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



# Green Pace Secure Development Policy

# Miranda Putnam

## 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 | Data input from all untrusted sources needs to be validated. Using appropriate input validation methods will ensure that systems remain free of most software vulnerabilities. It is best to remain skeptical of external data sources, including but not limited to, command line arguments, network interfaces, environmental variables, and user controlled files. |
| 1. Heed Compiler Warnings | Pay close attention to compiler warnings and eliminate them by editing the code until satisfactory. Always compile code using the highest warning level available. Static and dynamic analysis tools and methods can be used to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Create an architecture that makes sense for your software and design with security in mind so your software implements and enforces security policies. Subsystems can be used as necessary for a system with different roles and privileges. |
| 1. Keep It Simple | Keep your system, code, and design as simple and small as possible. As code becomes more complex, the likelihood of errors increases, as well as the need for more complex security measures. |
| 1. Default Deny | Access to a system should be denied by default, and access should be granted based on permission rather than exclusion. Protections put in place regarding access should identify conditions under which access is permitted. |
| 1. Adhere to the Principle of Least Privilege | Each process within the system should be accomplished using the least set of privileges necessary. If privilege is elevated, the process should execute in the least amount of time possible. This is all to ensure any and all attackers have limited access to the system and limited time to perform the attack. |
| 1. Sanitize Data Sent to Other Systems | Data sent to complex subsystems needs to first be sanitized in order to avoid attacks. This includes command shells, relational databases, and commercial off-the-shelf (COTS) components. This is necessary in order to prevent attacks such as SQL and other injections that exploit unused functionality in these components. |
| 1. Practice Defense in Depth | Utilize multiple defensive security strategies within different levels of the system in order to prevent attacks. This ensures a layered security strategy so that if one layer fails, there is another ready to protect the system and minimize any damage. An example of this strategy includes using both physical and technological security measures to defend the system in depth. |
| 1. Use Effective Quality Assurance Techniques | Use effective quality assurance techniques and methods in order to identify and eliminate vulnerabilities. Regular testing and audits of code should be performed and incorporated into an effective quality assurance program. Independent security checks should be performed in order to catch invalid assumptions and other security issues. |
| 1. Adopt a Secure Coding Standard | Develop and apply a secure coding standard for your specific development language and platform. |

### 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** | **Do not define a C-style variadic function.** |
| --- | --- | --- |
| **Data Type** | DCL50-CPP | Variadic functions accept a variable number of arguments from a caller. The use of a C-style ellipsis within a variadic function can be hazardous, as it does not have the means to check the type safety of arguments that are passed into the function or to check that the number of arguments that are passed match the semantics of the function definition. Do not define C-style variadic functions, as they can be hazardous. |

| **Noncompliant Code** |
| --- |
| This example is noncompliant because it uses C-style ellipsis within a variadic function to add multiple integers together. The function will read arguments until a 0 value is found. Using the ellipsis instead of defining 0 as an argument results in undefined behavior that can be hazardous. |
| #include <cstdarg>    int add(int first, int second, ...) {    int r = first + second;    va\_list va;    va\_start(va, second);    while (int v = va\_arg(va, int)) {      r += v;    }    va\_end(va);    return r;  } |

| **Compliant Code** |
| --- |
| This example is compliant because it uses a variadic function with a function parameter pack to implement the add() function. This works because it will not result in undefined behavior if the parameter list is not terminated with 0. |
| #include <type\_traits>    template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }    template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {    return f + add(rest...);  } |

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

| **Principles(s):** The principles that most closely maps to this standard are 1 and 4. This is because the argument needs to be clearly defined and validated in order to avoid hazardous behavior, and the arguments should be kept clear and simple. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Function-ellipsis | Fully checked. |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-DCL50-a | Functions shall not be defined with a variable number of arguments. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | INT50-CPP | When using enumerations, the range of values that can be represented by either form of enumeration can include enumerator values that are not specified by the enumeration itself. In order to avoid operating with unspecified values, the arithmetic value being used has to be in the range of the values the enumeration can represent. Perform checks for out-of-range values before casting the expression. |

| **Noncompliant Code** |
| --- |
| This example is noncompliant because it attempts to check whether a given value is within the range of the enumeration, but fails to do so before casting the enumeration type. If the enumeration were to result in an unspecified value and used within the if statement, it would result in unspecified behavior. |
| enum EnumType {    First,    Second,    Third  };    void f(int intVar) {    EnumType enumVar = static\_cast<EnumType>(intVar);      if (enumVar < First || enumVar > Third) {      // Handle error    }  } |

| **Compliant Code** |
| --- |
| This code is compliant because it ensures that the value can be represented by the enumeration type before performing the conversion. This ensures that both the value and resulting behavior will not be unspecified. |
| 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.**

| **Principles(s):** The principles that map to this standard are 1 and 9. The data value should be checked and validated to ensure it can be represented by the enumeration. This also applies to quality assurance, as it checks and eliminates any vulnerabilities before they occur. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | Data should not be copied to a buffer that is not large enough to hold the data, as it could lead to buffer overflow. To avoid buffer overflow, use methods to ensure that the destination is large enough to hold the size of the data. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because it uses no methods to ensure the input remains within the bounds of the buffer, which could result in buffer overflow and erratic behavior. |
| #include <iostream>    void f() {    char buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| This example is compliant because it utilizes strings as opposed to a bounded array in order to ensure that buffer overflow will not occur. |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** The principle that maps to this standard is number 1. The input data for a string should be validated to ensure it is not too large for the string to hold, avoiding buffer overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | MISC.MEM.NTERM  LANG.MEM.BO LANG.MEM.TO | * No space for null terminator * Buffer overrun * Type overrun |
| Parasoft C/C++ test | 2021.2 | 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'. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | FIO30-C | Do not call a formatted input/output function with a format string containing a tainted value. Attackers can take advantage of this and use it to access and change the system contents. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because the function used accepts the name of the user as a string referenced by the user. This is untrusted data that originates from an unauthenticated user. That data is then passed into the call to fprintf(), resulting in a vulnerability. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fprintf(stderr, msg);    free(msg);  } |

| **Compliant Code** |
| --- |
| This code example is compliant because it replaces the fprintf() call with fputs(), which will output the information without evaluating the contents. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {    int ret;    /\* User names are restricted to 256 or fewer characters \*/    static const char msg\_format[] = "%s cannot be authenticated.\n";    size\_t len = strlen(user) + sizeof(msg\_format);    char \*msg = (char \*)malloc(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }    fputs(msg, stderr);    free(msg);  } |

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

| **Principles(s):** The principles that map to this standard are 1 and 7. Input should always be validated to avoid tainted values, and data should be sanitized if possible before calling a function with a tainted value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | IO.INJ.FMT  MISC.FMT | * Format string injection * Format string |
| Parasoft C/C++ test | 2021.2 | **CERT\_C-FIO30-a CERT\_C-FIO30-b CERT\_C-FIO30-c** | * Avoid calling functions printf/wprintf with only one argument other than string constant. * Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable. * Never use unfiltered data from an untrusted user as the format parameter. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | MEM50-CPP | Do not evaluate a pointer that has been deallocated by a memory management function, as this will result in undefined behavior. These are called dangling pointers, and accessing them can result in vulnerabilities within the system. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because it dereferences s after it has already been deallocated. This can be exploited by running code with the permissions of the vulnerable process. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| This code is compliant because it deallocates s after it is still required for use instead of before. |
| #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):** The principles that map to this standard are 3 and 8. Both of these pertain to security and defensive strategies to ensure security, and ensuring your team knows not to access freed memory because it can lead to ungranted permissions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Handle all exceptions thrown before main() begins executing** |
| --- | --- | --- |
| **Assertions** | ERR58-CPP | If an uncaught exception is thrown before main() is executed, or after main() has finished executing, there are no more opportunities to handle the exception and it will result in implementation-defined behavior. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because the constructor for S could throw an uncaught exception when globalS is constructed during startup. |
| struct S {    S() noexcept(false);  };    static S globalS; |

| **Compliant Code** |
| --- |
| This code example is compliant because it allows exceptions thrown during object construction to be caught by making globalS into a local variable having S execute when the function called rather than at startup. |
| 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):** The principles that map to this standard are 2 and 9. One should always heed compiler warning, such as those that pertain to exceptions, and assure quality of code by catching and handling errors early on. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program. |
| Polyspace Bug Finder | R2021b | CERT C++: ERR58-CPP | Checks for exceptions raised during program startup (rule fully covered). |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Honor exception specifications** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | If a function throws an exception other than one allowed by its exception*-*specification, it could lead to an implementation-defined termination of the program. Abnormal termination should be avoided by having functions only throw allowed exceptions. |

| **Noncompliant Code** |
| --- |
| This code is a noncompliant example because it declares the function as non-throwing of exceptions, but it is still possible for an exception to be thrown when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) noexcept(true) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| This code example is compliant because it removes the noexcept specification from the function, signifying that all exceptions are allowed. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) {    v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):** The principle that maps to this standard is number 2. The programmer should pay close attentions to compiler warnings regarding exception specifications. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2021b | CERT C++: ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Use valid iterator ranges** |
| --- | --- | --- |
| **Containers** | CTR53-CPP | Iterators must iterate over a valid range when iterating over elements of a container. The range must have a pair of iterators that refer to the start and end of the range. Do not use iterators that are invalidated or do not refer to the same container, as it could result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This is an example of noncompliant code because while it uses valid iterators for the range, the first iterator does not precede the second, which will result in undefined behavior when looping through and incrementing. |
| #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** |
| --- |
| This code is compliant because it lists the iterators in the correct order, with begin preceding the end. |
| #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):** The principle that maps to this standard is number 9. The programmer and QA team should assure code quality to ensure that code is valid and bug free in order to avoid undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 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** | **Obey the one definition rule** |
| --- | --- | --- |
| **Declarations** | DCL60-CPP | All objects within a C++ program require a single definition, as multiple definitions will result in undefined behavior of the program. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because it defines a class of the same name, S, with differing definitions for each instance. This violates the rule and results in undefined behavior. |
| // a.cpp  struct S {    int a;  };    // b.cpp  class S {  public:    int a;  }; |

| **Compliant Code** |
| --- |
| If the programmer intends to use the same name for different classes, then header files can be used to keep the same name and introduce the object into different units. |
| // S.h  struct S {    int a;  };    // a.cpp  #include "S.h"    // b.cpp  #include "S.h" |

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

| **Principles(s):** The principles that map to this standard are 2 and 4. Code should be kept simple and the programmer should heed any compiler warnings regarding multiple definitions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | LANG.STRUCT.DEF.FDH LANG.STRUCT.DEF.ODH | * Function defined in header file. * Object defined in header file. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Detect errors when converting a string to a number** |
| --- | --- | --- |
| **Error Handling** | ERR62-CPP | Many errors can arise when converting a string to a number, and these errors must be detected and addressed when performing the conversion. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because it does not take into account that the input value may not be able to be represented as an int, thus resulting in an unexpected value to be stored in the variables. |
| #include <iostream>    void f() {    int i, j;    std::cin >> i >> j;    // ...  } |

| **Compliant Code** |
| --- |
| This example is compliant because it enables exceptions to be thrown for conversion failure. |
| #include <iostream>    void f() {    int i, j;      std::cin.exceptions(std::istream::failbit | std::istream::badbit);    try {      std::cin >> i >> j;      // ...    } catch (std::istream::failure &E) {      // Handle error    }  } |

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

| **Principles(s):** The principles that map to this standard are 1 and 9. Input data should be validated to ensure it is of the correct type, and QA teams should review and test code to ensure incorrect conversions are caught and handled. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR62 | N/A |

### 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.

In order to transform DevOps into DevSecOps to automate enforcement of the standards in this policy, security measures must be defined and integrated into each portion of both pre-production and production. This ensures that security is addressed every step of the way, and that each process is compliant with the principles and standards within this policy. For pre-production, planning can be adapted to include threat modeling and security tool training, designing and building can be adapted to include IDE security plug-ins, and verification should include testing and vulnerability scanning. For the production stage, prevention, detection, response, and prediction should include security measures such as penetration tests, intrusion detection, blocking attacks, and stabilizing the system after an attack. All of these measures will ensure security is addressed every step of the way in both pre-production and production so that DevOps can become DevSecOps and conform to the standards and principles outlined in this policy.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL50-CPP | High | Probable | Medium | 12 - High | 1 |
| INT50-CPP | Medium | Unlikely | Medium | 4 - Low | 3 |
| STR50-CPP | High | Likely | Medium | 18 - High | 1 |
| FIO30-C | High | Likely | Medium | 18 - High | 1 |
| MEM50-CPP | High | Likely | Medium | 18 - High | 1 |
| ERR58-CPP | Low | Likely | Low | 9 - Medium | 2 |
| ERR55-CPP | Low | Likely | Low | 9 - Medium | 2 |
| CTR53-CPP | High | Probable | High | 6 - Medium | 2 |
| DCL60-CPP | High | Unlikely | High | 3 - Low | 3 |
| ERR62-CPP | Medium | Unlikely | Medium | 4 - Low | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is when encrypted data is stored somewhere within a system, usually protected by an added layer of security, such as a firewall. Encryption at rest should be used as an added layer of defense from attacks. Important data should be encrypted and protected at all stages of its lifecycle, even when being stored, in order to ensure only those who need access to it are able to access it. |
| Encryption in flight | Encryption in flight is when data is transferred from one location to another, whether it be through user input or moving stored sensitive data from one computer to another. Data should be encrypted when being transferred, as attackers could try and intercept the data and gain access to it. If the data is encrypted, this allows the data to remain indecipherable to attackers who do manage to intercept it, and only decipherable to those with an encryption key. |
| Encryption in use | Encryption in use is when data created or accessed is protected between the at rest and in flight stages. This ensures that data remains secure at all stages of the lifecycle. It also protects data from being accessed by an attacker who has gained access to the system. If the data is secured at all times, it won’t matter if an attacker is let into the system, as long as there are limited access to encryption keys to prevent the attacker from accessing the secure data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication ensures the correct person is being granted access to the system. It verifies the users identity and credentials. Authentication can be used in many different ways, including but not limited to: username and password, biometrics, security questions, and code verification. This is important in ensuring that the correct users are being given access to a system, and only once they are authenticated will they be able to access the system based on their access level. This applies to both new and existing users. |
| Authorization | Authorization ensures the correct level of use for specific users, in other words, what resources each user is allowed to access and use. The system can specify and allocate these permissions based on different factors, such as location, time-of-day, and frequency of log-ins. This policy is important because it ensures there are different levels of access to data based on the specified user, and that only permitted users can access and use certain things within the system. |
| Accounting | Accounting is a form of log-keeping to track what resources were accessed, when, by whom, and how they were accessed. This allows the system to keep a record of log-in session statistics, user information and methods, and any changes made. This can be used to pin down attacks and how they occurred in order to prevent them in the future, as well as for trend analysis and resource utilization information to be collected and analyzed during audits. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 01/20/2022 | Coding Standards Milestone | Miranda Putnam |  |
| 3.0 | 02/07/2022 | Final Policy Submission | Miranda Putnam |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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