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



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 | Validate input from all external and untrusted data sources. Sources such as command line arguments, user-controlled files, network interfaces, and other user input can pose a threat to security. Validating these inputs can mitigate software vulnerabilities. |
| 1. Heed Compiler Warnings | Use the highest warning level available to the compiler and adjust code accordingly to eliminate all flagged warnings in your code. Static and dynamic tool are available to detect and eliminate these warnings. |
| 1. Architect and Design for Security Policies | Keep the security policies in mind while creating and designing software architecture. Create subsystems that require certain privileges separate from other subsystems that do not require the same privileges. |
| 1. Keep It Simple | Keep the design of your software as simple and small as possible. The more complex your code becomes, the more prone it is to errors. The complexity of the code also adds to the difficulty of maintaining and protecting against errors that arise from complex code. |
| 1. Default Deny | The base access for the code should be denial of access. Access should not be granted unless conditions are met, as opposed to permission being granted unless conditions are *not* met. |
| 1. Adhere to the Principle of Least Privilege | Only the necessary set of privileges should be executed to complete a job. Additional privileges should only be granted when needed and for the shortest time required to complete the job. This limits the amount of time a potential threat has access to elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data being sent to other systems such as databases, command shells, and other commercial software components. Hackers can use data being sent to other subsystems to invoke an injection attack. The receiving subsystem does not understand the data being sent so the sanitation must be handled before it is sent. |
| 1. Practice Defense in Depth | Multiple strategies for securing our system and data should be utilized. In the case that one of the strategies fails, there are other layers of security that may prevent a threat from exploiting the software. Combining security techniques with security tools reduces the likelihood that vulnerabilities remain at the time of deployment. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques are vital to detecting and removing software vulnerabilities. Implementing testing techniques and source code audits can find security vulnerabilities and lead to more secure software. |
| 1. Adopt a Secure Coding Standard | Developing and applying a secure coding standard allows the team to all have a similar focus when it comes to security, when creating software. Adhering to this standard the team will find, correct, and prevent vulnerabilities more efficiently during coding audits and peer review. |

## 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] | Guarantee that library functions do not overflow.  A buffer overflow can happen when copying data into a container that is not large enough. The container receiving the data must be guaranteed to be the size of the data being copied into it. |

| **Noncompliant Code** |
| --- |
| The std:copy() algorithm has no bounds check which can lead to a buffer overflow. The size of the src vector, in this instance could be larger than the receiving dest vector, and there is no check to expand dest if this occurs. |
| #include <algorithm>  #include <vector>    void f(const std::vector<int> &src) {    std::vector<int> dest;    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

| **Compliant Code** |
| --- |
| This code corrects the problems of the non-compliant code by creating the dest vector by using the size of the src vector. This creates a container that is the same size as the data being copied and will not result in a buffer overflow. |
| #include <algorithm>  #include <vector>  void f(const std::vector<int> &src) {    // Initialize dest with src.size() default-inserted elements    std::vector<int> dest(src.size());    std::copy(src.begin(), src.end(), dest.begin());    // ...  } |

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

|  |
| --- |
| **Principles(s):** **Validate Input Data** – checking the size of the container storing data can prevent library functions from overflowing |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **invalid\_pointer\_dereference** |  |
| CodeSonar | 6.0p0 | BADFUNC.BO.\*  LANG.MEM.BO | A collection of warning classes that report uses of library functions prone to internal buffer overflows.  Buffer Overrun |
| Parasoft C/C++ test | 2020.2 | **CERT\_CPP-CTR52-a** | Do not pass empty container iterators to std algorithms as destinations |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Value returning functions must return a value from all exit paths.  A returning function can lead to undefined behavior is all exit paths do not return a value. |

| **Noncompliant Code** |
| --- |
| This method does not return a proper value if the passed int is a positive number. |
| int absolute\_value(int a) {    if (a < 0) {      return -a;    }  } |

| **Compliant Code** |
| --- |
| This method corrects the noncompliant code by adding a return statement when the passed int is a positive number. |
| int absolute\_value(int a) {    if (a < 0) {      return -a;    }    return a;  } |

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

|  |
| --- |
| **Principles(s):** **Sanitize Data Sent to Other Systems** – returning a value from a function must be clear when returning the result to avoid potential injection attacks or hackers using the return value when there is not an explicit return call. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **return-implicit** | Fully checked |
| Clang | 3.9 | **-Wreturn-type** | Does not catch all instances of this rule, such as *function-try-blocks* |
| CodeSonar | 6.0p0 | **LANG.STRUCT.MRS** | Missing return statement |
| LDRA tool suite | 9.7.1 | **2 D, 36 S** | Fully implemented |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator.  A buffer overflow can happen when copying data to a container that is not large enough to contain that data. This can happen frequently when manipulating strings. |

| **Noncompliant Code** |
| --- |
| In this code, there is no check on the size of the user input and this could result in a buffer overflow. |
| #include <iostream>    void f() {    char buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| In this code, the bounded char array has been substituted for std::string which isn’t bound by a pre-determined size and can handle the user’s input. |
| #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):** **Validate Input Data** – checking the size of the container storing data can prevent strings from overflowing |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO LANG.MEM.TO** | No space for null terminator  Buffer overrun  Type overrun |
| LDRA tool suite | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| Parasoft C/C++ test | 2020.2 | **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' |
| Polyspace Bug Finder | R2020a | **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   Rule partially covered. |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C] | Whitelist the user input for SQL queries.  Using a hardcoded whitelist will restrict the user from using special characters that are needed for SQL injection attacks |

| **Noncompliant Code** |
| --- |
| This method of getting user input for the QUERY\_STRING will result in the user input not being checked for unaccepted characters. |
| int main(void) {  char \*user\_data;  user\_data = getenv("QUERY\_STRING");  printf("%s\n", user\_data);  exit(0)  } |

| **Compliant Code** |
| --- |
| This code creates a whitelist of accepted char values and checks them against the user’s input. This will sanitize the input and prevent SQL Injection attacks. |
| int main(void) {  static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz\  ABCDEFGHIJKLMNOPQRSTUVWXYZ\  1234567890";  char \*user\_data;  char \*cp;  user\_data = getenv("QUERY\_STRING");  printf("%s\n", user\_data);  for (cp = user\_data; \*(cp += strspn(cp, ok\_chars)); )  \*cp = '\_';  printf("%s\n", user\_data);  exit(0)  } |

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

|  |
| --- |
| **Principles(s):** **Sanitize Data Sent to Other Systems** – whitelisting user input for SQL queries prevents unwanted characters in the query being sent to the database.  **Default Deny** – the default is to deny all characters unless explicitly granted by the whitelist of accepted characters of the user input/SQL query. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported by stubbing/taint analysis |
| CodeSonar | 6.0p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Coverity | 6.5 | **TAINTED\_STRING** | Fully implemented |
| LDRA Tool suite | 9.7.1 | **108 D, 109 D** | Partially implemented |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not read uninitialized memory.  Declaring a variable assumes unexpected values if they are accessed before they are initialized. |

| **Noncompliant Code** |
| --- |
| This code declares an int variable and attempts to access it before it is initialized. This results in undefined behavior. |
| #include <iostream>    void f() {    int i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this code, the int variable is properly initialized before accessing it, leading to expected behavior. |
| #include <iostream>    void f() {    int i = 0;    std::cout << i;  } |

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

|  |
| --- |
| **Principles(s):** **Heed Compiler Warnings** – reading a variable that has been declared but not initialized will trigger a compiler warning.  **Validate Input Data** – validating the variable contains the proper value will prevent reading uninitialized memory. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **uninitialized-local-read**  **uninitialized-variable-use** | Fully checked |
| CodeSonar | 6.0p0 | **LANG.MEM.UVAR** | Uninitialized variable |
| Coverity | 2017.07 | **UNINIT** | Implemented |
| Cppcheck | 1.66 | **uninitvar uninitdata uninitstring uninitMemberVar uninitStructMember** | Detects uninitialized variables, uninitialized pointers, uninitialized struct members, and uninitialized array elements (However, if one element is initialized, then Cppcheck assumes the array is initialized.) There are FN compared to some other tools because Cppcheck tries to avoid FP in impossible paths. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Avoid side effects in arguments to unsafe macros.  Assertions should evaluate expressions that are devoid of potential side-effects. |

| **Noncompliant Code** |
| --- |
| A side effect is possible when using the index++ expression within the assert() macro. |
| #include <assert.h>  #include <stddef.h>    void process(size\_t index) {    assert(index++ > 0); /\* Side effect \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this code, the assert() macro only evaluates the index variable and ++index is performed outside the assert() macro expression. This will not cause side effects. |
| #include <assert.h>  #include <stddef.h>    void process(size\_t index) {    assert(index > 0); /\* No side effect \*/    ++index;    /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** **Keep It Simple** – do not attempt to make one line of code perform all the evaluations. Keep it to one line for one evaluation. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | **CertC-PRE31** | Fully Implemented |
| Coverityi | 2017.07 | **ASSERT\_SIDE\_EFFECTS** | Partially implemented  Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| ÉCLAIR | 1.2 | **CC2.EXP31 CC2.PRE31** | Fully implemented |
| LDRA tool suite | 9.7.1 | **9 S, 562 S, 572 S, 35 D, 1 Q** | Fully implemented |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Honor Exception Specifications  If a function throws an exception other than the one allowed by its specification, it can lead to termination of the program. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because the bar() method only claims to throw Exception1, yet Exception2 can also be thrown. |
| #include <exception>    class Exception1 : public std::exception {};  class Exception2 : public std::exception {};    void foo() {    throw Exception2{}; // Okay because foo() promises nothing about exceptions  }    void bar() throw (Exception1) {    foo();    // Bad because foo() can throw Exception2  } |

| **Compliant Code** |
| --- |
| This code corrects the noncompliant code by allowing the bar() method to handle Exception2 using a try/catch statement. |
| #include <exception>    class Exception1 : public std::exception {};  class Exception2 : public std::exception {};    void foo() {    throw Exception2{}; // Okay because foo() promises nothing about exceptions  }    void bar() throw (Exception1) {    try {      foo();    } catch (Exception2 e) {      // Handle error without rethrowing it    }  } |

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

|  |
| --- |
| **Principles(s):** **Use Effective Quality Assurance Techniques** – throwing and catching proper exceptions is vital to assuring the proper function and security of the software.  **Practice Defense in Depth** – although this standard is regarding two different exceptions, the exceptions being thrown could be layered, providing a depth of exception handing. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **unhandled-throw-noexcept** | Partially checked |
| LDRA tool suite | 9.7.1 | **56 D** | Partially implemented |
| Parasoft C/C++ Test | 2020.2 | **CERT\_CPP-ERR55-a** | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| RuleChecker | 20.10 | **unhandled-throw-noexcept** | Partially checked |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Values** | [STD-008-CPP] | Ensure that Integer conversions do not result in lost or misinterpreted data.  Integer conversions must be guaranteed not to result in lost or misinterpreted data. This is especially true for outside or untrusted sources. |

| **Noncompliant Code** |
| --- |
| This code results in an error due to data type casting between signed and unsigned integer types. |
| #include <limits.h>    void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    sc = (signed char)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This code catches errors associated with data type casting between signed and unsigned integer types. |
| #include <limits.h>    void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    if (u\_a <= SCHAR\_MAX) {      sc = (signed char)u\_a;  /\* Cast eliminates warning \*/    } else {      /\* Handle error \*/    }  } |

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

|  |
| --- |
| **Principles(s):** **Validate Input Data** – make sure that the integer variables contain the expected values before proceeding in the program. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR**  **LANG.CAST.PC.INT**  **LANG.CAST.COERCE LANG.CAST.VALUE**  **ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC**  **LANG.MEM.TBA** | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer  Coercion alters value Cast alters value  Truncation of allocation size Truncation of size  Tainted buffer access |
| Cppcheck | 1.66 | **memsetValueOutOfRange** | The second argument to memset() cannot be represented as unsigned char |
| LDRA tool suite | 9.7.1 | **93 S, 433 S, 434 S** | Partially implemented |
| Polyspace Bug Finder | R2020a | **CERT C: Rule INT31-C** | Checks for:   * Integer conversion overflow * Call to memset with unintended value * Sign change integer conversion overflow * Tainted sign change conversion * Unsigned integer conversion overflow   Rule partially covered. |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Declarations and Initializations** | [STD-009-CPP] | Explicitly specify void when a function accepts no arguments.  Defining a function without parameters differs from defining a function with void parameters in that a complier will not check for parameters of a function declared without. Defining the function with void parameters will result in the compiler rejecting parameters being passed. |

| **Noncompliant Code** |
| --- |
| In this code, the class foo() is defined without parameters. When foo() is called with a passed integer of 3, the compiler ignores it and runs the foo() function. |
| /\* In foo.h \*/  void foo();    /\* In foo.c \*/  void foo() {    int i = 3;    printf("i value: %d\n", i);  }    /\* In caller.c \*/  #include "foo.h"    foo(3); |

| **Compliant Code** |
| --- |
| In this code, the function foo() is defined with void parameters. When foo() is called with a passed integer of 3, the compiler will reject it and not allow the foo() function to execute. |
| /\* In foo.h \*/  void foo(void);    /\* In foo.c \*/  void foo(void) {    int i = 3;    printf("i value: %d\n", i);  }    /\* In caller.c \*/  #include "foo.h"    foo(3); |

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

|  |
| --- |
| **Principles(s):** **Heed Compiler Warnings** – creating a function with void parameters will cause a complier warning if parameters are passed to a function that excepts void parameters. The compiler may trigger a warning if the function is defined with empty parameters. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | **empty-parameter-list** | Fully Checked |
| CodeSonar | 6.0p0 | **LANG.FUNCS.PROT** | Incomplete function prototype |
| LDRA tool suite | 9.7.1 | **63 S** | Fully Implemented |
| Parasoft C/C++ test | 2020.2 | **CERT\_C-DCL20-a** | The number of arguments passed to a function shall match the number of parameters |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input/Output** | [STD-010-CPP] | Close files when they are no longer needed. |

| **Noncompliant Code** |
| --- |
| In this code, the file is never closed. The terminate() call ends the program but does not call destructors, which results in the file not being properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In this code, we see the call to file.close() before terminate() is called, which properly calls the deconstructors and closes the file properly. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

|  |
| --- |
| **Principles(s):** **Adhere to the Principle of Least Privilege** – files should only be open if the software needs to access them and should be closed immediately after they are no longer needed. This prevents unwanted access to an open file when the file is no longer actively needed. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **ALLOC.LEAK** | Leak |
| Coverity | 2017.07 | **RESOURCE\_LEAK (partial)** | Partially implemented |
| LDRA tool suite | 9.7.1 | **49 D** | Partially implemented |
| Parasoft C/C++ test | 2020.2 | **CERT\_C-FIO42-a** | Ensure resources are freed |

## 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.

Transitioning from DevOps to DevSecOps to implement better security practices is a great way for Green Pace to prepare for the security audit. The DevOps process that has been well-established is a solid starting point for pivoting to DevSecOps. Altering both pre-production and production phases of the DevOps process to include automation to accommodate the changes to the policy will be the starting point of the transition.

To implement these changes to Green Pace’s current process, unit tests, both positive and negative, will be implemented along side the building phase of pre-production, and used during the verify and test phase of pre-production and the monitor phase of production. These unit tests will test the individual functions within the software in an attempt mitigate these security issues. Third party testing software will be used during the various testing and monitoring phases on both the pre-production and production phases of the toolchain. Static code analysis tools such as Cppchecker and CodeSonar will be used alongside the unit tests to implement another layer of defense in our continuing pursuit of secure software development. These analysis tools put our software through much deeper analysis than the compiler of the IDE and seeks out a much wider array of vulnerabilities than our in-house unit tests.

## 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 | High | Likely | Medium | High | L1 |
| STD-002-CPP | Medium | Probable | Medium | Medium | L2 |
| STD-003-CPP | High | Likely | Medium | High | L1 |
| STD-004-C | High | Likely | High | High | L1 |
| STD-005-CPP | High | Probable | Medium | High | L1 |
| STD-006-CPP | Low | Unlikely | Low | Low | L3 |
| STD-007-CPP | Low | Likely | Low | Medium | L2 |
| STD-008-CPP | High | Probable | High | Medium | L2 |
| STD-009-CPP | Medium | Probable | Low | High | L1 |
| STD-010-CPP | Medium | Unlikely | Medium | Low | L3 |

## 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 | Data stored locally should be on encrypted storage.  Encryption in rest is the encrypting of files that are stored and not actively being used, such as data on a hard drive. Typically, this type of data is encrypted and protected by a firewall or a secondary level of security. Data at rest is encrypted as a layer of security in the instance someone gains unauthorized access to our system, they still will not be able to gain access to the data. |
| Encryption at flight | Transmitted data should remain encrypted, and only decrypted at the point of reading.  Encryption at flight is the encryption of data that is being sent from a database or data storage to be used. The data should remain or become encrypted while in transit and only decrypted when the program requires the data. This will protect the data until the moment the program requires the data to perform a function. |
| Encryption in use | Encrypted data should only be decrypted when necessary and encrypted again when no longer needed.  Encryption in use is the practice of encrypting data even when using it. In our case, we will be decrypting it for use and encrypting it as soon as it has been used. This will minimize the time and access anyone has to the decrypted data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The system requires two-factor authentication and a strong and unique password.  Authentication is permitting a user to access the system. Requiring a strong and unique password and a two-factor authentication can prevent brute force access into the system and only allow users with the proper credentials and two factor authentication code access. |
| Authorization | Users will be classified by different user types granting different authorizations in the systems.  Authorization is determined after authentication. The different user types will be based upon the data that is required for that user to fulfill their user role responsibilities. Only the data that is required will be authorized. |
| Accounting | A user’s activity will be logged when accessing certain data and navigating certain areas of the system.  Accounting, regarding the Triple-A framework, is referring to tracking a user’s activity, and keeping a log, or an accounting, of the behavior when logged into the system. In the event of a unauthorized access to data within the system, the logs will be able to indicate what user account was compromised and what areas of the system the user’s account attempted or did access during the activity. |

**\***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 | 3/21/2021 | Coding Standards | Greg MacPhelemy |  |
| 3.0 | 4/11/2021 | Completed Policy | Greg MacPhelemy |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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