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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

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

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

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

[Project One 15](#_z337ya)

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

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

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

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

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

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validating input is an important part of detecting potential injection or other types of attacks. When input is received from a trusted or untrusted source, validating the input data can catch malicious input and will help reduce negative impact from potential software vulnerabilities - known or unknown. Data that is provided from external sources could include SQL injections, command arguments, or other variables that could impact software without the appropriate defenses in place. |
| 1. Heed Compiler Warnings | When compiling your code it is important to heed each and every warning that is brought up. A missing line of “best practice code” for security could open the entire code to one or more vulnerabilities to be exploited. It is important to use both static and dynamic testing with any warnings to catch any vulnerability points or amend code to build code with the absolute minimum amount of exploitable points possible. |
| 1. Architect and Design for Security Policies | Before even starting to create your code, it is integral to take into account the requirements of the code being written as well as the appropriate secure design policies to properly implement and enforce security from end-to-end. |
| 1. Keep It Simple | Keeping it simple means that when you write your code it’s important to reduce the steps and actions where possible. The more code you write out, the more opportunity for errors, vulnerabilities, and structural mistakes. If one line of code can accurately and dependably do what 20 lines of code could do - and is still secure and meets best practices - it is best to use the one line of code with proper comments. |
| 1. Default Deny | When in doubt, keep it out! With this principles we are eschewing the, “Better to act first and ask for permission later” concept and defaulting to inaction/nonaccess until approved. One errant access is all a hacker needs to launch an attack so by committing to keeping access limited to only those who have been granted permission, and later allowing permissions and/or access as needed, there are less chances for malicious access. Standardizing this expectation via team and within the code will provide the best layer of security for this principle. |
| 1. Adhere to the Principle of Least Privilege | With the Principle of Least Privilege, we are piggy backing off of the last principle and essentially allowing only the bare minimum access/privileges required to complete a process. If the access is not necessary and the code/person can complete the task without accessing other databases or code, then they don’t need the privilege to complete the task. This reduces both malicious and accidental attacks by limiting the accessibility of those other areas. |
| 1. Sanitize Data Sent to Other Systems | If data is being sent to other systems, it’s important to sanitize (aka clean up) the data to reduce information being sent across complex or simple systems. This helps reduce the likelihood and ability of injection attacks and further validates the information being received. |
| 1. Practice Defense in Depth | Using DiD allows for there to be multiple catch points if something does go awry when writing code or if testing doesn’t entirely catch an error or vulnerability. If one layer fails, another will catch it, if two layers fail, the third will catch it, and so on. Even if there is one vulnerability it does not have to mean the entire destruction of the system. |
| 1. Use Effective Quality Assurance Techniques | By practicing quality assurance techniques there is a high chance of catching issues, identifying vulnerabilities, and eliminating them. Using coding best practice alongside static and dynamic testing, secondary reviews, and external reviews to ensure quality at every stage will result in a more secure system overall. |
| 1. Adopt a Secure Coding Standard | Last, it is vital to follow a secure coding standard for whatever language being used to code. By adopting this standard across the board, it can be assured that the teams, whether working together or on entirely separate projects, are creating software/programs/systems that are expected to meet the same standards and be read in the same format. These can then be easily cross referenced for easy testing and better comprehension when vulnerabilities do occur and improve future code. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not write syntactically ambiguous declarations |

| **Noncompliant Code** |
| --- |
| Noncompliant code would be code that is written in a syntax that is ambiguous and could be interpreted as an expression statement or a declaration, not decisively one or the other. |
| #include <mutex>    static std::mutex m;  static **int** shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex>(m);  shared\_resource += 42;  } |

| **Compliant Code** |
| --- |
| Compliant code is clearly written and there is no confusion on whether the code written is an expression statement or a declaration. |
| #include <mutex>    static std::mutex m;  static **int** shared\_resource;    void increment\_by\_42() {  std::unique\_lock<std::mutex> lock(m);  shared\_resource += 42;  } |

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

| **Principles(s):** Heed Compiler Warnings: If compiler warnings pop up they should be reviewed thoroughly. Ambiguous syntax for expressions that may be considered a declaration will be defaulted to a declaration. Always heed the compiler warnings! |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **LANG.STRUCT.DECL.FNEST** | Nested Function Declaration |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **296 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-DCL53-a**  **CERT\_CPP-DCL53-b**  **CERT\_CPP-DCL53-c** | Parameter names in function declarations should not be enclosed in parentheses  Local variable names in variable declarations should not be enclosed in parentheses  Avoid function declarations that are syntactically ambiguous |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Avoid information leakage when passing a class object across a trust boundary |

| **Noncompliant Code** |
| --- |
| Noncompliant code includes sensitive data being passed across a trust boundary or potentially leaked when copying to user space. |
| #include <cstddef>    struct test {  **int** a;  **char** b;  **int** c;  };    // Safely copy bytes to user space  extern **int** copy\_to\_user(void \*dest, void \*src, std::**size\_t** size);    void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

| **Compliant Code** |
| --- |
| Compliant code serlializes the structure data prior to allowing it to be transferred across a trust boundary or copied to user space. |
| #include <cstddef>  #include <cstring>    struct test {  **int** a;  **char** b;  **int** c;  };    // Safely copy bytes to user space.  extern **int** copy\_to\_user(void \*dest, void \*src, std::**size\_t** size);    void do\_stuff(void \*usr\_buf) {  test arg{1, 2, 3};  // May be larger than strictly needed.  unsigned **char** buf[sizeof(arg)];  std::**size\_t** offset = 0;    std::**memcpy**(buf + offset, &arg.a, sizeof(arg.a));  offset += sizeof(arg.a);  std::**memcpy**(buf + offset, &arg.b, sizeof(arg.b));  offset += sizeof(arg.b);  std::**memcpy**(buf + offset, &arg.c, sizeof(arg.c));  offset += sizeof(arg.c);    copy\_to\_user(usr\_buf, buf, offset /\* size of info copied \*/);  } |

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

| **Principles(s):** Validate Input Data: Any data that is being transferred across a trust boundary should be validated and properly accepted by both parties. This may mean encryption in use or other potential options. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **MISC.PADDING.POTB** | Padding Passed Across a Trust Boundary |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-DCL55-a** | A pointer to a structure should not be passed to a function that can copy data to the user space |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Use valid references, pointers, and iterators to reference elements of a basic\_string |

| **Noncompliant Code** |
| --- |
| noncompliant code uses an invalidated reference, pointer, or iterator and can result in undefined behavior. This means that code may act in unexpected ways and result in confusing errors. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

| **Compliant Code** |
| --- |
| Compliant code uses appropriate calls to direct code function without a loop or unreferenced action being impacted. This may be implementing an iterator or utilizing a different reference, pointer, or iterator overall. |
| #include <string>    void f(const std::string &input) {  std::string email;    // Copy input into email converting ";" to " "  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {  loc = email.insert(loc, \*i != ';' ? \*i : ' ');  }  } |

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

| **Principles(s):** Validate Input Data: If you are utilizing data that is not correct it may cause breakages in the system almost immediately or down the line due to irrelevant or inaccurate references. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.0p0 | **ALLOC.UAF** | Use After Free |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-STR52-a** | Use valid references, pointers, and iterators to reference elements of a basic\_string |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Use valid format strings and input validation. |

| **Noncompliant Code** |
| --- |
| Noncompliant code does not impose appropriate limits to ensure that SQL injections and other types of hacking attempts can not be implemented via an input method. |
| **#include <stdio.h>**    **void func(void) {**  **const char \*error\_msg = "Resource not available to user.";**  **int error\_type = 3;**  **/\* ... \*/**  **printf("Error (type %s): %d\n", error\_type, error\_msg);**  **/\* ... \*/**  **}** |

| **Compliant Code** |
| --- |
| Compliant code uses the correct arguments for input and output to ensure that all items match and meet limits specified in the code and/or react as expected to any actions. |
| #include <stdio.h>    void func(void) {  const **char** \*error\_msg = "Resource not available to user.";  **int** error\_type = 3;  /\* ... \*/  **printf**("Error (type %d): %s\n", error\_type, error\_msg);    /\* ... \*/  } |

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

| **Principles(s):** Validate Input Data: If data is not appropriately validated and even size restricted runtime errors, buffer overflows, or even SQL injection attacks can become a very real threat to the system functioning the way it should. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-FIO47** | Fully implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.0p0 | **IO.INJ.FMT**  **MISC.FMT**  **MISC.FMTTYPE** | Format string injection  Format string  Format string type error |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **PW** | Reports when the number of arguments differs from the number of required arguments according to the format string |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Free dynamically allocated memory when no longer needed |

| **Noncompliant Code** |
| --- |
| Noncompliant code may not attempt to appropriately free allocated memory prior to the end of the lifetime of the last pointer. |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    **int** f(void) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  if (text\_buffer == NULL) {  return -1;  }  return 0;  } |

| **Compliant Code** |
| --- |
| Complaint code will deallocated the pointer by calling to free() by the end of the lifetime of the last pointer. |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    **int** f(void) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);  if (text\_buffer == NULL) {  return -1;  }    **free**(text\_buffer);  return 0;  } |

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

| **Principles(s):** Architect and Design for Security: Data that is no longer in used, holding onto no longer valid pointers, or taking memory that could be freed should be freed to allow systems to run smoother and to preform system-level accounting. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **leakReturnValNotUsed** | Doesn't use return value of memory allocation function |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022a | [CERT C: Rule MEM31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem31c.html) | Checks for memory leak (rule fully covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| Noncompliant code uses assert to check a property or structure that is necessary for the code to behave correctly. This can result in code behaving poorly, throwing errors, or cause run time errors. |
| #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** |
| --- |
| Compliant code only uses assert in the event that potentially incorrect assumptions need verified. |
| #include <assert.h>    struct timer {  unsigned **char** MODE;  unsigned **int** DATA;  unsigned **int** COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned **char**) + sizeof(unsigned **int**) + sizeof(unsigned **int**),  "Structure must not have any padding"); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques: Validating information through static assertions during testing will allow a more stable and quality product to be provided with minimal error throws. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| Noncompliant code does not appropriately address an exception that is thrown. This may result in a lack of clarity in the unexpected or poor behvaior of a program and result in undetected vulnerabilities or issues as the code is developed and later maintained. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    **int** main() {  f();  } |

| **Compliant Code** |
| --- |
| Compliant code implements a handling action when an error or exception is thrown. This is done often with an error message and terminating the program to remove any potential issues resulting from continuing the program without the appropriate resolution to the exception. |
| **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):** Use Effective Quality Assurance Techniques: Well written code with all exceptions handled sets up the future of the security of the code by ensuring that the code handles each and every potential options with communication or notes, and even can prevent attacks down the line from events like injection attacks that may have passed a buffer but would be caught by a character limit exception throw. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-ERR51-a**  **CERT\_CPP-ERR51-b** | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-008-CPP] | Use valid iterator ranges |

| **Noncompliant Code** |
| --- |
| Noncompliant code does not outline an appropriate iterator range and may compare each equally or continue to iterate continuously. This may result in undefined behavior and unexpected errors or overloading. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| Compliant code utilizes the expected order to properly iterate values as they are passed to ensure there is no overload or redundancy in continued iterations. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<**int**> &c) {  std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):** Architect and Design for Security: Utilizing proper iterators will allow for quality code to be passed on and reduce redundancies, overloads, or issues in code compilation and runtime errors down the line. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.0p0 | **LANG.MEM.BO** | Buffer Overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-CTR53-a**  **CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range  Do not compare iterators from different containers |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-009-CPP] | Write constructor member initializers in the canonical order |

| **Noncompliant Code** |
| --- |
| Noncompliant code initializes constructors out of the order they are called in. This may be via inline reference or expected order in future code. |
| class C {  **int** dependsOnSomeVal;  **int** someVal;    public:  C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

| **Compliant Code** |
| --- |
| Compliant code initializes constructors in the canonical order as expected for code implementation. |
| class C {  **int** someVal;  **int** dependsOnSomeVal;    public:  C(**int** val) : someVal(val), dependsOnSomeVal(someVal + 1) {}  }; |

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

| **Principles(s):** Architect and Design for Security: Well written code includes setting up the entire code with a strong base and setting initializers in order of use so that code is easy to follow and amend as future security patches or code changes are needed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.0p0 | **LANG.STRUCT.INIT.OOMI** | Out of Order Member Initializers |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-OOP53-a** | List members in an initialization list in the order in which they are declared |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: OOP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcoop53cpp.html) | Checks for members not initialized in canonical order (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Exceptions | [STD-010-CPP] | Handle all exceptions thrown before main() begins executing |

| **Noncompliant Code** |
| --- |
| Noncompliant code fails to handle all exceptions prior to the execution of main() which may result in unexpected behavior, code breakage, or security concerns. |
| struct S {  S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| Compiant code attempts to address any potential exceptions that could be thrown prior to the implementation of main() and terminates the program prior to main() executing to reduce major crashing or breakage in the full run of the code. |
| struct S {  S() noexcept(false);  };    S &globalS() {  try {  static S s;  return s;  } catch (...) {  // Handle error, perhaps by logging it and gracefully terminating the application.  }  // Unreachable.  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques/Practice Defense in Depth: Iterating through the code and ensuring that exceptions are handled before main means that the essential run of the file remains clean and if there are any breaks they are thrown before initiating the entire program which could result in larger damage depending on the system and coding. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.2 | **CERT\_CPP-ERR58-a** | Exceptions shall be raised only after start-up and before termination of the program |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022a | [CERT C++: ERR58-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr58cpp.html) | Checks for exceptions raised during program startup (rule fully covered) |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-err58-cpp | Checked by clang-tidy |

### 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.

For automation it is important that security testing is automated at a physical and technical level. Utilizing threat modeling, risk management, and various tools of the trade end-to-end to automate functions the system can be created stronger through vigorous testing both dynamic, static, and beyond concept. The step to modify DEvOps to better address security is the DevSecOps model. DevSecOps was born from the realization that the DevOps model didn’t accurately address all security concerns that were required as the industry continued to grow and digital security became a larger concern. By choosing to create a new model, the industry was able to introduce security at the beginning and throughout the process model and automate it through and through instead of trying to piece it into a random portion of the DevOps model system. By following the latest model, applications, software, and systems are meant to be secure and risk-managed well before delivery to the end user. By having security in mind from end-to-end a better, safer, and more secure product can be delivered consistently - assuming policies are well taught, updated, and continuously implemented.

### 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 | Low | Unlikely | Medium | P2 | L3 |
| STD-002-CPP | Low | Unlikely | High | P1 | L3 |
| STD-003-CPP | High | Probable | High | P6 | L2 |
| STD-004-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-005-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-010-CPP | Low | Likely | Low | P9 | L2 |

### 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 | Encrypting data at rest means that when data is not being utilized it is still being stored with a secure encryption to prevent unused but important data from being left out from secure practices. In the case of the DeepRoot Analytics security breach, even if they weren’t actively using that data, Encryption at Rest would’ve meant that the data would’ve been secure even during a breach. |
| Encryption at flight | This is the process of encryption when information is being sent/received on any level. By encrypting during the transfer process, even if it is not encrypted during rest, the information can be protected from interfering parties. |
| Encryption in use | Encrypting data during use means that the data is actively being encrypted, decrypted, read, and re-encrypted again as it is being utilized by one or more systems. By encrypting the data in motion we can ensure secure data transfer and operation with minimal to no vulnerability. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the act of providing credentials into a system, often a username or user ID, which is then compared to a database on the backend that may also request matching passwords, 2-step verifcation or other potential ways to verify your person during accessing the system. In the event of a failed authentication process a user who may not be allowed into the system or means the system harm can be effectively refused access. |
| Authorization | Authorization is another step after authentication, but still on the backend, where access to certain part of the system is restricted based on your authorization level. If you are not authorized to enter the system and review specific information, even if you can log into parts of the system, you will not be able to see information you are not approved to see. Database access and management should not just be handled by any party. |
| Accounting | Accounting is the upkeep of a system and includes removing old account information, ensuring the right parties are utilizing or updating their credential, and unused access is regularly removed to reduce access and touch points where they are not necessary. |

**\***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 | 05/18/2022 | Initial Template | Teodoir O’Ceallaigh | O’Ceallaigh |
| 1.1 | 05/22/2022 | Module 3 | Teodoir O’Ceallaigh | O’Ceallaigh |
| 1.2 | 06/12/2022 | Module 6 | Teodoir O’Ceallaigh | O’Ceallaigh |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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