**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)

Data Type [4](#_Toc52464059)

Data Value [5](#_Toc52464060)

String Correctness [6](#_Toc52464061)

SQL Injection [7](#_Toc52464062)

Memory Protection [8](#_Toc52464063)

Assertions [9](#_Toc52464064)

Exceptions [10](#_Toc52464065)

Namespace in a header file [11](#_Toc52464066)

One-definition Rule [13](#_Toc52464067)

Standard Namespaces [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 | Implementing a robust input data validation mechanism is crucial for mitigating potential vulnerabilities within a system. It becomes imperative to thoroughly validate incoming data originating from both untrusted and trusted sources. This meticulous validation process serves as a proactive measure, safeguarding the system against the infiltration of any malicious data that could compromise its integrity and security. By scrutinizing data from diverse sources, organizations can fortify their defenses and establish a resilient barrier against potential threats, thereby fostering a more secure computing environment. |
| 1. Heed Compiler Warnings | Take heed of compiler warnings by diligently compiling and thoroughly testing the code to maintain its optimal efficiency. Strive to keep the code concise and error-free through frequent compilation and testing efforts. While some warnings may seem trivial, they could signify potential issues that range from mere annoyances to serious vulnerabilities. Therefore, it is essential to scrutinize and address all compiler warnings meticulously. This proactive approach ensures that every warning is thoroughly examined, minimizing the risk of overlooking critical issues and contributing to the overall robustness of the codebase. |
| 1. Architect and Design for Security Policies | By integrating security considerations into the design phase, you not only align your code with its intended functionality but also establish a robust foundation that can significantly reduce the time spent on implementing safeguards or addressing vulnerabilities later in the development process. Proactively incorporating security measures during the design stage helps create a resilient framework, ensuring that potential risks are systematically identified and mitigated. This approach not only enhances the overall security posture of your system but also contributes to the efficiency of the development lifecycle by minimizing the need for extensive retrospective adjustments. |
| 1. Keep It Simple | Adhering to the principle of simplicity in your code, maintaining clean and straightforward code not only streamlines the development process but also enhances the overall security of the system. When it comes to security, the clarity and simplicity of your code facilitate a better understanding of its structure and functionality. This, in turn, makes it easier to identify potential vulnerabilities and implement effective protective measures. Striving for simplicity not only contributes to code readability and maintainability but also reinforces the system's security by reducing the complexity that could potentially introduce security risks. |
| 1. Default Deny | "Default Deny" is a core security principle that advocates denying access by default and allowing only explicitly approved actions. This approach minimizes the attack surface, promotes explicit authorization, enhances security posture, enables granular control, and supports effective risk management by focusing on the principle of least privilege. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege (PoLP) is a crucial security principle advocating for the restriction of access rights for users, systems, and processes to the minimum necessary for their legitimate tasks. By limiting privileges to the essential level required, PoLP helps minimize the potential impact of security breaches and reduce the attack surface. This principle ensures that individuals or systems operate with the least amount of authority needed to perform their functions, preventing unnecessary exposure and potential misuse of sensitive information or critical resources, resulting improved overall system security by mitigating the risk of unauthorized access and limiting the potential damage caused by compromised accounts or systems. |
| 1. Sanitize Data Sent to Other Systems | It underscores the importance of thoroughly validating and cleansing data before transmitting it to external systems. This security principle aims to prevent vulnerabilities and potential exploits by ensuring that data shared with other systems is free from malicious elements, such as code injections or other forms of attacks. By implementing rigorous data sanitization practices, organizations can mitigate the risk of unintentionally providing harmful input to external systems, reducing the likelihood of security breaches and data compromise. |
| 1. Practice Defense in Depth | This principle advocates for the implementation of multiple layers of security measures to protect against a variety of threats and involves deploying a combination of technical, procedural, and physical safeguards to create a resilient and comprehensive defense strategy. By diversifying security measures, organizations can mitigate the impact of potential breaches and enhance overall system resilience. Some examples are firewalls, intrusion detection systems, encryption, access controls, and employee training. This principle reminds us that no single security measure is foolproof, and a layered approach is essential to address the evolving landscape of cybersecurity threats. |
| 1. Use Effective Quality Assurance Techniques | It focuses on the integration of robust quality assurance practices throughout the software development lifecycle and recognizes that secure software requires not only functional correctness but also resilience against potential vulnerabilities and security threats. Some examples are code reviews, static and dynamic analysis, comprehensive testing, organizations that help to identify and rectify security issues early in the development process. This proactive approach helps in minimizing the likelihood of deploying insecure code. |
| 1. Adopt a Secure Coding Standard | It emphasizes the importance of implementing and following secure coding practices throughout the software development process, and involves establishing standardized guidelines covering key security aspects, promoting consistency, and proactively addressing vulnerabilities. By adopting and adhering to secure coding standards, organizations enhance the overall security of their software, mitigating potential risks and facilitating early identification and resolution of security issues. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Rule DCL50-CPP in the SEI CERT C++ Coding Standard advises against defining C-style variadic functions in C++ code. Variadic functions, which can accept a variable number of arguments, offer flexibility but can be hazardous. C++ provides alternatives like function parameter packs for achieving similar flexibility with better type safety. Using a C-style ellipsis in a variadic function lacks mechanisms to ensure type safety and argument count, leading to undefined behavior and potential security vulnerabilities. The rule recommends avoiding C-style variadic functions and suggests using safer alternatives like variadic functions with function parameter packs or function currying to achieve similar functionality without compromising type safety and security. Declarations of C-style variadic functions without definitions are permitted in unevaluated contexts. The rule aims to enhance code safety and maintainability in C++ programs. |

| **Noncompliant Code** |
| --- |
| The provided noncompliant code example employs a C-style variadic function to sum a series of integers, reading arguments until the value 0 is encountered. However, calling this function without providing the value 0 as an argument after the first two arguments leads to undefined behavior. Additionally, passing any type other than an int to this function also results in undefined behavior. This lack of type safety and proper argument handling poses risks to the stability and security of the code. |
| #include <cstdarg>  **int** add(**int** first, **int** second, ...) {  **int** r = first + second;  **va\_list** va;  **va\_start**(va, second);  **while** (**int** v = **va\_arg**(va, **int**)) {      r += v;    }  **va\_end**(va);  **return** r;  } |

| **Compliant Code** |
| --- |
| In below provided compliant solution, the add() function is implemented using a variadic function with a function parameter pack, ensuring consistent behavior with the noncompliant example. Unlike the C-style variadic function in the noncompliant code, this compliant approach avoids undefined behavior even if the parameter list lacks termination with 0. Furthermore, if any non-integer values are passed to the function, the code becomes ill-formed, providing a more controlled and safer behavior compared to the undefined behavior in the above noncompliant example. This compliant solution incorporates std::enable\_if to enforce that nonintegral argument values result in an ill-formed program, enhancing type safety and robustness. |
| #include <type\_traits>  **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }  **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...);  } |

| **Principles(s):**  Ensure Input Data Validation  Make Security a Priority by Adopting Secure Coding Standards to Prevent Vulnerabilities Architect and Design with Security Policies to Prevent Vulnerabilities Implement Highly Effective Quality Assurance Techniques for Testing Prioritize Simplicity for Lightweight Code as Best Practice |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | function-ellipsis | static code analyzer, examining the source code of a program without executing it |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.20 | CertC++-DCL50 | various tools for static code analysis, software architecture visualization, and compliance checking. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | LANG.STRUCT.ELLIPSIS | deep analysis techniques that explore different code paths to uncover complex issues. |
| [Parasoft](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-DCL50-a | ensure the quality, reliability, and security of their software |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Rule DCL51-CPP in the SEI CERT C++ Coding Standard advises against declaring or defining a reserved identifier, as specified by the C++ Standard [ISO/IEC 14882-2014]. The rules outlined include restrictions on #defining or #undefining names declared in standard library headers, names lexically identical to keywords or specific identifiers, names containing double underscores or starting with an underscore, and names reserved for the implementation in various contexts such as global namespace, headers, and the Standard C library. Violating these rules, such as declaring an identifier in a reserved context, leads to undefined behavior. It is crucial to adhere to these guidelines to ensure code reliability and compliance with the C++ Standard. |

| **Noncompliant Code** |
| --- |
| Commonly, a macro in a preprocessor conditional guard against multiple header file inclusions. While recommended, some programs use reserved names for these guards, risking clashes with names in the C++ standard template library headers or compiler-defined names, even without including C++ standard library headers. Such conflicts can cause unintended issues, underscoring the need to choose non-reserved names for header guards to ensure compatibility in C++ programs. |
| #ifndef \_MY\_HEADER\_H\_  #define \_MY\_HEADER\_H\_  // Contents of <my\_header.h>  #endif // \_MY\_HEADER\_H\_ |

| **Compliant Code** |
| --- |
| This compliant solution refrains from using leading or trailing underscores in the header guard's name. |
| #ifndef MY\_HEADER\_H  #define MY\_HEADER\_H  // Contents of <my\_header.h>  #endif // MY\_HEADER\_H |

| **Principles(s):** Take notice of compiler warnings, as they serve a purpose and should not be overlooked Prioritize security by adhering to secure coding standards, actively working to prevent vulnerabilities  Design and construct code with security policies in mind to proactively mitigate vulnerabilities Employ quality assurance techniques that maximize the effectiveness of tests Embrace simplicity, striving to keep code lightweight, as it aligns with best practices |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-DCL51-a | Do not declare or define a reserved identifier |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | LANG.STRUCT.DECL.RESERVED | deep analysis techniques that explore different code paths to uncover complex issues. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.26 | [V1059](https://pvs-studio.com/en/docs/warnings/v1059/) | detect software bugs and security vulnerabilities in source code written in C, C++, C#, and Java |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | reserved-identifier | static code analyzer, examining the source code of a program without executing it |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Rule DCL52-CPP in the SEI CERT C++ Coding Standard advises against qualifying a reference type with const or volatile. C++ inherently treats all references as const qualified, and the standard explicitly mentions that cv-qualified references are ill-formed, except when introduced through typedef-name or decltype-specifier, in which case the cv-qualifiers are ignored. Attempting to const-qualify a reference type may lead to unintended issues, such as writing **char &const p;** instead of the correct **char const &p; or const char &p;.** This rule emphasizes not attempting to cv-qualify reference types to avoid undefined behavior, and any conforming compiler is expected to issue a diagnostic message for such cases. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a const-qualified reference to a char is mistakenly created instead of a reference to a const-qualified char. This error leads to undefined behavior. |
| #include <iostream>  **void** f(**char** c) {  **char** &**const** p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the const qualifier has been removed. |
| #include <iostream>  **void** f(**char** c) {  **char** &p = c;    p = 'p';    std::cout << c << std::endl;  } |

| **Principles(s):**  Prioritize the incorporation of security policies in your architectural and design processes to proactively build code that guards against vulnerabilities.  Embrace simplicity, as adhering to the principle of keeping code lightweight is considered a best practice. Pay careful attention to compiler warnings, as they serve a crucial purpose. Make security a top priority by adopting a secure coding standard, which plays a vital role in preventing vulnerabilities. Employ quality assurance techniques that maximize effectiveness when creating tests. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-DCL53-a | Do not write syntactically ambiguous declarations |
| Polyspace Bug Finder | R2023b | CERT C++: DCL53-CPP | Do not write syntactically ambiguous declarations |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -Wvexing-parse | Do not write syntactically ambiguous declarations |
| [PRQA QA-C++](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046345) | 4.4 | 2502, 2510 | Do not write syntactically ambiguous declarations Do not write syntactically ambiguous declarations |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Rule DCL53-CPP in the SEI CERT C++ Coding Standard advises against writing syntactically ambiguous declarations. Such declarations can lead to a vexing parse situation where the compiler may struggle to distinguish between expression statements and declarations. The C++ Standard mentions ambiguity in the grammar involving expression statements and declarations, particularly when function-style explicit type conversions and parentheses are involved. The resolution is to consider any construct that could possibly be a declaration as a declaration. The rule recommends avoiding such ambiguous declarations and suggests using alternatives like uniform initialization syntax, nonfunction-style casts, initializing with '=', or removing extraneous parentheses to disambiguate declarations. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, an attempt is made to create an anonymous local variable of type **std::unique\_lock** with the intention of utilizing RAII to lock and unlock the mutex **m**. However, the declaration is syntactically ambiguous, as it can be interpreted either as declaring an anonymous object and invoking its single-argument converting constructor or as declaring an object named **m** and default constructing it. Unfortunately, the syntax used in this example results in the latter interpretation, and consequently, the mutex object is never locked. |
| #include <mutex>  **static** std::mutex m;  **static** **int** shared\_resource;  **void** increment\_by\_42() {    std::unique\_lock<std::mutex>(m);    shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the lock object is assigned an identifier other than 'm', and the appropriate converting constructor is invoked. This avoids the syntactic ambiguity present in the noncompliant example, ensuring that the intended behavior of locking and unlocking the mutex through RAII is correctly implemented. |
| #include <mutex>  **static** std::mutex m;  **static** **int** shared\_resource;  **void** increment\_by\_42() {    std::unique\_lock<std::mutex> lock(m);    shared\_resource += 42;  } |

| **Principles(s):**  Validate received data as input – ensuring proper inputs from users and other programs.  Architect and design for security policies – constructing code to prevent vulnerabilities.  Embrace simplicity, as keeping code lightweight is a best practice.  Employ effective quality assurance techniques – creating tests that are as effective as possible.  Prioritize security by adopting a secure coding standard to help prevent vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C+ +test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-DCL53-a CERT\_CPP-DCL53-b | Declare functions at the file scope consistently. An identifier declared in a local or function prototype scope should not obscure an identifier declared in a global or namespace scope. |
| [LDRA](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 296 S | Partially implemented |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.1 | [**S3468**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-3468) |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.0p0 | **LANG.STRUCT.DECL.FNEST** | Nested Function Declaration |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Rule DCL54-CPP in the SEI CERT C++ Coding Standard recommends overloading allocation and deallocation functions as a pair in the same scope. If an allocation function is overloaded, the corresponding deallocation function should also be overloaded in the same scope, and vice versa. Failure to do so may lead to violations of other rules, such as MEM51-CPP, which concerns proper deallocation of dynamically allocated resources. For example, if an overloaded allocation function uses a private heap, passing a pointer returned by it to the default deallocation function might result in undefined behavior. Even if the allocation function ultimately uses the default allocator, not overloading the corresponding deallocation function could leave the program in an unexpected state. It is acceptable to define a deleted allocation or deallocation function without its corresponding free store function to prevent accidental allocation or deallocation. However, providing a definition for these deleted functions should be avoided to prevent access within a class member function. |

| **Noncompliant Code** |
| --- |
| Here, an allocation function is overloaded at the global scope. However, the corresponding deallocation function is not declared. If an object is allocated using this overloaded allocation function, any attempt to delete the object would lead to undefined behavior, violating MEM51-CPP, which emphasizes the proper deallocation of dynamically allocated resources. |
| #include <Windows.h>  #include <new>  **void** \*operator **new**(std::**size\_t** size) noexcept(**false**) {  **static** **HANDLE** h = ::HeapCreate(0, 0, 0); // Private, expandable heap.  **if** (h) {  **return** ::HeapAlloc(h, 0, size);    }  **throw** std::bad\_alloc();  }  // No corresponding global delete operator defined. |

| **Compliant Code** |
| --- |
| Here, the corresponding deallocation function is also defined at the global scope to ensure that the allocation and deallocation functions are properly overloaded as a pair. |
| #include <Windows.h>  #include <new>  **class** HeapAllocator {  **static** **HANDLE** h;  **static** **bool** init;  **public**:  **static** **void** \*alloc(std::**size\_t** size) noexcept(**false**) {  **if** (!init) {        h = ::HeapCreate(0, 0, 0); // Private, expandable heap.        init = **true**;      }  **if** (h) {  **return** ::HeapAlloc(h, 0, size);      }  **throw** std::bad\_alloc();    }  **static** **void** dealloc(**void** \*ptr) noexcept {  **if** (h) {        (**void**)::HeapFree(h, 0, ptr);      }    }  };  **HANDLE** HeapAllocator::h = nullptr;  **bool** HeapAllocator::init = **false**;  **void** \*operator **new**(std::**size\_t** size) noexcept(**false**) {  **return** HeapAllocator::alloc(size);  }  **void** operator **delete**(**void** \*ptr) noexcept {  **return** HeapAllocator::dealloc(ptr);  } |

| **Principles(s):**  Validating receiving Data as input– ensuring proper inputs from users and other programs  Use Effective Quality Assurance Techniques – making tests that are as effective as possible Architect and Design for Security Policies – building code to prevent vulnerabilities  Adopt a secure coding standard – making security a priority helps prevent vulnerabilities Keep it simple – always applies as keeping code as lightweight as possible is best practice |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL54-a** | Always provide new and delete together |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: DCL54-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl54cpp.html) | Checks for mismatch between overloaded operator new and operator delete (rule fully covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **new-delete-pairwise** | Partially checked |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | misc-new-delete-overloads | Checked with clang-tidy. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Rule DCL55-CPP in the SEI CERT C++ Coding Standard advises programmers to avoid information leakage when passing a class object across a trust boundary. The C++ Standard specifies that the layout of non-static data members in a non-union class is implementation-defined, and padding bits may be present at any location within a class object instance. Unless zero-initialized, these padding bits may contain indeterminate values, including potentially sensitive information.  When passing a pointer to a class object instance across a trust boundary to a different trusted domain, it is crucial to ensure that the padding bits of the object do not contain sensitive information. This practice helps prevent unintentional information leakage and promotes secure handling of class objects in scenarios involving different trust domains. |

| **Noncompliant Code** |
| --- |
| Here, the **arg** object is value-initialized through direct initialization, which involves zero-initialization. This process initializes padding bits to 0, akin to using **std::memset()** to set all bits in the object to 0. While this mitigates the risk of uninitialized padding bits, it's important to note potential portability issues. Caution is advised when relying on implementation details for zero-initialization, and alternative, more explicit methods should be considered for ensuring data security when passing objects across trust boundaries. |
| #include <cstddef>  **struct** test {  **int** a;  **char** b;  **int** c;  };  // Safely copy bytes to user space  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);  **void** do\_stuff(**void** \*usr\_buf) {    test arg{};    arg.a = 1;    arg.b = 2;    arg.c = 3;    copy\_to\_user(usr\_buf, &arg, **sizeof**(arg));  } |

| **Compliant Code** |
| --- |
| [Padding bits can be explicitly declared as fields within a structure. However, this approach is not portable, relying on implementation and target memory architecture. The provided solution is specific to the x86-32 architecture.  The use of **static\_assert()** is employed to validate assumptions about the absence of additional padding bytes. The compiler evaluates the constant expression at compile time, and if false, compilation terminates with the specified error message.  While explicit insertion of padding bytes aims to prevent additional padding, it is crucial to validate these assumptions for a specific implementation to ensure correctness. This solution is tailored for x86-32, and its effectiveness may vary across different architectures and compilers. |
| #include <cstddef>  **struct** test {  **int** a;  **char** b;  **char** padding\_1, padding\_2, padding\_3;  **int** c;    test(**int** a, **char** b, **int** c) : a(a), b(b),      padding\_1(0), padding\_2(0), padding\_3(0),      c(c) {}  };  // Ensure c is the next byte after the last padding byte.  static\_assert(offsetof(test, c) == offsetof(test, padding\_3) + 1,                "Object contains intermediate padding");  // Ensure there is no trailing padding.  static\_assert(**sizeof**(test) == offsetof(test, c) + **sizeof**(**int**),                "Object contains trailing padding");  // Safely copy bytes to user space.  **extern** **int** copy\_to\_user(**void** \*dest, **void** \*src, std::**size\_t** size);  **void** do\_stuff(**void** \*usr\_buf) {    test arg{1, 2, 3};    copy\_to\_user(usr\_buf, &arg, **sizeof**(arg));  } |

| **Principles(s):**  Ensuring proper inputs through the validation of input data  Keeping data sent to other systems limited to what is required/authorized through data sanitization Building code to prevent vulnerabilities by incorporating security policies into architecture and design Always applying the principle of simplicity, as keeping code lightweight is considered best practice Making security a priority to prevent vulnerabilities by adopting a secure coding standard Making tests as effective as possible through the use of quality assurance techniques |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.0p0 | **MISC.PADDING.POTB** | Padding Passed Across a Trust Boundary |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.3 | **DF4941, DF4942, DF4943** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL55-a** | A pointer to a structure should not be passed to a function that can copy data to the user space |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023b | [CERT C++: DCL55-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl55cpp.html) | Checks for information leakage due to structure padding (rule partially covered) |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Rule ERR55-CPP in the SEI CERT C++ Coding Standard underscores the significance of adhering to exception specifications, particularly in the context of terminating a destructor, operator delete, or operator delete[] by throwing an exception. The standard specifies that such behavior triggers undefined behavior.  For deallocation functions, it is essential not to terminate by throwing an exception. Declaring these functions with **noexcept(false)** is unacceptable. Instead, relying on the implicit **noexcept(true)** specification or explicitly declaring **noexcept** on the function signature is recommended.  In the case of destructors, terminating by throwing an exception can lead to undefined behavior. Destructors must not be declared with **noexcept(false)** . They may rely on the implicit **noexcept(true)** or declare noexcept explicitly.  Furthermore, any **noexcept** function that terminates by throwing an exception violates ERR55-CPP. Adhering to exception specifications is crucial for ensuring well-defined behavior in C++ programs. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the use of **std::uncaught\_exception()** in the destructor aims to address the termination problem by avoiding the propagation of the exception if an existing exception is being processed. However, this approach circumvents normal destructor processing and may prevent the destructor from releasing critical resources. |
| #include <exception>  #include <stdexcept>  **class** S {  **bool** has\_error() **const**;  **public**:    ~S() noexcept(**false**) {      // Normal processing  **if** (has\_error() && !std::uncaught\_exception()) {  **throw** std::logic\_error("Something bad");      }    }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the SomeClass destructor is designed to behave consistently, whether or not there is an active exception. It achieves this by having an explicit return statement, preventing control from reaching the end of the exception handler. As a result, the handler catches exceptions thrown by Bad::~Bad(), including the one when bad\_member is destroyed, as well as any exceptions thrown within the compound statement of the function-try-block. Importantly, the SomeClass destructor does not terminate by throwing an exception, aligning with the recommended behavior. |
| **class** SomeClass {    Bad bad\_member;  **public**:    ~SomeClass()  **try** {      // ...    } **catch**(...) {      // Catch exceptions thrown from noncompliant destructors of      // member objects or base class subobjects.      // NOTE: Flowing off the end of a destructor function-try-block causes      // the caught exception to be implicitly rethrown, but an explicit      // return statement will prevent that from happening.  **return**;    }  }; |

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

| **Principles(s):**  Designing and architecting with security policies in mind, constructing code to thwart vulnerabilities Embracing simplicity as a constant practice, aiming to keep code as lightweight as possible Implementing effective quality assurance techniques to create tests with maximum effectiveness  Prioritizing security by adopting a secure coding standard, thereby aiding in the prevention of vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.3 | **C++1552, C++1554, C++1704** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **6 D** | Enhanced Enforcement |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL56-a** | Avoid initialization order problems across translation units by replacing non-local static objects with local static objects |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023b | [CERT C++: DCL56-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl56cpp.html) | Checks for:   * Recursive initialization of static variables * Undetermined initialization order of global variables   Rule fully covered. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Namespace in a header file | [STD-008-CPP] | Rule DCL59-CPP in the SEI CERT C++ Coding Standard advises against defining an unnamed namespace in a header file. Unnamed namespaces create a namespace unique to the translation unit, with names having internal linkage by default. The C++ Standard [ISO/IEC 14882-2014] specifies that an unnamed-namespace-definition behaves as if it were replaced by certain elements.  Production-quality C++ code often uses header files to share code between translation units. However, defining an unnamed namespace in a header file can lead to unexpected outcomes. Since each translation unit defines its own unique instance of members in the unnamed namespace with default internal linkage, it can result in surprises, increase executable size, or unintentionally trigger undefined behavior due to violations of the one-definition rule (ODR). The guideline advises against this practice to ensure consistent and predictable behavior across translation units. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the variable v is defined in an unnamed namespace within a header file and is accessed from two separate translation units. Each translation unit prints the current value of v and then assigns a new value to it. However, because v is defined within an unnamed namespace, each translation unit operates on its own instance of v, resulting in unexpected output.  When executed, this program prints the following: f(): 0  g(): 0  f(): 42  g(): 100 |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  **namespace** {  **int** v;  }  #endif // A\_HEADER\_FILE  // a.cpp  #include "a.h"  #include <iostream>  **void** f() {    std::cout << "f(): " << v << std::endl;    v = 42;    // ...  }  // b.cpp  #include "a.h"  #include <iostream>  **void** g() {    std::cout << "g(): " << v << std::endl;    v = 100;  }  **int** main() {  **extern** **void** f();    f(); // Prints v, sets it to 42    g(); // Prints v, sets it to 100    f();    g();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, v is defined in only one translation unit but is externally visible to all translation units, resulting in the expected behavior.  When executed, this program prints the following:  f(): 0  g(): 42  f(): 100  g(): 42 |
| // a.h  #ifndef A\_HEADER\_FILE  #define A\_HEADER\_FILE  **extern** **int** v;  #endif // A\_HEADER\_FILE  // a.cpp  #include "a.h"  #include <iostream>  **int** v; // Definition of global variable v  **void** f() {    std::cout << "f(): " << v << std::endl;    v = 42;    // ...  }  // b.cpp  #include "a.h"  #include <iostream>  **void** g() {    std::cout << "g(): " << v << std::endl;    v = 100;  }  **int** main() {  **extern** **void** f();    f(); // Prints v, sets it to 42    g(); // Prints v, sets it to 100    f(); // Prints v, sets it back to 42    g(); // Prints v, sets it back to 100  } |

| **Principles(s):**  Design code to prevent vulnerabilities by incorporating security policies into the architecture and design Create tests with maximum effectiveness by employing quality assurance techniques Prioritize security by adopting a coding standard that ensures secure practices, thereby preventing vulnerabilities Always apply the principle of simplicity, aiming to keep code as lightweight as possible, which is considered best practice |
| --- |

**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/cplusplus/LDRA) | 9.7.1 | **286 S, 512 S** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL59-a** | There shall be no unnamed namespaces in header files |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: DCL59-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl59cpp.html) | Checks for unnamed namespaces in header files (rule fully covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **unnamed-namespace-header** | Fully checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| one-definition rule (ODR) | [STD-009-CPP] | Rule DCL60-CPP in the SEI CERT C++ Coding Standard stresses the importance of adhering to the one-definition rule (ODR) in C++. The ODR ensures deterministic linking by requiring a single definition for each non-inline function or variable ODR-used in a program. While certain entities allow more than one definition under specific conditions, violating the ODR results in undefined behavior. Common practices like using header files with declarations usually don't violate the ODR, but caution is needed with language linkage specifications and vendor-specific extensions to avoid accidental violations that could lead to unpredictable behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two different translation units define a class with the same name, but with differing definitions. While the two definitions are functionally equivalent, as they both define a class named S with a single public non-static data member int a, they are not identical in terms of the sequence of tokens used. This violation of the ODR leads to undefined behavior. |
| // a.cpp  **struct** S {  **int** a;  };  // b.cpp  **class** S {  **public**:  **int** a;  }; |

| **Compliant Code** |
| --- |
| The appropriate mitigation depends on the programmer's intent. If the intention is for the same class definition to be visible in both translation units due to common usage, the compliant solution is to use a header file to introduce the object into both translation units, as demonstrated here. |
| // S.h  **struct** S {  **int** a;  };  // a.cpp  #include "S.h"  // b.cpp  #include "S.h" |

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

| **Principles(s):** Maintaining code integrity ensures the creation of clear, maintainable code, minimizing the potential for security issues during updates. This includes preventing code injection and manipulation, aligning with the overarching secure coding principle of "security through design" to establish a robust foundation less susceptible to security vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.0p0 | **LANG.STRUCT.DEF.FDH** **LANG.STRUCT.DEF.ODH** | Function defined in header file Object defined in header file |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.3 | **C++1067, C++1509, C++1510** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **286 S, 287 S** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL60-a** | A class, union or enum name (including qualification, if any) shall be a unique identifier |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Standard Namespaces | [STD-010-CPP] | Rule DCL58-CPP in the SEI CERT C++ Coding Standard advises against modifying the standard namespaces, specifically std and posix. The C++ Standard [ISO/IEC 14882-2014] defines undefined behavior for introducing new declarations or definitions in these namespaces, except for specific circumstances, such as a template specialization that depends on a user-defined type meeting certain requirements. It is crucial not to add declarations or definitions to std or posix, or to a namespace contained within them, to maintain compatibility with standard library conventions and avoid undefined behavior. |

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

| **Compliant Code** |
| --- |
| In this compliant solution, assuming the programmer intended to prevent collisions with other global identifiers by placing the declaration of x into a namespace, the declaration is put into a namespace without a reserved name. This avoids placing the declaration in the **namespace std**, adhering to the guideline and preventing potential undefined behavior. |
| **namespace** nonstd {  **int** x;  } |

| **Principles(s):**  Design code with security policies in mind to build a foundation that prevents vulnerabilities.  Implement effective quality assurance techniques to create tests with optimal effectiveness.  Embrace simplicity as a consistent practice, aiming to keep code as lightweight as possible, which is considered best practice.  Prioritize security by adopting a secure coding standard, contributing to the prevention of vulnerabilities. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | P6 | L12 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-DCL58-a** | Do not modify the standard namespaces 'std' and 'posix' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023b | [CERT C++: DCL58-CPP](https://www.mathworks.com/help/bugfinder/ref/certcdcl58cpp.html) | Checks for modification of standard namespaces (rule fully covered) |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.0p0 | **LANG.STRUCT.DECL.SNM** | Modification of Standard Namespaces |
| PVS-Studio | 7.28 | **V1061** |  |

### 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.

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Probable | Medium | P12 | L1 |
| STD-002-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-003-CPP | Low | Unlikely | Meduim | P3 | L3 |
| STD-004-CPP | Low | Unlikely | Meduim | P2 | L3 |
| STD-005-CPP | Low | Probable | Low | P6 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Unlikely | Meduim | P3 | L3 |
| STD-008-CPP | Meduim | Unlikely | Meduim | P4 | L3 |
| STD-009-CPP | High | Unlikely | High | **P3** | **L3** |
| STD-0010-CPP | High | Unlikely | Medium | P6 | L12 |

### 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 | Data-at-rest encryption involves securely encoding data during storage writes and decrypting it when retrieved for use. Employing a symmetric encryption key during storage writing safeguards the data from unauthorized access, as only those with the decryption key can retrieve and use the information. This practice is essential for sensitive data that, if accessed by unauthorized entities, could cause harm. |
| Encryption at flight | The process of encrypting data in transit involves securely encoding information as it is being transmitted. The application of encryption depends on the method of data transfer. When using a web browser, prioritize secure protocols, and when sending emails, ensure encryption of the content before transmission, along with the use of digital signatures. |
| Encryption in use | Protecting data in use involves encrypting it as it is utilized in memory. The primary method for achieving this is by employing password-protected profiles. These profiles secure the memory associated with each user, preventing the compromise of data stored in memory, which could otherwise pose a risk to data at rest or in transit. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The authentication process verifies a user's identity through credentials like a username and password. Enhanced security measures, such as secure tokens, two-step verification, PINs, and other hardware credentials, may also be utilized for added authentication security |
| Authorization | After authentication and gaining access to a system, the user is granted specific permissions to various parts of that system, roles, or performing specific actions. System administrators authorize access to specific drives, folders, programs, or data based on the user's privileges. |
| Accounting | Upon successful or unsuccessful completion of authentication and authorization steps, it is recommended to monitor and log the activities of all users on the system. This process, known as accounting, provides a comprehensive understanding of who is attempting to access the system and the specific actions they undertake when granted authorization to the system's data. |

**\***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
  + - A straightforward solution that avoids unnecessary complexity.
    - All accounts are denied access by default until explicitly granted.
    - An integral part of the overall process.
    - Mitigate the risk of accumulating unnecessary privileges.
    - Integral to the entirety but not standalone.
* Firewall logs
  + - A straightforward solution that avoids unnecessary complexity.
    - All accounts are denied access by default until explicitly given permission.
    - Mitigate the risk of accumulating unnecessary privileges.
    - Prevent the unnecessary transmission of data.
    - An integral part of the overall system, not standalone.
    - It is a part of the entire process.
* Anti-malware logs
  + - A straightforward solution that avoids unnecessary complexity.
    - Set a default where all accounts are denied access until explicitly granted.
    - Guard against the accumulation of creeping privileges.
    - Avoid the transmission of unnecessary data.
    - Acknowledge its role as a part of the whole, not standalone.
    - Recognize its integration as a vital component in the overall process.

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.21 | 12/08/2023 | Added coding standards, threat levels and explanations | Mahdi Bigdely | Mahdi Bigdely |
| [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 |