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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc174024843)

[Contents 1](#_Toc174024844)

[Overview 3](#_Toc174024845)

[Purpose 3](#_Toc174024846)

[Scope 3](#_Toc174024847)

[Module Three Milestone 3](#_Toc174024848)

[Ten Core Security Principles 3](#_Toc174024849)

[C/C++ Ten Coding Standards 4](#_Toc174024850)

[Coding Standard 1 5](#_Toc174024851)

[Coding Standard 2 7](#_Toc174024852)

[Coding Standard 3 9](#_Toc174024853)

[Coding Standard 4 12](#_Toc174024854)

[Coding Standard 5 17](#_Toc174024855)

[Coding Standard 6 19](#_Toc174024856)

[Coding Standard 7 21](#_Toc174024857)

[Coding Standard 8 23](#_Toc174024858)

[Coding Standard 9 26](#_Toc174024859)

[Coding Standard 10 28](#_Toc174024860)

[Defense-in-Depth Illustration 30](#_Toc174024861)

[Project One 30](#_Toc174024862)

[Revise the C/C++ Standards 30](#_Toc174024863)

[Risk Assessment 30](#_Toc174024864)

[Automated Detection 30](#_Toc174024865)

[Automation 30](#_Toc174024866)

[Summary of Risk Assessments 32](#_Toc174024867)

[Create Policies for Encryption and Triple A 32](#_Toc174024868)

[Map the Principles 34](#_Toc174024869)

[Audit Controls and Management 35](#_Toc174024870)

[Enforcement 35](#_Toc174024871)

[Exceptions Process 35](#_Toc174024872)

[Distribution 36](#_Toc174024873)

[Policy Change Control 36](#_Toc174024874)

[Policy Version History 36](#_Toc174024875)

[Appendix A Lookups 36](#_Toc174024876)

[Approved C/C++ Language Acronyms 36](#_Toc174024877)

[References 37](#_Toc174024878)

## 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 | To eliminate the majority of software vulnerabilities, input should be validated from all untrusted sources. External data sources should be treated as suspicious, which include user-controlled files, command line arguments, environmental variables, as well as network interfaces (Seacord, 2023). |
| 1. Heed Compiler Warnings | Always use the highest warning level available when compiling code. Only eliminate warnings by modifying code meaning don’t suppress warnings unless necessary. To help detect and eliminate vulnerabilities, use static and dynamic analysis tools (Seacord, 2023). |
| 1. Architect and Design for Security Policies | When creating a software architecture, the software should be designed to use security policies and enforce them. For example, consider implementing an appropriate privilege set if and when your system requires different privileges at various times (Seacord, 2023). |
| 1. Keep It Simple | This means keeping the design as small and simple as possible because the bigger the design the higher the likelihood that errors will occur in the design’s implementation, configuration, and use (Seacord, 2023). |
| 1. Default Deny | Instead of basing access decisions on exclusions, they should be based on permissions. Essentially, this means that access should be denied by default and the protection scheme then identifies under what conditions access should be permitted (Seacord, 2023). |
| 1. Adhere to the Principle of Least Privilege | Processes should execute with the least amount of privileges that are needed to complete their job. If elevated permissions are accessed, this should be done in the shortest amount of time possible to complete the privileged task. This reduces the amount of time an attacker is afforded the opportunity to run arbitrary code with these permissions (Seacord, 2023). |
| 1. Sanitize Data Sent to Other Systems | All data that is passed to complex subsystems should be sanitized. These subsystems include command shells, relational databases, and commercial off-the-shelf (COTS) components. With the use of SQL, command, or other means of injection attacks, attackers could potentially invoke unused functionality into these components. This is not necessarily a problem of input validation because the subsystem being invoked does not understand the context in which the call is made, however, the calling process understands it and is therefore responsible for the data to be sanitized before the subsystem is invoked (Seacord, 2023). |
| 1. Practice Defense in Depth | Multiple defense strategies should be utilized to manage risk. Implementing multiple layers of defense helps ensure that if one layer fails another layer is in place to help prevent a security flaw from becoming a vulnerability that can be exploited, which helps to limit the potential consequences of an exploit that is successful. For example, combining two layers, one with secure programming techniques, and another with secure runtime environments, helps to reduce the risk of any vulnerabilities that remain in the code when it is time to be deployed that could be exploited in the operational environment (Seacord, 2023). |
| 1. Use Effective Quality Assurance Techniques | Fuzz testing, penetration testing, and source code audits are all good quality assurance techniques, which when applied can be effective in identifying and eliminating vulnerabilities, and should be incorporated as part of an effective quality assurance program. Independent security reviews, which offer an independent perspective can lead to more secure systems; for example, their perspectives do not include the biases that may exist from internal sources and can help in finding and correcting assumptions that are invalid (Seacord, 2023). |
| 1. Adopt a Secure Coding Standard | A secure coding standard should be developed for and applied to your targeted development language and platform (Seacord, 2023). |

### 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** | DCL-50-CPP | Do not define a C-style variadic function. |

| **Noncompliant Code** |
| --- |
| This example uses a C-style variadic function to add a series of integers together. The function continues to read arguments until a value of 0 is found and therefore calling this function without passing it a value of 0 as an argument (after the first two) results in undefined behavior. Also, if a data type other than an integer is passed to the function, this also results in undefined behavior (Seacord, 2023, Rule 01 Declarations and Initialization). |
| #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** |
| --- |
| For this particular solution, to allow identical behavior for call sites, a variadic function using function parameter pack is used to implement the function add(). Unlike the variadic function used in the non-compliant example, this function does not result in undefined behavior if the list of parameters passed in is not terminated with a 0 value. Also, unlike before, if a value is passed in that is not of the integer data type, this time it does not result in undefined behavior, and the code is ill-formed (Seacord, 2023, Rule 01 Declarations and Initialization). |
| #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...);  } |

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

| **Principles(s):**  1 - Validate Input – Command line arguments could potentially be a source for incorrect parameter types being passed to a function, which could lead to undefined behavior.  2 - Head Compiler Warnings – Using static analysis tools could help in identifying this problem.  3 - Architect and Design for Security Policies – Design software to enforce security policies, including implementing coding industry standards and best practices.  8 - Practice defense in depth – There should be multiple defense strategies in place to help prevent security flaws from becoming exploitable vulnerabilities. One of these defense strategies should be using secure programming techniques.  9 - Use effective quality assurance techniques – This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Adopt a secure coding standard – having a secure coding standard in place will help ensure secure coding practices are implemented/enforced. Using static analysis tools should be a part of the secure coding standard, and should be a tool that is regularly utilized by developers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Function-ellipsis | Fully Checked |
| Clang | 3.9 | Cert-dcl50-cpp | Checked by clang-tidy |
| Parasoft C/C++test | 2023.1 | CRET\_CPP\_DCL50-a | Functions shall not be defined with a variable number of arguments |
| Polyspace Bug Finder | R2024a | CERT C++: DCL50-CPP | Checks for function definition with ellipsis notation (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | EXP-53-CPP | Do not read uninitialized memory. |

| **Noncompliant Code** |
| --- |
| For this non-compliant example, a local variable is uninitialized and used as part of an expression to print the value of it, which results in undefined behavior (Seacord, 2023, Rule 02 Expressions). |
| #include <iostream>    **void** f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| This compliant solution shows that the object is initialized before printing its value (Seacord, 2023, Rule 02 Expressions). |
| #include <iostream>    **void** f() {  **int** i = 0;    std::cout << i;  } |

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

| **Principles(s):**  2 - Head Compiler Warnings – This type of error will generate a compiler error/warning. Developers should address these types of errors and warnings to ensure the software does not result in undefined behavior. Also, using static analysis tools can help identify this issue.  3 - Architect and Design for Security Policies – Design software to enforce security policies, including implementing coding industry standards and best practices.  8 - Practice defense in depth – There should be multiple defense strategies in place to help prevent security flaws from becoming exploitable vulnerabilities. One of these defense strategies should be using secure programming techniques.  9 - Use effective quality assurance techniques – This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Adopt a secure coding standard – Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local Uninitialized variable |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug Finder | R2024a | CERT C++: EXP53-CPP | Checks for non-initialized variable and non-initialized pointer. Rule partially covered. |
| RuleChecker | 22.10 | Uninitialized-read | Partially checked. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR-50-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| In this non-compliant code example, the input is unbounded, which leads to a buffer overflow (Seacord, 2023, Rule 05 Characters and Strings). |
| #include <iostream>    **void** f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| As seen in the following compliant code block, to ensure data is not truncated and to protect against buffer overflows, it is best to use std::string instead of a bounded array (Seacord, 2023, Rule 05 Characters and Strings). |
| #include <iostream>  #include <string>    **void** f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):**  1 - Validate Input – If a bounded array were used, validating the size of the input before it is assigned to the array would help mitigate this vulnerability  2 - Head Compiler Warnings – Using static analysis tools will help identify potential buffer overflows  3 - Architect and Design for Security Policies – Design software to enforce security policies, including implementing coding industry standards and best practices.  4 - Keep it simple – using strings instead of character arrays for strings will help ensure this type of problem does not happen. Keeping the design more simple will help reduce the likeliness of a potential vulnerability within the software.  8 - Practice defense in depth – There should be multiple defense strategies in place to help prevent security flaws from becoming exploitable vulnerabilities. One of these defense strategies should be using secure programming techniques.  9 - Use effective quality assurance techniques – This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Adopt a secure coding standard – Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Stream-input-char-array | Partially checked + soundly supported |
| 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 | 2023.1 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | (b) – Avoid overflow due to reading a not zero terminating string  (c) – Avoid overflow when writing to a buffer  (e) – Prevent buffer overflows from trained data  (f) – Avoid buffer write overflow from trained data  (g) – Do not use the ‘char’ buffer to store input from ‘std::cin’ |
| RuleChecker | 22.10 | Stream-input-char-array | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS-00-J | Prevent SQL injection. |

| **Noncompliant Code** |
| --- |
| This non-compliant code example of JDBC code shows that authenticates a user to a system. The password is passed as a character array where the database connection is created, then the passwords are hashed. This code allows a SQL injection attack to occur by using an unsanitized input argument “username” into the SQL command allowing the attacker to use the OR value = value attack, in this case, validuser’ OR ‘1’=’1’ is injected (Seacord, 2023, Rule 00 Input Validation and Data Sanitization). |
| **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;    **class** Login {  **public** Connection getConnection() **throws** SQLException {      DriverManager.registerDriver(**new**              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"  **return** DriverManager.getConnection(dbConnection);    }      String hashPassword(**char**[] password) {      // Create hash of password    }    **public** **void** doPrivilegedAction(String username, **char**[] password)  **throws** SQLException {      Connection connection = getConnection();  **if** (connection == **null**) {        // Handle error      }  **try** {        String pwd = hashPassword(password);          String sqlString = "SELECT \* FROM db\_user WHERE username = '"                           + username +                           "' AND password = '" + pwd + "'";        Statement stmt = connection.createStatement();        ResultSet rs = stmt.executeQuery(sqlString);    **if** (!rs.next()) {  **throw** **new** SecurityException(            "User name or password incorrect"          );        }          // Authenticated; proceed      } **finally** {  **try** {          connection.close();        } **catch** (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a parametric query with the ‘?’ character as a placeholder for the argument. Also, to prevent an attacker from submitting an arbitrarily long user name, the code validates the length of the username argument (Seacord, 2023, Rule 00 Input Validation and Data Sanitization). |
| **public** **void** doPrivilegedAction(    String username, **char**[] password  ) **throws** SQLException {    Connection connection = getConnection();  **if** (connection == **null**) {      // Handle error    }  **try** {      String pwd = hashPassword(password);        // Validate username length  **if** (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();  **if** (!rs.next()) {  **throw** **new** SecurityException("User name or password incorrect");      }        // Authenticated; proceed    } **finally** {  **try** {        connection.close();      } **catch** (SQLException x) {        // Forward to handler      }    }  } |

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

| **Principles(s):**  1 - Validate Input – SQL injection attacks are highly susceptible to user input and how it is handled. By keeping user input separate from SQL queries and ensuring it is treated as data rather than executable code will help prevent SQL injection attacks. User input should be sanitized, meaning characters that could be used to construct malicious SQL queries should be removed or escaped (Martinez, 2024).  2 - Head Compiler Warnings – Use static and dynamic analysis tools to detect and eliminate additional security flaws such as implementations that could lead to possible SQL injection.  3 - Architect and design for security policies – It is beneficial to divide a system into distinct intercommunicating subsystems, each with an appropriate privilege set (Seacord, 2023). One way this can be achieved to help reduce issues involving privilege escalation is to keep the database system separate from other system components. Maintaining a database on its own server is an example of keeping one system component separated from other system components.  5 – Default Deny – Default denial can help prevent SQL injection by making sure that any input that is not explicitly allowed is denied. This can be done through the use of parameterized queries and input validation. If query conditions are satisfied, default denial ensures the protection scheme identifies the conditions in which access should be permitted.  6 – Adhere to the principle of least privilege – If an SQL injection occurs, this reduces the likelihood of an attacker being able to execute malicious code, or perform any tasks that would require elevated privileges.  7 - Sanitize data sent to other systems – SQL injection attacks involve SQL which is used in relational databases. Therefore, as previously mentioned when describing the principle of validating input, user input should be sanitized. This is an effective way of reducing the likelihood of possible SQL injection attacks.  8 - Practice defense in depth – There should be multiple defense strategies in place to help prevent security flaws from becoming exploitable vulnerabilities. One of these defense strategies should be using secure programming techniques, which include writing code that prevents SQL injection attacks. Such techniques include using parameterized queries, proper error handling, and sanitizing inputs.  9 - Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Use effective quality assurance techniques – Implementing routine security audits is an effective way to reduce the likelihood of SQL injection. These types of audits should include thoroughly testing database systems and applications. The involvement of penetration testing and independent security reviews on database systems and applications will help identify potential vulnerabilities related to SQL injection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| CodeSonar | 8.1p0 | JAVA.IO.INJ.SQL | SQL Injection (Java) |
| Parasoft Jtest | 2024.1 | CERT.IDS00.TDSQL | Protect against SQL injection |
| SONARQube | 9.9 | S2077  S3649 | Executing SQL queries is security-sensitive  SQL queries should not be vulnerable to injection attacks |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM-50-CPP | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| In this example, s is dereferenced “ s->f();” after it has been deallocated “delete d”. This can result in a write-after-free vulnerability, which can be exploited to run arbitrary code with the permissions that the vulnerable process has (Seacord, 2023, Rule 06 Memory Management). |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...  **delete** s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, only when the dynamically allocated memory is no longer needed is it then deallocated (Seacord, 2023, Rule 06 Memory Management). |
| #include <new>    **struct** S {  **void** f();  };    **void** g() noexcept(**false**) {    S \*s = **new** S;    // ...    s->f();  **delete** s;  } |

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

| **Principles(s):**  2 - Head Compiler Warnings – Use static and dynamic analysis tools to detect and eliminate additional security flaws such as accessing memory that has been freed  8 - Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 - Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |
| Coverity | V7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Polyspace Bug Finder | R2024a | CERT C++: MEM50-CPP | Checks for Pointer access out of bounds, Deallocation of previously deallocated pointer, and use of previously freed pointer |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL-03-C | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| In this example, the assert() macro is used to assert a property concerning a memory-mapped structure, which is essential for the code to behave correctly. While this is better than nothing, it should be placed in a function and executed, meaning it would typically be far away from the definition of the actual structure it is referring to (Seacord, 2023, REC 02 Declarations and Initializations). |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| A preprocessor conditional statement can be used for assertions involving only constant expressions, which can be seen in the following compliant example. Using #error directives allows for more clearer diagnostic messages. Since assertions are evaluated at compile time, no runtime penalty occurs (Seacord, 2023, REC 02 Declarations and Initializations). |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| **Principles(s):**  2 – Head Compiler Warnings – Using static analysis tools could help in identifying this problem.  8 - Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 - Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 - Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | UNLIKELY | HIGH | LOW | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR-51-CPP | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this particular example of non-compliant code, neither functions f() or main() catch exceptions thrown by throwing\_func(), and since no matching handler can be found for the thrown exception, std::terminate() is called (Seacord, 2023, Rule 08 Exceptions and Error Handling). |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| As seen in this compliant solution, all exceptions are handled by the main entry point, ensuring that the stack unwound up to the main() function, which allows for graceful management of external resources (Seacord, 2023, Rule 08 Exceptions and Error Handling). |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

| **Principles(s):**  2 – Head Compiler Warnings – Static and dynamic analysis tools should be used to help detect and eliminate additional security flaws.  3 – Architect and Design for Security Policies – The software should be designed to implement and enforce security policies, which should include handling all exceptions in code.  4 – Keep it simple – Keeping the code simple will help reduce the likelihood that errors will be made, such as not handling all exceptions.  8 – Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 – Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 – Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | PROBABLE | MEDIUM | LOW | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | a – Always catch exceptions  b – 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 | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.20 | Main-function-catch-all-early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | ERR-50-CPP | Do not abruptly terminate the program |

| **Noncompliant Code** |
| --- |
| In this non-compliant code example, f() is registered as an exit handler with std::at\_exit(), which could result in a call to std::terminate because an exception may be thrown from the throwing\_func() (Seacord, 2023, Rule 08 Exceptions and Error Handling). |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.    throwing\_func();  }    **int** main() {  **if** (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

| **Compliant Code** |
| --- |
| As seen in this compliant code solution, f() function will handle any exceptions thrown by throwing\_func() and therefore will not rethrow (Seacord, 2023, Rule 08 Exceptions and Error handling). |
| #include <cstdlib>    **void** throwing\_func() noexcept(**false**);    **void** f() { // Not invoked by the program except as an exit handler.  **try** {      throwing\_func();    } **catch** (...) {      // Handle error    }  }    **int** main() {  **if** (0 != std::**atexit**(f)) {      // Handle error    }    // ...  } |

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

| **Principles(s):**  2 – Head Compiler Warnings – Static and dynamic analysis tools should be used to help detect and eliminate additional security flaws.  3 – Architect and Design for Security Policies – The software should be designed to implement and enforce security policies, which should include handling all exceptions in code properly. If a program terminates abruptly, this can lead to resources not being freed, which can lead to vulnerabilities, such as denial-of-service attacks.  4 – Keep it simple – Keeping the code simple will help reduce the likelihood that errors will be made, and reduce the likeliness of abnormal or abrupt program termination.  8 – Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 – Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 – Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | PROBABLE | MEDIUM | LOW | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| Polyspace Bug Finder | R2024a | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |
| RuleChecker | 22.10 | Stdlib-use | Partially checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| String Correctness | STR-51-CPP | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| For this particular non-compliant example, a std::string object is created as a result of a call to std::getenv(). This call to std::getenv() returns a null pointer on failure, which can lead to undefined behavior when the environmental variable does not exist or when a different error occurs (Seacord, 2023, Rule 05 Characters and Strings). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant code solution, before the std::string object is constructed, the results from the call to std::getenv() are first checked to see if it is null (Seacord, 2023, Rule 05 Characters and Strings). |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):**  2 – Head Compiler Warnings – Static and dynamic analysis tools should be used to help detect and eliminate additional security flaws.  3 – Architect and Design for Security Policies – The software should be designed to implement and enforce security policies, which should include not allowing a string to be created from a null pointer. This can lead to undefined behavior.  4 – Keep it simple – Keeping the code simple will help reduce the likelihood that errors will be made, such as creating a string from a null pointer.  8 – Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 – Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 – Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | assert\_failure | checks null pointer on failure |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Protection | MEM-52-CPP | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| For this non-compliant code example, an array of type int is created using std::operator new[](std::size\_t), and the results of the allocation are not checked. The caller believes function will not throw any exceptions because it is marked as noexcept. However, because new[](std::size\_t) is able to throw exceptions, this could lead to the program being terminated abnormally (Seacord, 2023, Rule 06 Memory Management). |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the new operator returns either a null pointer or a pointer to the allocated space when using std::nothrow. This solution handles the error condition appropriately when the returned pointer is nullptr, which should always be tested to ensure it is not nullptr before referencing the pointer (Seacord, 2023, Rule 06 Memory Management). |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

| **Principles(s):**  2 – Head Compiler Warnings – Static and dynamic analysis tools should be used to help detect and eliminate additional security flaws.  3 – Architect and Design for Security Policies – The software should be designed to implement and enforce security policies, which should include checking for memory allocation errors. Assuming that this would not cause any errors could be potentially hazardous and could lead to abnormal program termination.  4 – Keep it simple – Keeping the code simple will help reduce the likelihood that errors will be made, such as not handling exceptions caused during memory allocation.  8 – Practice defense in depth – Using multiple layers of defense such as implementing industry standards and best practices while developing code, and then using static analysis tools to detect and eliminate potential security flaws will help prevent vulnerabilities. Doing this in conjunction with performing regularly scheduled code audits will help eliminate potential flaws that could lead to vulnerabilities.  9 – Use effective quality assurance techniques - This principle involves the use of testing and source code audits, which could help identify and mitigate vulnerabilities that could potentially arise from this coding standard. Independent security reviews should also be conducted to avoid biased results and provide the company with a credible and objective assessment of the systems in review.  10 – Adopt a secure coding standard - Having a secure coding standard in place for the targeted development language and platform will help reduce the likeliness of introducing vulnerabilities associated with this coding standard. Using static analysis tools will help identify potential vulnerabilities and should be a part of the adopted secure coding standard. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | HIGH | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | a – Check the return value of new  b – Do not allocate resources in function argument list because the order of evaluation of a function’s parameters is undefined |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2024a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation (tools) are one component of a DevSecOps model that can be implemented throughout various stages to help identify security vulnerabilities and respond accordingly. In a DevSecOps model, security is implemented and prioritized in all phases of the Software Development Lifecycle (SDLC) instead of leaving security till the end, which often happens. In doing so, a DevOps process and infrastructure then become a DevSecOps process and infrastructure, ensuring that security is not an afterthought and left until the end.

In the above DevSecOps diagram, we see that security is prioritized throughout the entire development lifecycle, from pre-production through production. During the pre-production phases of development, the way that automation can be utilized to enforce the standards in this policy is by incorporating threat modeling, which can be implemented from the initial planning and assessment phase, all the way through the verification and testing phase. Threat modeling helps to identify and prioritize potential threats and vulnerabilities that may exist in software applications.

Static analysis tools should be used specifically in the building and testing phases to help identify potential security vulnerabilities. During the design phase, which is test-driven, static tools can be used during testing to ensure this security policy is enforced. This also includes building into applications the use of static analyzers, such as OWASP plugins which can be beneficial in identifying vulnerabilities within application code.

During the build phase, static tools can be used during commits to ensure that any changes to code will automatically be checked for any new or potential issues that may arise from the changes that are made. This helps to promote the maintainability of code quality and security. Additional static analysis tools can be used during the build and testing phases to help identify any other potential vulnerabilities that code may produce.

During production, additional processes can be automated to reduce the complexity of enforcing security policies. Tools exist that can be utilized for penetration testing, the monitoring of applications and networks, responding to threats, and for maintenance and stabilization of these networks and applications.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL-50-CPP | HIGH | PROBABLE | MEDIUM | HIGH | 1 |
| EXP-53-CPP | HIGH | PROBABLE | MEDIUM | HIGH | 1 |
| STR-50-CPP | HIGH | LIKELY | MEDIUM | HIGH | 1 |
| IDS-00-J | HIGH | LIKELY | MEDIUM | HIGH | 1 |
| MEM-50-CPP | HIGH | LIKELY | MEDIUM | HIGH | 1 |
| DCL-03-C | LOW | UNLIKELY | HIGH | LOW | 3 |
| ERR-51-CPP | LOW | PROBABLE | MEDIUM | LOW | 3 |
| ERR-50-CPP | LOW | PROBABLE | MEDIUM | LOW | 3 |
| STR-51-CPP | HIGH | LIKELY | MEDIUM | HIGH | 1 |
| MEM-52-CPP | HIGH | LIKELY | MEDIUM | HIGH | 1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the encryption of data that is not being used and is not in motion (in flight) across networks and between devices. It is the practice of encrypting data stored on a device by encoding it using encryption algorithms (Clinton, 2023). Encryption at rest applies to this security policy because one layer of defense is to encrypt data that is not being used to protect it from potential attacks and to reduce the likelihood of the encrypted data being mishandled. Even if an attacker gains access to the encrypted data, the likelihood of the attacker being able to unencrypt the data is highly unlikely without the appropriate key, especially with the higher-bit encryption that exists today. |
| Encryption in flight | Encrypting data in flight or in motion is the process of encrypting data that is moving from one device to another over a network (OpsCompass, 2015). This is important and applies to this security policy because when data is in motion across a network or networks, it is then while in flight that it is at most risk for attack. If data in motion is not encrypted, an attacker that intercepts this data would be able to read all of the data. Another layer of defense for this security policy is to ensure that all data in flight is encrypted, making the likelihood of an attacker who could potentially intercept this data and read it, far more unlikely than if the data were not encrypted while in motion. |
| Encryption in use | Encryption in use refers to encrypting data while it is being used (phoenixNAP, 2023). Typically, when data is being used (accessed or used by an application), the data is decrypted and converted to plaintext so that it can be more easily read and understood (phoenixNAP, 2023). This poses a significant threat where if an attacker gained access to a system while data that was unencrypted was being used, the attacker could also be able to read the unencrypted data. Encrypting data in use is a method for using the encrypted data without ever having to decrypt the data, maintaining its security. Encryption in use is another defensive strategy that is implemented in this policy to help ensure that if an attacker did gain access to a system while data was being used, they would still not be able to read the data without the appropriate decryption key. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication, the first step in the Triple-A security process, is a network or application's way of identifying who a user is and that that user is who they say they are (Mylonas, 2018). When a user sends a request to be authenticated (a log-in request), the user’s credentials are compared to existing credentials within a database. If the user credentials are found and match the credentials in the database, the user is authenticated (Mylonas, 2018). Authentication applies to this policy because it is an essential component for keeping a network or application secure against potential attackers. It is essentially one component of Triple-A defense, which is a layer of our defense-in-depth. |
| Authorization | Authorization in Triple-A refers to enforcing policies, such as determining what services, resources, and the quality of activities a user is allowed to use (Mylonas, 2018). This could include what changes a user is allowed to make to a database, are they allowed to enroll or create new users, and what level of access the user has. Once a user is authenticated, the user is then assigned a specific authorization level that defines what level of access to a network and its resources they can have (Mylonas, 2018). This is another critical component of Triple-A defense that limits a user to only what is necessary for them to accomplish the tasks they need to accomplish and is a part of the defensive strategy outlined by this policy. |
| Accounting | Accounting in Triple-A defense refers to monitoring and keeping track of the resources that are accessed by users, when, by whom, and what if any commands are issued (Mylonas, 2018). This includes keeping track of what users log in, who makes changes to databases, and what files are accessed. This is also a critical component of the Triple-A defense, which plays a significant role in this policy's defensive strategy. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 07/21/2024 | Updated 10 Core Security Principles, and 10 Coding Standards | Justin Starr |  |
| 1.2 | 8/11/2024 | Updated Coding Standards: Mapping to Principles, Threat Levels, and Automation  Updated DevSecOps Automation, Summary of Risk Assessment, and Policies for Encryption and Triple-A | Justin Starr |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

## 

## References

Clinton, D. (2023, April 4). *What is Encryption at Rest? Explained for Security*

*Beginners* freeCodeCamp. <https://www.freecodecamp.org/news/encryption-at-rest>

Martinez, J. (2024, July 12). *How to Prevent SQL Injection Attacks: 6 Proven*

*Methods*. Stringdm. <https://www.strongdm.com/blog/how-to-prevent-sql-injection-attacks>

Mylonas, L. (2018, November 27). *What is AAA security? An introduction to authentication,*

*authorisation and accounting* Codebots. <https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting>

OpsCompass. (2015, July 10). *Are you encrypting your data-in-flight? If not, you should be*

<https://opscompass.com/resources/are-you-encrypting-your-data-in-flight-if-not-you-should-be/>

phoenixNAP. (2023, November 16). *Data Encryption in Use Explained*.

<https://phoenixnap.com/blog/encryption-in-use>

Seacord, R. (2023). *Top 10 Secure Coding Practices*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCommentId=88044413>