). What is AAA Security? an introduction to authentication, Authorisation and Accounting. Codebots. Retrieved from <https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting>

Saltzer, J. H. (1974). Protection and the Control of Information Sharing in Multics. Communications of the ACM, 17(7), 388–402. <https://doi-org.ezproxy.snhu.edu/10.1145/361011.361067>

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. Proceedings of the IEEE, Proc. IEEE, 63(9), 1278–1308. <https://doi-org.ezproxy.snhu.edu/10.1109/PROC.1975.9939>

Seacord, R. C. (2013). Secure Coding in C and C++. [MBS Direct]. Retrieved from <https://mbsdirect.vitalsource.com/#/books/9780132981972/>

Seacord, R. (2018). Top 10 secure coding practices. Top 10 Secure Coding Practices - CERT Secure Coding - Confluence. Retrieved from <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCommentId=88044413>