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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

[C/C++ Ten Coding Standards 3](#_3dy6vkm)

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

[Defense-in-Depth Illustration 15](#_2jxsxqh)

[Project One 15](#_z337ya)

[1.](#_3j2qqm3) Revise the C/C++ Standards 15

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

[5.](#_1ci93xb) Summary of Risk Assessments 16

[6.](#_3whwml4) Create Policies for Encryption and Triple A 16

[7.](#_2bn6wsx) Map the Principles 17

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

[Approved C/C++ Language Acronyms 19](#_23ckvvd)

## 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 the process of verifying that any input that comes into the program is well formed and will not cause errors or vulnerabilities. Any application that receives input should assume that the input is malicious and properly validate it before storing it or using it to perform operations. Validation can take many forms, including range checking and character or string whitelisting. However, what is essential is that all inputs into the program are rigorously validated to ensure they are well-formed. |
| 1. Heed Compiler Warnings | Heeding compiler warnings means that developers should always strive to be able to compile their code with no warnings. Compiler warnings are essential to developing secure and correct code and are often neglected or not taken as seriously as they should be. A compiler warning is often the first sign that the programmer has either made a mistake or has created the conditions for a potential error or vulnerability. Common compiler warnings include potential loss of data due to implicit conversions, buffer overrun warnings, and uninitialized variables. All of these warnings can result in exploitable vulnerabilities if not rectified by the programmer. |
| 1. Architect and Design for Security Policies | Architecting and designing for security policies means that developers should incorporate security into the design and implementation of the system from the ground up. This principle highlights the importance of adhering to security policies at every stage of the software development lifecycle (SDLC) in order to architect a secure system. From the requirements gathering to the design, implementation, and testing, each phase should adhere to a strict set of security policies in order to both design and implement secure and maintainable code. Some of the security principles that might be used throughout the SDLC include the principle of least privilege, defense in depth, and separation of duties. Adhering to these principles throughout the entire SDLC is essential for architecting secure systems. |
| 1. Keep It Simple | Keeping it simple in the context of security principles means that developers should always focus on simplicity when implementing systems. Keeping your applications simple by reducing the number of components or only accepting input when necessary helps minimize the application's attack surface. Only essential functionality should be included in the application, as unused components unnecessarily increase the potential attack surface. For example, including a deactivated UI feature with a 3rd party library dependency introduces possible vulnerabilities and attack vectors for no reason; instead, this feature should be removed to simplify and thus increase the application's security. Additionally, keeping your security mechanisms simple helps to reduce mistakes and makes it easier for developers to understand what aspects of the system are secure and how the mechanisms work. |
| 1. Default Deny | Default deny means that an application's default behavior to an access request should be to deny. Deny by default increases an application's security by ensuring that access control is only granted to allowed entities. This principle can be achieved in practice through the use of whitelists, or allow-lists, that specify what entities can be allowed and deny anything not present in the list. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege specifies that an application should only grant an entity the minimum privilege required to perform its task. Minimizing such privileges helps to reduce the damage an attacker or malicious program could cause. In practice, least-privilege access is generally implemented with role-based access control, where certain roles can be granted to entities to allow them to perform certain tasks |
| 1. Sanitize Data Sent to Other Systems | Data sanitization is the process of ensuring that data sent to other systems is not invalid or malicious. Data that is sent to other systems should be properly sanitized to ensure that it adheres to the requirements of that system. In practice, data sanitization can include removing or escaping certain characters or any other formatting that is required by the receiving system. Ultimately, sanitizing data that is sent across systems helps to prevent errors and reduce the scope of damage a malformed input can cause. |
| 1. Practice Defense in Depth | Applications should use multiple overlapping layers of security, known as a defense in depth, to prevent attackers from exploiting single points of failure. By using multiple redundant layers, an attacker must exploit each layer independently, thereby making the risk of them succeeding in gaining unauthorized access much lower than if only a single layer of defense was used. In practice, defense in depth often looks like employing a firewall to prevent certain kinds of network traffic, granting an authorized user the minimum privileges required for their tasks while also validating any user input to ensure that it is not malicious. The likelihood of an attacker defeating all three of these layers is much lower than if only one of them was employed. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques like testing and code reviews help ensure that applications are free of errors. Coding errors often can result in vulnerabilities. Thus, employing a robust test suite, including black box and white box tests along with code reviews, static analysis tools, and dependency scanners can help identify potential errors or vulnerabilities and give developers a chance to fix them. Additionally, more security-focused quality assurance, like penetration tests, can help further increase the security of the system. |
| 1. Adopt a Secure Coding Standard | A secure coding standard establishes a set of guidelines for developers to adhere to when writing code. One common secure coding standard may be to never allow implicit conversions between data types, and another may be to ensure that integer overflow or wrapping is prevented. By adopting a standard which enforces the use of best practices, developers will write code that is more secure and correct than if they were left on their own, where mistakes and oversights often result in vulnerabilities. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | **Avoid syntactically ambiguous declarations**  Syntactically ambiguous declarations can be understood by the compiler as either expressions or declarations, thus causing unexpected and incorrect behavior. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code below, a local variable of type Entity is declared using ambiguous syntax. The use of parentheses in this example could cause the compiler to treat e as a function pointer which takes no arguments and returns an Entity or as a declaration of a local variable of type Entity . |
| #include <iostream>    **struct** Entity {  Entity () { std::cout << "Constructed" << std::endl; }  };    **void** foo() {  Entity e();  } |

| **Compliant Code** |
| --- |
| The solution below shows two ways to declare an Entity type variable without the declaration being ambiguous. The first way drops the parentheses entirely which ensures that the compiler treats e as a variable declaration. The second method uses curly braces to perform direct initialization of the variable. |
| #include <iostream>    **struct** Entity {  Entity () { std::cout << "Constructed" << std::endl; }  };    **void** foo() {  Entity e1; // no parentheses  Entity e2{}; // direct initialization  } |

| **Principles(s):**  **Principle #2**  This principle states that developers should always heed compiler warnings. In the case of standard 001, most compilers will indeed generate a warning if a syntactically ambiguous declaration occurs, generally in the form of a warning that the empty parentheses are interpreted as a function declaration. Thus, following principle #2 will help support the standard of avoiding syntactically ambiguous declarations.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. Rudimentary testing is not likely to uncover syntactically ambiguous declarations. Thus, following principle #9 by employing quality tests that gain high code coverage will help support the standard of avoiding syntactically ambiguous declarations.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 001 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unikely | Medium | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Visual Studio](https://learn.microsoft.com/en-us/cpp/error-messages/compiler-warnings/compiler-warning-level-1-c4930?view=msvc-170) | 2022 | C4930 | 'Entity e(void)': prototyped function not called (was a variable definition intended?) |
| [SonarQube](https://rules.sonarsource.com/cpp/RSPEC-3468/?search=Function%20declarations%20that%20look%20like%20variable%20declarations%20should%20not%20be%20used) | 8.7.0.11108 | cpp:S3468 | Empty parentheses interpreted as a function declaration. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-DCL53-a  CERT\_CPP-DCL53-b  CERT\_CPP-DCL53-c | Parameter names in function declarations should not be enclosed in parentheses. Local variable names in variable declarations should not be enclosed in parentheses. Avoid function declarations that are syntactically ambiguous |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | LANG.STRUCT.DECL.FNEST | Nested Function Declaration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | **Always verify that operations on signed integers does not result in overflow.**  Signed integer overflow can lead to unexpected behavior and exploitable vulnerabilities particularly if the values are used in subsequent operations like pointer arithmetic and array indexing. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below can result in an overflow which can lead to exploitable vulnerabilities or unexpected behavior depending on how it is subsequently used. |
| **void** foo(**signed** **int** a, **signed** **int** b) {  **signed** **int** sum = a + b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The solution is to use the standard precondition test for signed integer overflow anywhere signed integer operations are used. |
| #include <limits.h>    **void** foo(**signed** **int** a, **signed** **int** b) {  **signed** **int** sum;  **if** (((b > 0) && (a > (INT\_MAX - b))) ||  ((b < 0) && (a < (INT\_MIN - b)))) {  /\* Handle error \*/  } **else** {  sum = a + b;  }  /\* ... \*/  } |

| **Principles(s):**  **Principle #1**  This principle states that input data should always be validated. Standard 002 dictates that integer operations should not be allowed to overflow, and this can often occur due to malicious input by an attacker attempting to smash the stack or otherwise take the system down. Thus, following principle #1 by validating input data helps to support the standard of avoiding integer overflow.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 002 can lead to many vulnerabilities resulting from signed integer overflow. As such, following this principle will support standard 002 by ensuring that developers think ahead about preventing common vulnerabilities like integer overflow and wrapping.  **Principle #8**  This principle states that a system should utilize an in-depth defense. Standard 002 dictates that integer operations should not be allowed to overflow because this can lead to serious vulnerabilities like stack smashing and arbitrary code execution. As such, following principle #8 will help to ensure these vulnerabilities are addressed by employing many layers of defense, including a layer implementing standard 002 and other layers that support it like input validation.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely test for edge cases including values that would cause signed integer types to overflow. Thus, following principle #9 by employing quality tests that gain high code coverage will help support the standard of avoiding signed integer overflow.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 002 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid signed integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.3 | C2800, C2860  C++2800, C++2860  DF2801, DF2802, DF2803, DF2861, DF2862, DF2863 |  |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.9.0 | premium-cert-int32-c | Partially implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Always ensure that enough storage has been allocated for strings to contain character data and the null terminator Not ensuring that a buffer has enough storage to contain the data it will receive can result in a buffer overflow which can lead to a host of extremely serious vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The use of a fixed size buffer together with an unbounded input results in a buffer overflow. |
| #include <iostream>    **void** foo() {  **char** buffer[12];  std::cin >> buffer;  } |

| **Compliant Code** |
| --- |
| Using dynamically allocated storage when the input size is unbounded is the only way to to prevent buffer overflow, thus std::string should be used to store the input data. |
| #include <iostream>  #include <string>    **void** foo() {  std::string user\_input;  std::cin >> user\_input;  } |

| **Principles(s):**  **Principle #1**  This principle states that input data should always be validated. Standard 003 dictates that storage for strings should have enough space for the character data and the null terminator. Failure to adhere to this standard can often occur due to reading a string of size greater than the buffer it will be stored in from user input, causing stack variables to be overwritten and thus opening the door to a host of vulnerabilities. Therefore, following principle #1 by validating input data helps to support the standard of avoiding buffer overflows due to incorrectly sized strings.  **Principle #2**  This principle states that developers should always heed compiler warnings. In the case of standard 003, most compilers will indeed generate a warning if a string is copied into a fixed-size array. Thus, principle #2 will help support the standard of avoiding buffer overflows due to incorrectly sized strings.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 002 can lead to many vulnerabilities resulting from buffer overflows. As such, following this principle will support standard 002 by ensuring that developers think ahead about preventing common vulnerabilities like buffer overflows.  **Principle #4**  This principle says to “keep it simple.” In the context of standard 003, it is almost always the case that std::string can be used instead of a C-style string. Instead of trying to gain a little bit of performance or trying to ensure compatibility with libraries written in C, a more simple approach using std::string should probably be used. Thus, following principle #4 will support standard 003 by encouraging developers to keep their code simple and robust.  **Principle #7**  This principle states that data sent between systems should be sanitized. Standard 003 dictates that storage for strings should have enough space for the character data and the null terminator. Sending an incorrectly sized string or any other type of data that does not conform to another system's format to that system can cause vulnerabilities and errors. Thus, following principle #7 supports standard 002 by ensuring that the data sent across systems is correctly sanitized, including ensuring strings are of the appropriate size.  **Principle #8**  This principle states that a system should utilize an in-depth defense. Standard 003 dictates that storage for strings should have enough space for the character data and the null terminator. Failure to adhere to this standard can often occur due to reading a string of size greater than the buffer it will be stored in from user input, causing stack variables to be overwritten and thus opening the door to a host of vulnerabilities. As such, following principle #8 will help to ensure these vulnerabilities are addressed by employing many layers of defense, including a layer implementing standard 003.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely test for edge cases including values that would cause character buffers to overflow. Thus, following principle #9 by employing quality tests that gain high code coverage will help support the standard of ensuring that strings have enough memory to avoid overflows.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 003 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [SonarQube](https://rules.sonarsource.com/cpp/RSPEC-5945/?search=C-style%20array%20should%20not%20be%20used) | 8.7.0.11108 | cpp:S5945 | Use "std::string" instead of a C-style char array. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | MISC.MEM.NTERM  LANG.MEM.BO LANG.MEM.TO | No space for null terminator  Buffer overrun Type overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | 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' |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | C++5216  DF2835, DF2836, DF2839, |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | **Never allow user input to be executed in an SQL query**  Allowing user input to be directly executed in an SQL query allows for SQL injection attacks which can completely destroy a database or result in the attacker gaining access to all of the data. |

| **Noncompliant Code** |
| --- |
| This SQL statement takes user input and places it directly into the query string. When the statement is then executed it also executes the user input which can include SQL expressions that cause an injection attack. |
| #include <iostream>  #include “sqlite3.h”  int main()  {  db = initializeDb();  std::string username;  std::cin >> username;    // user input concatenated directly into the query  std::string query = “SELECT AGE FROM USERS WHERE NAME = “ + username;  // will execute the whole query, including the user input  sqlite3\_exec(db, query.c\_str(), callback, NULL, &error\_message);  } |

| **Compliant Code** |
| --- |
| In the following compliant code, the user input is bound to a prepared statement which executes the query without executing the user input. |
| #include <iostream>  #include “sqlite3.h”  int main()  {  db = initializeDb();  std::string userInput;  std::cin >> userInput;    std::string preparedQuery = “SELECT AGE FROM USERS WHERE NAME = ?“;  sqlite3\_stmt\* sqlStatement;    sqlite3\_prepare\_v2(  db,  preparedQuery.c\_str(),  preparedQuery.length(),  &sqlStatement,  nullptr  );    sqlite3\_bind\_text(  sqlStatement,  1,  userInput.c\_str(),  preparedQuery.length(),  SQLITE\_STATIC  );  std::string query = sqlite3\_expanded\_sql(sqlStatement);  sqlite3\_exec(db, query, callback, NULL, &error\_message);  } |

| **Principles(s):**  **Principle #1**  This principle states that input data should always be validated. Standard 004 dictates that user input should never be executed in an SQL query, as this can result in an SQL injection attack that compromises the application and database. Therefore, following principle #1 by validating input data helps to support the standard of avoiding SQL injection by preventing user input from being directly executed and rejecting input that matches the patterns of common SQL injection attacks.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 002 can lead to many vulnerabilities resulting from SQL injection. As such, following this principle will support standard 002 by ensuring that developers think ahead about preventing common vulnerabilities like SQL injection.  **Principle #6**  This principle states that a user or program should be granted the least amount of privilege necessary to complete its tasks. Standard 004 states that users should never be able to directly query the database. Thus, following principle #6 supports standard 004 by encouraging developers to ensure that user input cannot be directly executed against the database.  **Principle #5**  This principle states that an application should deny all access requests by default. This principle applies to SQL injection by denying any from being directly executed in an SQL query. Thus, following this principle supports standard 004 by preventing suspicious user input from being executed by the SQL interpreter.  **Principle #7**  This principle states that data sent between systems should be sanitized. If an SQL query containing user input was sent to another system, it could gain access and compromise that system. Thus, following this principle by only sending prepared statement queries to other systems supports standard 004 by ensuring user input is never executed in an SQL query.  **Principle #8**  This principle states that a system should utilize an in-depth defense. Standard 004 dictates that user input should never be executed in an SQL query, as this can result in an SQL injection attack that compromises the application and database. As such, following principle #8 will help to ensure SQL injection is not possible by employing many layers of defense, including a layer implementing standard 004.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely have dynamic tests like input fuzzing or penetration testing that would uncover an SQL injection vulnerability. Thus, following principle #9 by employing quality tests that gain high code coverage will help support the standard of preventing user input from being executed in an SQL query.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 004 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [sqlmap](http://org) | any |  | sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers |
| [jsql-injection](https://github.com/ron190/jsql-injection) | any |  | jSQL Injection is a lightweight application used to find database information from a server |

#### 

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | **Always check for and properly handle memory allocation errors**  Unhandled memory allocation errors, especially if they result from heap exhaustion, can result in undefined behavior and vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The noncompliant code creates an array of integers using ::operator new[](std::size\_t),which throws a std::bad\_alloc exception if it fails, without checking the result of the allocation. Because the function is marked as noexcept the caller assumes that it will not generate an exception, but when a std::bad\_alloc exception is generated it could lead to undefined behavior and abnormal termination of the program. |
| #include <cstring>    **void** foo(**const** **int** \*arr, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];  std::**memcpy**(copy, arr, size \* **sizeof**(\*copy));  // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| The solution is to use the new operator with std::nothrow which returns a null pointer if the allocation fails or a pointer to the allocated memory. Then, a test should always be performed to check whether the pointer is null and to properly handle the failed allocation. |
| #include <cstring>  #include <new>    **void** foo(**const** **int** \*arr, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {  // Handle error  **return**;  }  std::**memcpy**(copy, arr, size \* **sizeof**(\*copy));  // ...  **delete** [] copy;  } |

| **Principles(s):**  **Principle #1**  This principle states that input data should always be validated. Standard 004 dictates that memory allocation failures should be detected and properly handled. Sometimes, malicious user input can cause heap exhaustion and, thus, memory allocation failure. Therefore, following principle #1 by validating input data helps to support the standard of handling memory allocation errors by preventing user input from causing a memory allocation failure.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves preventing errors, and standard 005 dictates that the program should detect and handle memory allocation errors. Not following standard 005 can lead to runtime errors from allocation failure, causing vulnerabilities and undefined behavior. As such, following this principle will support standard 005 by ensuring that memory allocation errors are properly handled.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely have dynamic tests that could test how a program behaves in the event of heap exhaustion and thus would uncover a lack of memory allocation failure handling. Therefore, following principle #9 by employing quality tests that gain high code coverage and thoroughly test edge cases will help support the standard of handling memory allocation errors.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 005 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [SonarQube](https://rules.sonarsource.com/cpp/RSPEC-5025/?search=Memory%20hould%20) | 8.7.0.11108 | cpp:S5025 | Replace the use of "new" with an operation that automatically manages the memory. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | **Assertions are not substitutes for error handling code**  Using assertions instead of proper error handling code can result in early termination of the program. |

| **Noncompliant Code** |
| --- |
| The following code calls a function which returns a success code. No error handling is performed based on the error code but an assertion is used to see if the function call failed. |
| int main()  {  int err = foo();  // no error handling code  ASSERT(!err);  } |

| **Compliant Code** |
| --- |
| Use an assertion to see if the error has gone unhandled. |
| int main()  {  int err = foo();  // error handling code that resets err if successful  ASSERT(!err);  } |

| **Principles(s):**  **Principle #9**  This principle states that effective quality assurance techniques should be used. Standard 006 dictates that assertions should not be used instead of error handling code. Asserting that an error code is some value is no replacement for actually testing the value and determining what the application should do in response to the error. Following principle #9 helps support standard 006 by ensuring that code is thoroughly tested and unhandled errors are identified.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 006 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GoogleTest | any |  | GoogleTest is a specialized library utilized to conduct unit testing in the C++ programming language. It can be used to uncover problems and errors resulting from a lack of error handling. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP] | **Always handle exceptions**  If no matching handler for an exception is found then the program will terminate by calling std::terminate, potentially before unwinding the stack. Thus unhandled exceptions can allow for denial of service attacks by causing the program to abnormally terminate. |

| **Noncompliant Code** |
| --- |
| In the noncompliant example below, exceptions thrown by exception\_func()are not caught, resulting in std::terminate()being called. |
| **void** exception\_func() noexcept(**false**);    **void** foo() {  exception\_func();  }    **int** main() {  foo();  } |

| **Compliant Code** |
| --- |
| The solution is to correctly handle any exception-generating code using a try/catch block in the main()function which ensures that the stack is unwound and the program gracefully manages external resources without terminating abnormally. |
| **void** exception\_func() noexcept(**false**);    **void** foo() {  exception\_func();  }    **int** main() {  **try** {  foo();  } **catch** (...) {  // handle error  }  } |

| **Principles(s):**  **Principle #2**  This principle states that developers should always heed compiler warnings. In the case of standard 007, most compilers will indeed generate a warning if the exception-generating code does not have an accompanying exception handler. Thus, principle #2 will help support the standard of avoiding unhandled exceptions.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 007 can lead to a program terminating abnormally and not freeing resources which is a common denial-of-service attack vector. As such, following this principle will support standard 007 by ensuring that developers think ahead about preventing common vulnerabilities like handling all exceptions.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 007 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-008-CPP | **Verify that unsigned integer operations do not wrap**  Unsigned integer wrapping can cause unexpected behavior and vulnerabilities particularly if the values are used in subsequent operations like allocating memory or pointer arithmetic. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below adds two unsigned integers without checking for wrapping. The resulting value sum may then be used later to allocate insufficient memory which could result in a further cascade of vulnerabilities. |
| **void** foo(unsigned **int** a, unsigned **int** b) {  unsigned **int** sum = a + b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| A standard precondition test should be used to ensure that the addition of the two integers will not wrap. |
| #include <limits.h>    **void** func(unsigned **int** a, unsigned **int** b) {  unsigned **int** sum;  **if** (UINT\_MAX - a < b) {  /\* Handle error \*/  } **else** {  sum = a + b;  }  /\* ... \*/  } |

| **Principles(s):**  **Principle #1**  This principle states that input data should always be validated. Standard 008 dictates that unsigned integer operations should not be allowed to wrap, and this can often occur due to malicious input by an attacker attempting to smash the stack or otherwise take the system down. Thus, following principle #1 by validating input data helps to support the standard of avoiding unsigned integer wrapping.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 008 can lead to many vulnerabilities resulting from unsigned integer wrapping. As such, following this principle will support standard 008 by ensuring that developers think ahead about preventing common vulnerabilities like integer overflow and wrapping.  **Principle #8**  This principle states that a system should utilize an in-depth defense. Standard 008 dictates that unsigned integer operations should not be allowed to wrap because this can lead to serious vulnerabilities like stack smashing and arbitrary code execution. As such, following principle #8 will help to ensure these vulnerabilities are addressed by employing many layers of defense, including a layer implementing standard 008.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely test for edge cases including values that would cause unsigned integer types to wrap. Thus, following principle #9 By employing quality tests that gain high code coverage will help support the standard of avoiding unsigned integer wrapping.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 008 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | CERT\_C-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoid wraparounds when performing arithmetic integer operations  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | CERT C: Rule INT30-C | Checks for:   * Unsigned integer overflow * Unsigned integer constant overflow   Rule partially covered |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.9.0 | premium-cert-int30-c | Partially implemented |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-009-CPP | **Always construct and destruct objects when manually managing object lifetimes**  Failure to call the constructor and destructor of an object to begin and end its lifetime can result in the use of the object outside of its lifetime, which is undefined behavior. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code below, an object of the Entity class is created using a call to std::malloc(). Because the class has non-trivial initialization due to the user-specified constructor, the constructor is never called, and the later call to the member function e->foo()results in undefined behavior. |
| #include <cstdlib>    **struct** Entity {  Entity();    **void** foo();  };    **void** bar() {  Entity \*e = **static\_cast**<Entity \*>(std::**malloc**(**sizeof**(Entity)));    e->f();    std::**free**(e);  } |

| **Compliant Code** |
| --- |
| The solution is to allocate memory for the object in a separate variable, and then explicitly call both the constructor and destructor of the object. |
| #include <cstdlib>  #include <new>    **struct** Entity {  Entity();    **void** foo();  };    **void** bar() {  **void** \*ptr = std::**malloc**(**sizeof**(Entity));  Entity \*e = **new** (ptr) Entity;    e->f();    e->~Entity();  std::**free**(ptr);  } |

| **Principles(s):**  **Principle #2**  This principle states that developers should always heed compiler warnings. In the case of standard 009, most compilers will indeed generate a warning if an object is constructed and not destroyed or vice versa. Thus, principle #2 will help support the standard of avoiding manual memory management errors due to a lack of a constructor or destructor.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 009 can result in failing to construct or destruct an object, leaving its internal state inconsistent, which can result in accidental information exposure. As such, following this principle will support standard 009 by ensuring that developers think ahead about preventing common vulnerabilities like failure to construct or destruct objects.  **Principle #9**  This principle states that developers should use effective quality assurance techniques. A good test environment will likely include tests for an object's constructors and destructors to ensure they are called and the memory is created or destroyed for the object. Thus, following principle #9 By employing quality tests that gain high code coverage will help support the standard of properly constructing and destructing objects when manually managing memory.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 009 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-MEM53-a | Do not invoke malloc/realloc for objects having constructors |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | DF4761, DF4762, DF4766, DF4767 |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | CERT C++: MEM53-CPP | Checks for objects allocated but not initialized (rule fully covered). |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.33 | V630, V749 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-010-CPP | **Never access freed memory**  An attempt to access an object outside of its lifetime,which begins when memory for the object is allocated and ends when the memory has been released or reused, results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code below, a pointer to an object of type Entity is dereferenced after it has been deallocated which can result in a write-after-free vulnerability. |
| #include <new>    **struct** Entity {  **void** f();  };    **void** g() noexcept(**false**) {  Entity \*e = **new** Entity;  // ...  **delete** e;  // ...  e->f();  } |

| **Compliant Code** |
| --- |
| The solution below deallocates memory after is done being used. |
| #include <new>    **struct** Entity {  **void** f();  };    **void** g() noexcept(**false**) {  Entity \*e = **new** Entity;  // ...  e->f();  **delete** e;  } |

| **Principles(s):**  **Principle #2**  This principle states that developers should always heed compiler warnings. In the case of standard 009, most compilers will indeed generate a warning if an object is accessed outside of its lifetime. Thus, following principle #2 will help support the standard of avoiding errors due to attempting to access an object before or after its lifetime.  **Principle #3**  This principle states that a system should be architected and designed for security. Part of applying this principle involves thinking ahead about common vulnerabilities and coding defensively to prevent them. Not following standard 010 can result in writing memory that has been deallocated and arbitrary code execution with the permissions of the vulnerable process. As such, following this principle will support standard 010 by ensuring that developers think ahead about preventing common vulnerabilities like not accessing freed memory.  **Principle #10**  This principle states that a secure coding policy should be adopted. Because standard 010 is part of the Green Pace security policy, following principle #10 will help ensure that the standard is upheld. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.1p0 | ALLOC.UAF | Use after free |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.3 | C++4303, C++4304 |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

#### 

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



Automation is essential to enforcing compliance with the standards defined in this policy. Automation tools like static and dynamic analyzers, input fuzzers, and penetration testers can be used to identify code that does not conform to the standards outlined in the policy. As such, these automation tools must be integrated into Green Paces' current DevOps pipeline to transition to a DevSecOps approach. There are two stages of the traditional DevOps pipeline where automation tools that identify non-compliant code should be implemented: the building stage and the verification and testing stage. Incorporating security-focused automation into these stages of DevOps is a critical part of transitioning to a DevSecOps pipeline and improving the defense in depth of the system, as it brings security into the development process so that developers can take a proactive and defensive approach.

Most automation typically occurs in the verification and testing stage of the software development lifecycle (SDLC). Indeed, the dynamic analysis tools that test for runtime issues like SQL injection or memory allocation failures should be incorporated into this stage as they require the application, or at least components of the application, to be in a production-ready state and ideally deployed in a private test environment. Automated static analysis tools, however, will be more effective if incorporated into the building stage. The reason for the inclusion of automated static analysis into the building stage is that it is more efficient and cost-effective to identify code that is not compliant with the standards in this policy early in the DevSecOps pipeline, ideally while the developer is actually writing the code. Identifying non-compliant code in the building process allows it to be rectified before the verification and testing phase begins, which can save time and costs that occur when code has to be sent back to the build stage because it has failed certain tests in a later stage.

This type of automation strategy can be implemented in the building stage by leveraging tools that integrate seamlessly with the development environment. In the case of Green Pace, most of the development team uses the Visual Studio integrated development environment (IDE). Several static analysis plugins available for this IDE, including SonarQube and CppCheck, can inform the developer that they are writing non-compliant code in real-time as they type. Practicing test-driven development (TDD) by using tools like GoogleTest can also help developers identify non-compliant code as they are writing it simply by executing the test cases. Additionally, dynamic analysis tools, like Valgrind and sqlmap, can be incorporated into the verification and testing phase either as a part of the existing testing environment or as a part of the CI/CD pipeline. These dynamic analysis tools can then be run against a production-ready application so that issues and vulnerabilities can be uncovered in a controlled environment. Utilizing these tools and techniques in the building verification and testing phase will ensure that non-compliant code is caught early and can be fixed before moving on to production, where it can cause vulnerabilities and potentially be exploited.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unikely | Medium | Low | 2 |
| STD-002-CPP | High | Likely | High | High | 5 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Likely | Medium | Medium | 4 |
| STD-006-CPP | Low | Likely | Low | Low | 1 |
| STD-007-CPP | Low | Unlikely | Medium | Medium | 2 |
| STD-008-CPP | High | Likely | High | High | 5 |
| STD-009-CPP | High | Likely | Medium | High | 5 |
| STD-010-CPP | High | Likely | Medium | Medium | 4 |

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | All data at rest on a server or any other persistent storage device should be encrypted using an algorithm appropriate for the use case. Encryption at rest refers to the process of encrypting data that is stored on disk. This type of encryption is vital to prevent attacks and exploits that target server hard drives or other physical storage devices. An example of how encryption at rest can be used in practice would be the use of a symmetric cipher like the XOR algorithm, which uses the same key for encryption and decryption to transform data into an encrypted form before it is written to the disk. Once it is time to read this data again, it can be decrypted using the same key and cipher. As such, any time file I/O occurs within an application, the file data would need to be encrypted or decrypted before writing or being used. |
| Encryption in flight | All data transmitted over a network should be encrypted using an appropriate protocol, e.g., SSL, TLS, or SFTP. Encryption in flight is the process of encrypting data that is being transmitted over a network. This type of encryption is vital to prevent man-in-the-middle and other attacks that attempt to gain sensitive data by intercepting network traffic. A practical example of how this principle should be used is always using the HTTPS protocol for all browser traffic. HTTPS ensures that all browser traffic is encrypted using the Transport Layer Security (TLS) protocol. TLS, in turn, leverages an asymmetric cipher to establish a secure session between a client and server and then uses a symmetric cipher to encrypt the data sent over the network. |
| Encryption in use | Encryption in use is the process of encrypting data that is in memory and being used by a program such as an open file or a user input string. Generally speaking, encryption in use works by establishing a region of memory called the trusted execution environment (TEE) which cannot be accessed by any process outside of the TEE, including even the operating system. Data is only decrypted in the TEE using keys stored in the CPU. As such, no other process running on the machine ever sees the decrypted data. Encryption in use does incur a performance cost because the data must be encrypted and decrypted in real-time. However, it can lead to a significant increase in security and should be used when sensitive data is being processed by a third party or is being frequently accessed by other processes. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | All users of an application should be authenticated via credentials like usernames, passwords, and security certificates before they can perform any tasks in the system. Authentication is the process of verifying that a user is who they say they are, usually by requiring them to log in to a user account that stores their credentials or by evaluating the security certificate of a server. A practical example of how this policy should be used includes requiring all users of an application to log into an account using their username and password before they can gain access to the system. This policy is vital to ensure that the system and its data are protected from malicious actors and denial of service attacks. |
| Authorization | All users in a system should be given a certain level of privilege based on a role-based access control (RBAC) system that allows them to perform only the actions necessary to complete their tasks. Authorization is the process of determining what tasks a user can do with their account; for instance, a user account should never be able to delete or modify other user accounts in the database. Similarly, some files or other data present in the application should only be accessible by the admin and should never be exposed to any other users. A practical example of how this principle should be used is to create different roles like users and admins and then specify what types of actions those roles can have, like read-only or read-and-write or read-and-write-and-delete. This policy is essential in all systems that store user data to ensure that the application precisely controls who can do what within the system in order to protect data. |
| Accounting | Detailed accounting in the form of operating system, network, firewall, anti-malware, database, and general application logging should be implemented throughout a system to ensure that abnormal events like security breaches or errors are recorded and reviewed. Accounting is the process of logging or keeping track of particular events like database writes, new accounts being created, admin activities, and suspicious activity like an unusual number of login attempts or user input that resembles an SQL injection attack. A practical example of how this principle should be used includes keeping an operating system log. This log will record all operating events like kernel exceptions and stack dumps, which can be invaluable in understanding how and where an error or attack occurred in an application. This policy applies to all systems that store user data, as it is essential to review logs in the event of a security breach to understand how it occurred and prevent it. |

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 | 12/05/2024 | Initial Policy | Mason Shaner | Mason Shaner |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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