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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

Data Types4

Data Values7

String Correctness9

SQL Injection11

Memory Protection13

Assertions15

Exceptions17

Encryption19

No Except Functions22

Authentication24

[Defense-in-Depth Illustration](#_heading=h.1mrcu09) 17

[Project One](#_heading=h.46r0co2) 28

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 28

[2.](#_heading=h.111kx3o) Risk Assessment 28

[3.](#_heading=h.3l18frh) Automated Detection 28

[4.](#_heading=h.206ipza) Automation 28

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 29

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 30

[7.](#_heading=h.1egqt2p) Map the Principles 31

[Audit Controls and Management](#_heading=h.3ygebqi) 32

[Enforcement](#_heading=h.2dlolyb) 32

[Exceptions Process](#_heading=h.sqyw64) 32

[Distribution](#_heading=h.3cqmetx) 33

[Policy Change Control](#_heading=h.1rvwp1q) 33

[Policy Version History](#_heading=h.4bvk7pj) 33

[Appendix A Lookups](#_heading=h.2r0uhxc) 33

[Approved C/C++ Language Acronyms](#_heading=h.1664s55) 33

## 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 | Validating data input is a measure instilled to make sure the system will not break itself, as well as protecting from malicious users. Buffer overflow is an example of how not validating input can infect the system. A simple check for string length can ensure valid function, as well as managing which datatypes are allowed to be input. |
| 1. Heed Compiler Warnings | When designing systems it is important to not disregard compiler warnings, as even though your code may compile and run, its base function may be impacted when, for example using deprecated functions. Often warnings will do just what they say, warn of a problem to come, that may not be apparent immediately. |
| 1. Architect and Design for Security Policies | When systems are in the design and architecting stage it’s important to consider potential vulnerabilities at the time and proactively so when and if attacks happen the system is already designed to be protected. This allows for robust code that is readable and maintainable overtime, instead of having many security patches when problems arise. |
| 1. Keep It Simple | Keeping code simple and understandable ensures that adding new members to your team is easy as well as lessening risk against attack, and again, maintainable code over time. Less complicated and convoluted code, means less vulnerabilities for malicious attackers to expose, and also is easier to update and maintain over time. |
| 1. Default Deny | Default Deny is a security principle that means all elements of your system should by default be locked, and only with explicit access should they be allowed. This is a low level protection that ensures sensitive data is protected by default. Continued verification is a key element of default deny, such as two factor authentication, or even a financial trading key that updates constantly. |
| 1. Adhere to the Principle of Least Privilege | The Principle of Least Privilege states that a user or function should only have its necessary components to function. For example if a user of a database should only have access to certain collections, then they should be given access to a database containing only those collections, even if others can be locked behind other privileges, why even have the user in the same system as sensitive components? |
| 1. Sanitize Data Sent to Other Systems | Ensuring data sent out to other systems is sanitized is important as it protects the other system from breaking or being attacked. For example if a dataset that has been maliciously altered triggers a data leak in the other system. This protects both systems in turn and ensures that the receiving system is functioning correctly as well. |
| 1. Practice Defense in Depth | Defense in Depth is an important principle that instills all of these principles, as it ensures they are applied into every system and function in combination with each other to create a truly robust system. For example, data validation and sanitization should overlap, as you are validating data input into the system, but it still should be checked as it leaves to another system to ensure proper function all around. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance means testing your code for all sorts of different situations. Extraneous values should be tested, in addition to unit testing, as well as testing across multiple different machines and architectures. Ensuring robustness across different machines, especially in consumer products where many different behaviors may be observed. |
| 1. Provide Only One Way to Do an Operation | Maintaining multiple pathways for the same operation often results in tangled code or configuration, making it harder to keep the system running smoothly. By limiting options, you reduce the complexity of the system, making it easier to manage. With only one way to perform an operation, documentation becomes more straightforward. It's easier to provide clear instructions and maintain up-to-date documentation when there's only one path to document. |

### C/C++ 10 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** | [DAT-001-CPP] | The standard emphasizes the importance of using data types correctly and ensuring that data is used in a way that is consistent with its intended type. This includes avoiding type casting or type mismatches that can lead to vulnerabilities. |

| **Noncompliant Code** |
| --- |
| This code has two variables: ‘secret’, and ‘user-input’, the problem is that ‘user-input’ is a different data type than ‘secret’ and that can create unexpected behavior at runtime |
| int main() {  int secret = 12345;  char user-input[10];  std::cout << "Enter the secret number: ";  std::cin >> user-input;  if (secret == user-input) {  std::cout << "Access granted!" << std::endl;  } else {  std::cout << "Access denied!" << std::endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This code fixes the issue by simply changing the input to the same type as ‘secret’, as well as checking input type before, and will fail automatically if not |
| int main() {  int secret = 12345;  int user-input;  std::cout << "Enter the secret number: ";  std::cin >> user-input;  if (type(secret) == type(user-input)) {  if (secret == user-input) {  std::cout << "Access granted!" << std::endl;  }  } else {  std::cout << "Access denied!" << std::endl;  }  return 0;  } |

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

| **Principles(s):** Data Validation, Least Privilege   * By using stricter data types, the system becomes more robust and sturdy against attack * Even if you type cast back the data becomes more vulnerable to attack, and breaks the principle of Least Privilege, adding unnecessary dangers and vectors of attack |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Low  <$1000 | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Unit Testing | N/A | Bounds Checking: (int)  -2147483648 to 2147483647 | Use a function to test the systems different variables prior to deployment, checking potential bounds problems. This is for int, see [here](https://learn.microsoft.com/en-us/cpp/cpp/data-type-ranges?view=msvc-170) for other data types limits |
| Type Validation | N/A | Type Checking | By adding additional logic and checks, admin is able to ensure the system will not crash, or fault due to type mismatch |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [DAT-002-CPP] | When a system is only expecting a certain pattern, range, or type of value, an extraneous value can throw off its accuracy, or even be vulnerable to attack. |

| **Noncompliant Code** |
| --- |
| Buffer Overflow vulnerability, there is no limit on input, and user\_input only has space of 21. Once 21 is exceeded extraneous behavior is abound. |
| int main()  {  const std::string account\_number = "CharlieBrown42";  char user\_input[21];  std::cout << "Enter a value: ";  std::cin >> user\_input;  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  } |

| **Compliant Code** |
| --- |
| By limiting input and then checking after here you avoid a buffer overflow and are informed if one still somehow occurs. |
| int main()  {  std::cout << "Buffer Overflow Example" << std::endl;  const std::string account\_number = "CharlieBrown42";  char user\_input[21];  std::cout << "Enter a value: ";  std::cin >> std::setw(20) >> user\_input;  if (strlen(user\_input) > 20) {  std::cout << "Buffer Overflow Detected" << std::endl;  }  else {  std::cout << "No Overflows Detected" << std::endl;  }  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  } |

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

| **Principles:** Data Validation, Architect and Design for Security   * Again validating the data, in this case for length, ensures that the system continues to run normally when given unexpected input * Architecting for this attack vector increases security, and should be a principle from the design stage as it is a well documented security vulnerability and easy to protect against |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High  (Can expose many unexpected parts of system) | Low | Medium-High  (Varies on severity)  Buffer Overflow and similar attacks like SQL Slammer, have been cropping up throughout computing/internet history and for microsoft alone patches are being released often | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck  (Static Code Analysis) | 2.12 | Static Code Analysis | Can identify potential buffer overflow vulnerabilities before they unexpectedly occur at runtime, and potentially break systems |
| Unit Testing | N/A | Bounds Checking | Since the input is limited as it is given, there shouldn’t be any problems but checking the bounds anyway is safer, and increase robustness |
| Modern OS | N/A | Built-in to OS | Many Modern OS have protections against buffer overflow, like mitigating executable space in memory at runtime. This protects potential attacks when they happen and block access to parts in the memory that aren’t supposed to be accessed, i.e. passwords, and private data |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-001-CPP] | Strings are a fundamental data type in most programming languages, and improper handling of strings can lead to security vulnerabilities such as buffer overflows, injection attacks, and information leakage. |

| **Noncompliant Code** |
| --- |
| The noncompliant aspect of this code is that it doesn't perform proper bounds checking on the destination string. If the destination string doesn't have enough memory allocated to accommodate the concatenated result, it can lead to undefined behavior and potential memory corruption. |
| int main() {  char destination[10];  // Destination buffer with a size of 10 characters  char source[] = "Hello, World!"; // Source string    strcat(destination, source); // Unsafe string concatenation  std::cout << "Destination: " << destination << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| We safely concatenate the source string to the destination string using the += operator, which ensures that memory is managed correctly. This code is compliant with secure coding standards and does not risk buffer overflows or memory corruption. |
| int main() {  std::string destination = "Hello, ";  std::string source = "World!";    // Perform safe string concatenation using the + operator  destination += source;  std::cout << "Destination: " << destination << std::endl;  return 0;  } |

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

| **Principles(s):** Data Validation, Data Sanitation,   * In this case we don’t restrict the bounds explicitly with string, so that the data type can self manage its bounds as it is updated * The assurance that both the strings are correctly concatenated and strictly the same increases robustness throughout the system as it can be consistently worked with as a string, as it is processed throughout the system |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium-High  (Can involve data corruption if not handled properly) | Medium-  Low | Varies  Low - Medium  (Usually just a system fix) | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| cppcheck | 2.12 | Static Code Analysis  (Buffer Overflow) | Buffer Overflow is again a risk that can be checked for prior to any corruption or attacks with SCA |
| Unit Testing | N/A | Checks Extreme Cases | Using Unit Testing we can find many different extreme cases and identify vulnerabilities in the string type potentially |
| Validation | N/A | Validating Data Coming in and Going out | Checking alone if the data type is a string, (if using strings throughout), us an effective way to verify that all data is accurate and workable |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [SQL-001-CPP] | SQL-Injection infects all types of databases, even noSQL and object based, and has been one of the leading security vulnerabilities since the late 2000s. |

| **Noncompliant Code** |
| --- |
| Example using python linked with SQL, using DynamicSQL style, or a query coded with user input within. This leads very vulnerable systems as the attacker has a direct link to a query statement |
| query = "SELECT \* FROM users WHERE username = '" + user\_input + "'"; |

| **Compliant Code** |
| --- |
| Instead of using dynamic sql, when you regularly receive input you should check it to ensure correctness and validity of input to system |
| if (!std::all\_of(user\_input.begin(), user\_input.end(), ::isdigit)) {  throw std::invalid\_argument("Invalid input");  } |

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

| **Principles(s):** Keep Simple, One Function, Least Privilege, User Authentication, Defense in Depth   * Keeping the code simple in its function allows for easier onboarding of new team members and upkeep of code in general * One Function, or only allowing one way to perform an operation applies here, as dynamic SQL leaves vulnerability for additional unexpected functions and behavior * Least Privilege applies here as the dynamic sql leaves another vector for attack by leaving the functionality open ended to be misused * User Authentication, The SQL code itself is often only accessible by some kind of user and is often exploited by bad actors that have low level access, like consumer * For Defense in Depth, the SQL injection obviously invites many different vectors of attack and vulnerability so defending and planning for this across the board is vital to system health and robustness |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High  High | Medium  Medium | High  Results often in reputation loss, potential for data leaks, DDos, Ransoms, Generally low cost to fix, but outside factors leave it very detrimental to not only system but enterprise as a whole | High  High | 5  5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Third Party “White Hat” Hackers | N/A | Checking for potential vulnerabilities with DynamicSQL and other potential SQL vulnerabilities | Bringing in help outside of the network of people within, allows for a different perspective that a hacker might have and allows consideration for factors possibly not accounted for ahead of time by team |
| Entra ID | N/A | Multi Factored Authentication | Included with Azure SQL applications, as well as SQLServer 2022, Entra ID is a multifactor authentication system which will increase security against potential attackers |
| Restricting Function | N/A | User Permissions and Levels | By completing restricting a users ability to query certain databases, or which fields they can access from databases is a solid step towards |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-001-CPP] | Many of the previous standards go into vulnerabilities in memory protection, but don’t show the risks of the vulnerabilities, like losing accounts, large data leaks, and even total corruption. |

| **Noncompliant Code** |
| --- |
| Similar to the string verification example, the effect of this strcpy can leave leaks in the memory and over time and many executions can corrupt the system greatly |
| int main() {  char buffer[10];  std::string input;  std::cout << "Enter a string: ";  std::cin >> input;  // Unsafe copy  strcpy(buffer, input);  std::cout << "Buffer: " << buffer << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| Changing both types to strings protects against further memory leaks and potentially erratic behavior |
| int main() {  std::string buffer;  std::string input;  std::cout << "Enter a string: ";  std::cin >> input;  // safe operation between strings  buffer+=input;  std::cout << "Buffer: " << buffer << std::endl;  return 0;  } |

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

| **Principles(s):** Data Validation, Architect and Design for Security, Effective Quality Assurance Techniques   * Data Validation comes back into play as it is wise to ensure that data stored in memory doesn’t get overwritten unexpectedly or used when it is incorrect * Design always be cognizant of the vulnerabilities associated with different methods that will be accessing, storing, and executing, different blocks of memory * Memory should be strained and tested in a controlled environment prior to deployment to ensure the different functions are properly storing, and deleting |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium-High | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Unit Testing | N/A | Check different memory stores, potential overloading, stack overflow | Memory can often be impacted by by getting too many requests at a time or through deliberate corruption, unit testing will strain the limits of storage values and show the effects of corrupt data |
| Cppcheck | 2.12 | Static Code Analysis for Potential Memory Leaks | Leaked memory can be dangerous as it can cause both loss of memory and expose sensitive data to unexpected attacks, static code analysis will expose these potential vectors of corruption |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [AST-001-CPP] | Assertions in secure coding standards are used as a defensive programming technique to validate assumptions and ensure that a program behaves as expected in a secure and predictable manner. |

| **Noncompliant Code** |
| --- |
| Without a simple check for zero division, an incorrect input can make the whole system fail, as it will here if division by 0 occurs |
| double calculateAverage(const vector<int>& numbers) {  int sum = 0;  int count = 0;  for (int num : numbers) {  sum += num;  count++;  }  // No assertion to check for an empty vector, which can lead to division by zero  return sum / count;  } |

| **Compliant Code** |
| --- |
| Simple fix by checking to see if count is 0 and, returning an error |
| double calculateAverage(const vector<int>& numbers) {  int sum = 0;  int count = 0;  for (int num : numbers) {  sum += num;  count++;  }  if (count != 0) {  return sum / count;  }  else {  std::cout << "Error: Zero Division " << std::endl;  return 0;  }  } |

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

| **Principles(s):** Data Validation, Data Sanitation, Defense in Depth   * Data validation ensures that we will not have unexpected exceptions occur when an unexpected value is given * Data Sanitation, all incoming values should be checked prior to the function even being called ensuring that the data is already correct and the check within is just an additional security blanket * The above also shows defense in depth, as the system should ensure that all parts are covered even if the protection may seem redundant |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low  Fixing the code is often low cost although it may be broad and implemented throughout the system, can cause issues if important threads or functionality is lost because of exceptions or crashes | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Static Code Analysis | Cppcheck here will alert the admin to potential vulnerabilities with exceptions that may go unchecked |
| Extremes | N/A | Check for Extreme Variations | Not all unexpected values will create an exception but some may create unexpected results and the system should be prepared for this anyway to defend |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [EXP-001-CPP] | Exceptions |

| **Noncompliant Code** |
| --- |
| Same 0 division exception without try-catch operator |
| int main() {  int denominator = 0;  int result = 10 / denominator;  return 0;  } |

| **Compliant Code** |
| --- |
| The try catch enables the code to test an example before the exception and extraneous behavior occurs |
| int main() {  try {  int denominator = 0;  int result = 10 / denominator; // Division by zero will throw an exception  std::cout << "Result: " << result << std::endl;  } catch (const std::exception& e) {  std::cerr << "An error occurred: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Defense in Depth, Heed Compiler Warnings, Architect and Design for Security   * Defense in Depth is represented here as all exceptions should be prepped and prepared for across systems, and will also be handled by the system when they do eventually occur * Heeding Potential dangers in the warnings and accounting for there possible malfunctions/exceptions will further protect against unexpected behavior and errors at runtime * Designing around exceptions is paramount in security as they define how a language is used and checked around |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Static Code Analysis | Again cppcheck will provide additional warnings for potential exceptions and vulnerable parts of code |
| Unit Testing | N/A | Testing Bounds and Extraneous Input | Testing every vulnerable piece of code for potential bounds errors or unexpected behavior from data can be prepared for if it is uncovered during unit testing |
| Ice | 3.7 | Restricts Regions during Runtime | At Server level certain functions need to be locked to a specific thread at a time, ensuring that different requests are not overlapped and causal for error |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Encryption** | [ENC-001-CPP] | Encrypting and Decrypting Data Properly is vital in security as improper encryption or a straight up lack thereof incurs significant risk of data being accessed maliciously and stolen or corrupted |

| **Noncompliant Code** |
| --- |
| This just shows a text file that is not encrypted, if a hacker manages to breach a database through injection or any other means the data is extremely vulnerable |
|  |

| **Compliant Code** |
| --- |
| This shows the same text file in which the contents are encrypted, and should a hacker breach the database it will be meaningless without the proper key and proper encryption method (XOR based) |
|  |

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

| **Principles(s):** Defense in Depth, One Function, Architect and Design for Security, Default Deny   * Defense in Depth applies as it is not enough to protect your data against attacks you should also protect your data in the case that an attack eventually does occur * One Function relates to the strategy of cryptography as it shouldn’t be done in multiple ways as that could lead to problems internally when accessing data * Encryption should be a primary focus of your security design as it protects data even in the face of attacks, and XOR based is near impossible to brute force * This is a shining example of default deny as even if data is breached it is still protected, and MFA also trends toward this principle |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Low | High  If the worst happens and you are breached without encryption expect levels of financial loss similar to injection, although fixing for future should be easier | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Entra ID | N/A | Multi Factored Authentication | Even though brute force is nearly impossible if the attacker is able to be authenticated they can bypass the need to decrypt as the system will do it on its own |
| Verification of Encryption | N/A | Verifying the Encryption Process | This ensure that data that is encrypted can be reworked backwards as if it can’t the data is just as useless to the admin or user as it should be to an attacker |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **No Except**  **Functions** | [EXP-002-CPP] | No Except functions are methods that are declared to not throw exceptions at runtime. They can be useful in efficiency and efficacy as it indicates to other developers that this code will never throw an exception, as well as opening up stack space |

| **Noncompliant Code** |
| --- |
| The same code from the exceptions example earlier with the added argument noexcept, it will however throw an exception if 0 is given as input |
| int main() noexcept {  int denominator = 0;  std::cin >> denominator;  int result = 10 / denominator;  return 0;  } |

| **Compliant Code** |
| --- |
| If Try does fail in this situation the exception is handled within and will not throw or break the noexcept declaration |
| int main() noexcept {  try {  int denominator = 0;  std::cin >> denominator;  int result = 10 / denominator; // Division by zero will throw an exception  std::cout << "Result: " << result << std::endl;  } catch (const std::exception& e) {  std::cerr << "An error occurred: " << e.what() << std::endl;  }  return 0;  } |

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

| **Principles(s):** Defense in Depth, Keep it Simple, Architect and Design for Security   * This enables Defense in Depth as it broadens the robustness of each function and entire system to run more smoothly throughout * In Keeping it Simple by declaring different functions which are protected from exception increases readability, useability, and maintainability by the active team and newcomers alike * This exemplifies Designing for Security as it focuses on streamlining while also further protecting and understanding the system as a whole |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.12 | Static Code Analysis | Cppcheck will identify functions declared with noexcept that will potentially still result in an exception. This allows for the developer to reevaluate solutions for these exceptions as well as the need for the noexcept modifier itself |
| Unit Testing | N/A | Check Bounds for Extreme an Extraneous Errors | Although Cppcheck will likely identify any rogue noexcept functions this will ensure that thorough checking is done around all potential exception points |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Authentication** | [AUT-001-CPP] | Within your C++ system it should always be true that only valid users are accessing sensitive parts or the system altogether |

| **Noncompliant Code** |
| --- |
| Ignoring the fact that our username and password are hardcoded in the system (assume they would be in the database in actual implementation) the password itself is not protected at all and is stored in its standard string form. If the database was to be breached the password would be open for all to see. In addition to this, the failed authentication provides much more information than need be by telling the user, or potential attacker, that either the username or password doesn’t match |
| int main() {  string username = "admin";  string password = “password123”;  string input\_username;  string input\_password;  cout << "Enter username: ";  cin >> input\_username;  cout << "Enter password: ";  cin >> input\_password;  if (input\_username == username) {  if (input\_password == password) {  cout << "Authentication successful!" << endl;  }  else {  cout << “Password Incorrect” << endl;  } else {  cout << "Authentication failed!, Username incorrect" << endl;  }  return 0;  } |

| **Compliant Code** |
| --- |
| This code fixes the problems above by encrypting passwords, and also being more secretive with revealing the problem with what was entered, beyond just “authentication failed” |
| int main() {  string username = "admin";  string password = \*Hashed Password\*;  string input\_username;  string input\_password;  cout << "Enter username: ";  cin >> input\_username;  cout << "Enter password: ";  cin >> input\_password;  input\_password = Hash(input\_password) // Hash is a function that takes a hashed or unhashed //string and transforms it the other way  if (input\_username == username && input\_password == password) {  cout << "Authentication successful!" << endl;  } else {  cout << "Authentication failed!" << endl;  }  return 0;  } |

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

| **Principles(s):** Default Deny, Keep it Simple, Defense in Depth   * Default Deny applies here as passwords are not only protected at storing level, but little information will be given on what field was wrong * Keeping it simple applies to the same factor, as adding unnecessary checks like in the noncompliant code, not only protrudes efficiency but also robustness * Defense in Depth applies here, as although in theory passwords would be protected by database, an additional level of “redundant” defense can not hurt, only help |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Multi Factor Authentication | N/A | Most likely Entra ID in Databases, it will Vary by application | Adding a MFA to any password system will increase security in the system and |
| Rate Limiting | N/A | Leaving the User a specified number of attempts at Authentication | This enables robustness and defense against potential brute force hackers as if no limit is given they can indefinitely attempt until they eventually breach a solution |
| Timing Limit or Function | N/A | In Encrypting it’s important to Limit and Restrict the Timing | Attackers can find passwords and other encrypted data by using specific timing techniques to figure out how the password is processed through encryption. This can be used to reverse engineer the process and break the encryption. Implementing a generic time for the process to take place, eliminates this attack vector almost completely |

### 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.

Unit Testing:

Testing upper and lower bounds of every type of input and output allows Developers to know in the preproduction phase ‘Verify and test’, what their vulnerabilities are, and how to properly manage them. This can also be applied in the ‘Maintain and stabilize’ phase of production

MFA:

Ensuring Multi-Factor Authentication across all relevant structures of systems ensures that all users are who they say they are, and allows us to respond accordingly, as we monitor and detect during the production, and deployment process

Rate Limiting:

Whether it is someone trying to brute force a password, or a buffer overflow, users should be rate limited across the board when submitting too many requests, as well as monitored closely from then on out. This should be planned for in pre-production, tested, and implemented in production, and post-production.

Static Code Analysis:

During production all code should be subject to static code analysis (cppcheck for c++), as it can identify more risks and dangers not immediately apparent at runtime to the compiler. This fits into monitor and detect as well as maintain and stabilize.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DAT-001-CPP | Low | Somewhat Likely | Low | Medium | 1 |
| DAT-002-CPP | High | Unlikely | Medium-High | High | 2 |
| STR-001-CPP | Medium-  High | Somewhat Unlikely | Low-Medium | High | 2 |
| SQL-001-CPP | High | Likely | High | High | 5 |
| MEM-001-CPP | High | Unlikely | Medium-High | High | 3 |
| AST-001-CPP | Medium | Likely | Low | Low | 2 |
| EXP-001-CPP | Medium | Likely | Low | High | 1 |
| ENC-001-CPP | High | Unlikely | High | High | 5 |
| EXP-002-CPP | Low | Likely | Low | Medium | 1 |
| AUT-001-CPP | High | Unlikely | Medium-High | High | 4 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest, is the encryption of data that is not actively being used and is often stored in a hard drive or database. This data at rest is often some of the most sensitive, such as bank information, address, and contact details. This makes it a target for attackers even though it may be harder to breach than data in transit. It is important to have strong encryption at rest for this reason so breaches are less likely to be successful. |
| Encryption at flight | Encryption at flight describes the type of encryptions needed to protect data moving from device to device or within the cloud. It is used when passwords are being authenticated through third party systems or between an owned server. This is important to protect for authentication purposes obviously and data can be particularly vulnerable at flight so strong encryption is key. |
| Encryption in use | Encrypting data in use describes the encryption of data while it is being actively processed and worked through. This seems impractical as it will make many functions seemingly unnecessarily complicated, but in fact it is vital to protect data at this stage as unencrypted data actively in use is much more likely to suffer from data breaches. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is usually the verification of an end-user, to make sure they’re actually who they are. This is usually through username and passwords, but now two factor authentication using your cell phone or email is also fairly common, especially in enterprise accounts. This policy is important to differentiate admins from basic users, as well as verify a valid user and operations. New users being added are usually authenticated, or confirmed through an email system or text system as well. |
| Authorization | After authentication, authorization takes place to designate where different users are allowed to go, or what commands they can enter or perform. Authorization ensures bad actors will at least not have an easy time corrupting the system or even an ignorant user who accidentally sets off a command or function they shouldn’t. |
| Accounting | Gives the system information on the resources the end-user is taking up as well as authorization control. This is useful for learning where resources need to be devoted more, or if there may even be a bad actor abusing resources to try to take advantage of the system. This will also account for the different features the account was using such as file access and will enable the system to track down anyone misusing. |

**\***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 | 10/15/2023 | Initial Template | Michael Hanlon |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |
| Structured-Query-Language | SQL |