**Daulton Truett: Green Pace Security Policy**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All data read in from an input stream, both from user input and external files, must be evaluated for type correctness, length to prevent overflow, and the potential for SQL injection attempts. |
| 1. Heed Compiler Warnings | Do NOT ignore compile time warnings even if the code will run with them. Compile time warnings may often point to errors at runtime leading to security vulnerabilities and/or prevent future errors that lead to security vulnerabilities. |
| 1. Architect and Design for Security Policies | Develop code that is designed with security as a priority from beginning to end. Implementing security in the architecture and design process through the entire application lifecycle reduces the potential for security vulnerabilities to be deployed into live production. |
| 1. Keep It Simple | Code design that is overly complex may lead to hard-to-find errors and security vulnerabilities that can be difficult to identify, remediate, and may lead to further issues. Simple solutions often solve complex problems while enhancing code readability, reducing hard-to-find errors, and expediate debugging. |
| 1. Default Deny | All external data and user access should be denied by default unless already “whitelisted” to reduce the potential attack vector. By integrating a “default deny” system, malicious users and data are significantly reduced by increasing the difficulty of access and injection of malicious code. External users must be both authenticated and authorized to least privilege while external data must be validated for its integrity. |
| 1. Adhere to the Principle of Least Privilege | All users should be authorized with a level of privilege that allows them to perform their essential duties without exceeding a higher level of privilege that does not apply to their duties. |
| 1. Sanitize Data Sent to Other Systems | Ensure datasets sent to external systems are free of all secure and/or sensitive data that could be used to gain an understanding of system architecture. SQL commands must also be sanitized of any potential malicious commands attempting SQL injection. |
| 1. Practice Defense in Depth | Utilize various methods and tools concurrently to prevent, detect, test, and respond to software security vulnerabilities to reduce the overall attack vector. Defense in depth provides a broader a deeper picture of the software to decrease the likelihood that security vulnerabilities are implemented in the code, and if they are, fast and efficient remediation of the vulnerability will be deployed. |
| 1. Use Effective Quality Assurance Techniques | Software design and code architecture should be incrementally reviewed to ensure a high-quality product is delivered. Frequent code auditing ensures software vulnerabilities are not released with the final deliverable and allows for any vulnerabilities that are discovered to be fixed before release. Testing the software frequently during production produces a higher quality product at release, identifies errors, and security vulnerabilities which reduces the potential attack vector and allows for a cheaper, less complex remediation of any issues discovered. Software testing includes unit testing, stress testing, integration testing, and pen testing among others, all of which need to be utilized during and post-production. |
| 1. Adopt a Secure Coding Standard | Implementing a secure coding standard provides all developers with methodologies to prevent writing security vulnerabilities into code and permits consistency in software architecture. Should a vulnerability be found in the software, a coding standard is a blueprint to aid developers in finding and fixing the vulnerability. |

### 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 & Rationale** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | INT50-CPP: Do not cast to an out-of-range enumeration value.  When comparing values against each other it is important to ensure that the values being compared are of the same type. If casting from one type to another, make sure that if any comparisons are made, that the casting occurs post comparison to prevent unspecified values and unspecified behavior. |

| **Noncompliant Code** |
| --- |
| The code below defines three enum types, the function then casts the argument passed to the function from an integer to the enum type, which may result in a value out of bounds for the integer type. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);      if (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| Three enum types are defined and a function takes in an integer as an argument. The function compares the integer argument against the enum types to ensure it can be represented as that type, and if the check passes the integer variable is then cast to the enum type, preventing an unspecified value. |
| enum EnumType {    First,    Second,    Third  };  void f(int intVar) {    if (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

| **Principle(s):**  1 – This coding standard maps to the security principle ‘Validate input data’ because it ensures type correctness prior to attempting to cast from one type to another.  3 – This coding standard maps to the security principle ‘Architect and Design for Security Policies’ because it is anticipating future events where casting from one type to a different type may be required, such as when receiving user input, for comparison. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | cast-integer-to-enum | Partially checked |
| RuleChecker | 22.10 | cast-integer-to-enum | Partially checked |
| Polyspace Bug Finder | R2023b | CERT C++: INT50-CPP | Checks for casting to out-of-range enumeration value (rule fully covered) |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name & Rationale** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | CTR50: Guarantee that container indices and iterators are within the valid range.  When evaluating ranges on data types, the value of the index performing the range check needs to be within the expected values to not throw out-of-bounds errors. This error frequently occurs when passing user supplied data into functions that perform these checks with data types whose values fall over or under the data types ranges for accepted values. Ensuring that values used for these operations are within expected ranges for the given type prevents runtime errors. |

| **Noncompliant Code** |
| --- |
| The pos argument passed into the function is a signed int type, which allows a user to input a negative value and the function does not check for the lower bounds of the array that the table pointer is referencing. This results in an out-of-bounds error and causes runtime errors. |
| #include <cstddef>  void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| The pos argument is declared as type size\_t which prevents a suer from passing negative values, so the function checks for the upper bounds of the array size and cannot fall below the lower bounds of the array. |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {    if (pos >= tableSize) {      // Handle error      return;    }    table[pos] = value;  } |

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

| **Principle(s):**  1 – This coding standard maps to the security principle ‘Validate input data’ because it ensures that user supplied data are within valid range by first validating the data they input. It also ensures that the correct data type is used for indices and iterators.  5 - This coding standard maps to the security principle ‘Default Deny’ because when a user supplies a value to be used for an iterator or as an index for container enumeration it should be denied by default unless it falls within a valid range. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-CRT-50-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2023b | CERT C++: CTR50-CPP | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name & Rationale** |
| --- | --- | --- |
| **String correctness** | STD-003-CPP | STR50: Guarantee that storage for strings has sufficient space for character data and the null terminator.  Buffer-overflows cause abrupt runtime termination and/or severe security vulnerabilities in the form of memory overwrites and malicious code injection. Using string data types for user input over char arrays prevents overflow and truncation. |

| **Noncompliant Code** |
| --- |
| Utilizing a fixed char array buffer size that will be assigned user input with an unknown size at runtime is a cause of buffer-overflow since the user input is unbounded. |
| #include <iostream>    void getName() {    char userInput[12];    std::cin >> userInput;  } |

| **Compliant Code** |
| --- |
| Using the std::string data type for user input prevents buffer overflow because strings are not fixed length data types. |
| #include <iostream>  #include <string>    void userInput() {    std::string userInput;    std::cin >> userInput  } |

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

| **Principle(s):**  2 - This coding standard maps to the security principle ‘Heed Compiler Warnings’ because the compiler will warn against using a fixed char array buffer size for receiving user input as it is a frequent cause of buffer overflow. Using a string data type for user input will prevent unnecessary buffer overflows.  4 – This coding standard maps to the security principle ‘ Keep It Simple’ because opting for a string data type for user input is much simpler than trying to be clever and reduce a small amount of memory allocation by attempting to use a fixed char array for input. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| 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** | - Avoid overflow due to reading a not zero terminated string - Avoid overflow when writing to a buffer - Prevent buffer overflows from tainted data - Avoid buffer write overflow from tainted data - Do not use the 'char' buffer to store input from 'std::cin' |
| Polyspace Bug Finder | R2023b | CERT C++: STR50-CPP | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| RuleChecker | 22.10 | stream-input-char-array | Partially checked |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | STR53-CPP. Range check element access.  External string inputs must be accessed to ensure the input is free of potential SQL injection commands by inspecting the characters at each index. When accessing an input string, ensure that the index variable used for access is within acceptable range to prevent modification to an out-of-range object resulting in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The code below assumes that the getIndex() variable of type size\_t will be within a valid range for accessing the string while the string may not have that many elements stored in it. This approach will result in undefined behavior and may allow the SQL injection to execute. |
| #include <string>    extern std::size\_t get\_index();    void f() {    std::string SQL\_injection = “1=1”;  if(SQL\_injection[get\_index()] = “=”){  if(SQL\_injection[get\_index() – 1] == (SQL\_injection[get\_index() + 1){  // SQL injection found, handle  }  }  } |

| **Compliant Code** |
| --- |
| The below code assigns the index to be used for accessing elements of the input string to 0 and ensures that any access is less than the length of the string, ensuring that access cannot be made outside of the acceptable range and any potential SQL injection commands are found. |
| include <string>  std::string SQL\_injection;  std::cin >> SQL\_injection;  // assume “1=1” is passed    void f() {  for (int i = 0; i < SQL\_injection.size(); i++){  if(SQL\_injection[i] = “=”){  if(SQL\_injection[i – 1] == SQL\_injection[i + 1]){  // SQL injection found, handle  }  }  }  } |

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

| **Principle(s):**  **7 -** This coding standard maps to the security principle ‘Sanitize Data Sent to Other Systems’ because it ensures that data received by users will be free of potentially malicious SQL injection commands.  8 - This coding standard maps to the security principle ‘Practice Defense in Depth’ because it gaurds against potential SQL injection even after the assumption that the users passing data into a Database have been authenticated and authorized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure | NA |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR-53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2023b | CERT C++: STR53-CPP | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |
| CodeSonar | 8.0p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | MEM50-CPP. Do not access freed memory.  When dynamically allocating memory, it is critical to ensure that deallocating memory should occur only after any reference to the object is no longer required and that any pointer dereferencing occurs prior to allocation. Failure to properly deallocate dynamically allocated memory results in memory leak and referencing deallocated memory can result in exploitable vulnerabilities if access results in a write-after-free. |

| **Noncompliant Code** |
| --- |
| The pointer is dereferenced after the memory has been deallocated. This could result in an exploitable vulnerability to run arbitrary code if a write-after-free occurs with permissions of the process. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| Memory should be deallocated only after any reference to it is no longer needed. |
| #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 - This coding standard maps to the security principle ‘ Heed Compiler Warnings’ because the compiler will warn against code that accessing free memory.  10- This coding standard maps to the security principle ‘Adopt a Secure Coding Standard’ because it provides developers with the knowledge to prevent a common but severe security vulnerability. Adopting this coding standard creates consistency amongst developers to ensure memory that has been freed will not be accessed after that point. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use | NA |
| CodeSonar | 8.0p0 | ALLOC.UAF | Use after free |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Polyspace Bug Finder | R2023b | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | ERR50-CPP. Do not abruptly terminate the program.  Using assertions at runtime can cause the program to abort abruptly if the assertion fails. Therefore, assertions should only be used post and/or prior to runtime during testing or if a critical error is encountered. |

| **Noncompliant Code** |
| --- |
| If a user inputs a password that fails outside of the upper and lower bounds of the range, then the assertion fails and aborts the program. |
| #inlcude <string>  Int main(){  std::string password;  std::cout << “Input your password: ”;  std::cin >> password;  // input is “pass”  assert(password.size() >= 8 && password.size() <= 20);  // program calls abort() and ends  return 0;  } |

| **Compliant Code** |
| --- |
| This below code allows for the program to continue if a user inputs a string that is outside of our expected condition. Custom error checking prevents abrupt termination during situations that are not critical errors but still allows us to ensure that expected conditions are met. |
| #inlcude <string>  Int main(){  std::string password;  std::cout << “Input your password: ”;  std::cin >> password;  while(password.size() < 8 && password.size() > 20){  std::cout << “ERROR! Password must be at least 8 characters and no  more than 20 characters”;  std::cin >> password;  }  return 0;  } |

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

| **Principle(s):**  6 - This coding standard maps to the security principle ‘ Adhere to the Principle of Least Privilege’ as it permits the use of assert() during testing to ensure that users are authorized at a level of least privilege.  9 - This coding standard maps to the security principle ‘Use Effective Quality Assurance Techniques’ because assertions are frequently used during software testing to ensure an input produces the expected output or a particular state of the software is as expected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use | Partially checked |
| CodeSonar | 8.0p0 | BADFUNC.ABORT  BADFUNC.EXIT | use of abort  Use of exit |
| Polyspace Bug Finder | R2023b | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |
| RuleChecker | 22.10 | stdlib-use | Partially check |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name & Rationale** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | ERR51-CPP. Handle all exceptions.  Failing to handle exceptions results in abrupt abnormal program termination which is a common method for denial-of-service attacks. When abnormal program termination occurs, allocated memory may not be freed and/or closed resulting in memory leaks. |

| **Noncompliant Code** |
| --- |
| The code below does not utilize any methods to catch exceptions that may be thrown, resulting in abnormal program termination. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| main() employs a try-catch block with f() to catch any exceptions thrown with throwing\_func(), ensuring that exceptions are handled, resources can be managed, and preventing abrupt abnormal program termination. |
| 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.**

| **Principle(s):**  3- This coding standard maps to the security principle ‘Architect and Design for Security Policies’ as handling all exceptions puts security as a priority. Leaving unhandled exceptions in the software leads to program termination which could result in potential DDoS attack and also provides malicious actors knowledge on the software architecture and ways to exploit it.  9 - This coding standard maps to the security principle ‘Use Effective Quality Assurance Techniques’ because it ensures the software operates without interruption at runtime when exceptions are thrown. It also ensures developers re-visit areas of the code where exceptions are thrown which allows frequent code reviews and audits. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | - Always catch exceptions  - Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2023b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Sensitive data | STD-008-C | MSC41-C. Never hard code sensitive information.  This C standard applies to C++ development along with any other chosen language for development. Hardcoding sensitive information may result in malicious users obtaining access to the data if they have access to the executable or dynamic library files. In the event of memory leak, this data may also be obtained by malicious users and/or shared unintentionally. |

| **Noncompliant Code** |
| --- |
| The code below hardcoded the username and password into the application. If a malicious actor obtained access to the source code, they have instant and unobstructed access to the system. |
| #include <string>  bool authenticateUser(std::string username, std::string password);  Int main(){  std::string userName = “User”;  std::string userPassword = “qwerty123”;    If(!authenticateUser(userName, userPassword)) {  // deny access  }else{  // grant access  }  return 0;  } |

| **Compliant Code** |
| --- |
| This code takes the username and password in as input dynamically from a user so the sensitive data is not hard coded into the application. |
| #include <string>  bool authenticateUser(std::string username, std::string password);  Int main(){  std::string userName;  std::string userPassword;  std::cout << “Enter your username and password: ”;  std::cin >> userName, userPassword;    if(!authenticateUser(userName, userPassword)) {  // deny access  }else{  // grant access  }  return 0;  } |

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

| **Principle(s):**  8 **-** This coding standard maps to the security principle ‘Practice Defense in Depth’ because it reduces the attack vector by adding a layer of security over sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Hardcoded Authentication  Hardcoded DNS Name  Hardcoded Crypto Key  Hardcoded Crypto Salt  Hardcoded Seed in PRNG |
| Parasoft C/C++ test | 2023.1 | CErT\_C-MSC41-a | Do not hard code string literals |
| Polyspace Bug Finder | R2023b | CERT C: Rule MSC41-C | Checks for hard coded sensitive data (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Uninitialized memory | STD-009-CPP | EXP53-CPP. Do not read uninitialized memory.  Local and automatic variables that are declared and read before initialization will hold unexpected values. Reading uninitialized memory results in undefined behavior that may allow a malicious actor to run arbitrary code. |

| **Noncompliant Code** |
| --- |
| The variable “I” is declared but not initialized. The program then attempts to read the uninitialized memory, resulting in undefined behavior. |
| #include <iostream>    void f() {    int i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| The variable “I” is declared and initialized before being read by the program. |
| #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.**

| **Principle(s):**  2 - This coding standard maps to the security principle ‘Heed Compiler Warning’ because the compiler will warn when the code is attempting to read uninitialized memory.  3 - This coding standard maps to the security principle ‘Architect and Design for Security Policies’ as following this coding standard will prevent security vulnerabilities by ensuring developers initialize memory before attempting to read from it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | uninitialized-read | Partially checked |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug Finder | R2023b | CERT C++: EXP53-CPP | Checks for:   * Non-initialized variable * Non-initialized pointer   Rule partially covered. |
| RuleChecker | 22.10 | uninitialized-read | Partially checked |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object lifetime | STD-010-CPP | EXP54-CPP. Do not access an object outside of its lifetime.  Object oriented programming is commonly used in applications and therefore require an understanding on how to use object along with object lifetime. Attempting to access object members outside of the objects lifetime results in undefined behavior that can lead to security vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The pointer “s” is attempting to call a non-static member function before the lifetime of the pointer is established. |
| struct S {    void mem\_fn();  };    void f() {    S \*s;    s->mem\_fn();  } |

| **Compliant Code** |
| --- |
| The lifetime of the pointer “s” has been properly established allowing for the pointer to call the non-static member function correctly. |
| struct S {    void mem\_fn();  };    void f() {    S \*s = new S;    s->mem\_fn();    delete s;  } |

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

| **Principle(s):**  3 - This coding standard maps to the security principle ‘Architect and Design for Security Policies’ as following this coding standard will prevent security vulnerabilities by ensuring developers initialize memory before attempting to read from it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | return-reference-local  dangling\_pointer\_use | Partially checked. |
| CodeSonar | 8.0p0 | IO.UAC  ALLOC.UAF | Use after close.  Use after free. |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | - Do not use resources that have been freed.  - The address of an object with automatic storage shall not be returned from a function.  - The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist. |
| Polyspace Bug finder | R2023b | CERT C++: EXP54-CPP | Checks for:   * Non-initialized variable or pointer * Use of previously freed pointer * Pointer or reference to stack variable leaving scope * Accessing object with temporary lifetime   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.

[Insert your written explanations here.]

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Medium | 4 | 3 |
| STD-002-CPP | High | Likely | High | 9 | 2 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Unlikely | Medium | 6 | 2 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Probable | Medium | 4 | 3 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-C | High | Probable | Medium | 12 | 1 |
| STD-009-CPP | High | Probable | Medium | 12 | 1 |
| STD-010-CPP | High | Probable | High | 6 | 2 |

### 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 in rest is data that has been encrypted prior to it being stored, archived, or is in an inactive state. Data at rest is less vulnerable than data at flight or data in use, but that data must still be encrypted as it can be compromised by malicious actors. Stored data on a device or drive is vulnerable to physical theft, but if the data is encrypted, the data is still secured from further exploitation. Inactive data that is archived in a database may not be well known to malicious actors or have an attack vector as wide as data at flight or in use, but if discovered, ensuring it is encrypted will slow and may prevent further exploitation. |
| Encryption at flight | Encryption at flight is data that has been encrypted prior to its movement when using services such as email, messaging apps, and remote collaboration platforms. Data at flight is vulnerable to attack since it is travelling over a network whether private or public. To ensure that data is secure at flight, data must be encrypted prior to its movement and can be strictly enforced with automated tools that check that encryption has been implemented before any data leaves its current location over a network. One of the main vulnerabilities of data at flight without encryption is a ‘main-in-the-middle’ attack, where an attacker intercepts the data and is then able to modify, delete, redirect, and/or take the data. If utilizing a public network such as public internet, a VPN should be enforced to further protect the encrypted data at flight but is NOT to be a replacement for encryption. |
| Encryption in use | Encryption in use is data that has been previously encrypted but must be decrypted for use. To ensure security of data in use, strict authentication protocols must be enforced since the data will be decrypted and available for the authenticated user. Another aspect of security to consider for data in use is strict authorization of least privilege for authenticated users. Enforcing authorization with least privilege reduces the attack vector of malicious actors in the event they gain access to a trusted users account. The greatest security risk that data in use suffers from is malicious actors gaining access to trusted users authentication information. 2FA or multi-factor authentication must be employed in the system to prevent malicious actors from gaining access to trusted user access. Of final note, when data is decrypted for use by a user, it must be encrypted again when use has ceased every time whether it is done manually or automatically. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of a user proving their identity to gain access to a system. The process of authenticating a user includes what a user knows such as a username and password, what a user has such as a keycard or 2FA device input, and who a user is such as biometric data. Restricting access via user authentication provides an important layer of security as it prevents malicious actors from freely obtaining company data and information about system infrastructure. All access to company data and infrastructure must be obtained strictly through 2FA which will include username/email and password and physical device input that will be prompted during every account login and prior to any safety critical operation such as movement of data outside of the company infrastructure or network. |
| Authorization | Authorization is the level of access and privilege an authenticated user is granted to perform their essential duties. Authorization is an essential security protocol as it limits the level of access and privilege users have to critical data and system infrastructure so that the attack vector is limited in the instance of a security breach. Authorization will be implemented to grant users the level of least privilege needed to perform their essential job duties at the time of account creation by the account administrator and will only be modified if and when required by the account administrator. |
| Accounting | Accounting is the process of monitoring user activity and the activity on company infrastructure. Company infrastructure will be monitored to ensure authentication protocols such as 2FA are properly utilized and implemented for all operations requiring it, such as login attempts and data access. Authenticated user activity will be monitored to ensure that the level of user authorization is stable and consistent. This will also help to ensure attempts to access unauthorized data/infrastructure are detected and mitigated as it could be a sign of a malicious actor gaining access to a user account. Accounting will monitor the state of the database(s) used by the company to ensure that all access and changes to any database is permitted by authenticated users with the level of authorization allowed. |

**\***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 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 | 01/28/2024 | Implemented the 10 security principles and 10 coding standards. | Daulton Truett | Daulton Truett |
| 1.2 | 02/18/2024 | Finalized security policy. | Daulton Truett | Daulton Truett |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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