**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.

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate the data that is being inputted to make sure it’s not harmful or corrupted as this could cause damage to the system. |
| 1. Heed Compiler Warnings | Don’t ignore compiler warnings as these warnings could pop up for other users and cause issues to the code/program. Not fixing compiler warnings could possibly be an open door for hackers which is not ideal. Fix during debug to make sure all warnings are fixed and/or known. |
| 1. Architect and Design for Security Policies | Build a secure system that has proper variables and follows security policies. Make sure that business requirements align with the security policies put in place. |
| 1. Keep It Simple | Don’t over complicate code. If there is a better, easier, and less intrusive way, go that path. Keep code to the point. Typically, complex code leaves more room for errors. |
| 1. Default Deny | Give user permissions to access the data in case it is needed. Deny everyone access unless they have the proper permissions and then they can access only allowed data. |
| 1. Adhere to the Principle of Least Privilege | Limit access to the information that can be obtained. Users only get the information that is needed, nothing more. |
| 1. Sanitize Data Sent to Other Systems | Remove any unused data in the system as hackers can use that to gather more information. Sanitize the system so that only data that is used is present. |
| 1. Practice Defense in Depth | Defense in depth is where there are multiple layers of security to a system to make sure that if one layer fails, there is another layer to take its place. Make sure there are multiple layers to the security system to make sure the system is protected. |
| 1. Use Effective Quality Assurance Techniques | Make sure that testing is done efficiently as this can help prevent any faults/errors. White-hat hackers, penetration testing, etc. ca be used to make sure the system is secure. |
| 1. Adopt a Secure Coding Standard | Depending on the choice of language and platform, have a secure coding standard put in place. |

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Do not cast to an out-of-range enumeration value |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results 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 compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| **enum** EnumType {    First,    Second,    Third  };    **void** f(**int** intVar) {  **if** (intVar < First || intVar > Third) {      // Handle error    }    EnumType enumVar = **static\_cast**<EnumType>(intVar);  } |

**Principles(s):**

| *Validate Input Data* – Make sure the values are specified as this can help prevent buffer overflow which in turn can lead a hacker to potentially running malicious code.  *Architect and Design for Security Policies* – System needs to be secure and casting an out-of-range enumeration value puts the system at risk. |
| --- |

**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++-INT50 |  |
| CodeSonar | 7.0p0 | LANG.CAST.COERCE  LANG.CAST.VALUE | Coercion Alters Value  Cast Alters Value |
| 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. |
| PVS-Studio | 7.18 | V1016 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Do not store an already-owned pointer value in an unrelated smart pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    **void** f() {  **int** \*i = **new** **int**;    std::shared\_ptr<**int**> p1(i);    std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction. When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable. |
| **void** f() {    std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();    std::shared\_ptr<**int**> p2(p1);  } |

**Principles(s):**

| *Architect and Design for Security Policies* – The result of storing an already-owned pointer value in an unrelated smart pointer is undefined behavior of the system. This puts the system at risk by having exploitable vulnerabilities.  *Heed Compiler Warnings* – With the undefined behavior, read the compiler warnings and address them as such to prevent any vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Dangling\_pointer\_use |  |
| Helix QAC | 2022.1 | C++4721, C++4722, C++4723 |  |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer. |
| Polyspace Bug Finder | R2022a | CERT C++:MEM56-CPP | Checks for use of already-owned pointers (rule fully covered). |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::string object is created from the results of a call to std::getenv(). However, because std::getenv() returns a null pointer on failure, this code can lead to undefined behavior when the environment variable does not exist (or some other error occurs). |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the results from the call to std::getenv() are checked for null before the std::string object is constructed. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

**Principles(s):**

| *Heed Compiler Warnings –* Results from creating a std string from a null pointer can cause the program to crash. Read the compiler warnings and address accordingly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Assert\_failure |  |
| CodeSonar | 7.0p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Helix QAC | 2022.1 | C++4770, C++4771, C++4772, C++4773, C++4774 |  |
| Parasoft C/C++ test | 2021.2 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Klocwork | 2022.1 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-C | Do not attempt to modify string literals |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior: |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| **char** str[] = "string literal";  str[0] = 'S'; |

**Principles(s):**

| *Use Effective Quality Assurance Techniques* – Attempt to modify the string and verify that it is done safely and doesn’t lead to undefined behavior.  *Heed Compiler Warnings* – If there are any warnings due to the program being terminated, look into them and adjust code accordingly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 | string-literal-modfication  write-to-string-literal | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR30 | Fully implemented |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| LDRA tool suite | 9.7.1 | 157 S | Partially implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Provide placement new with properly aligned pointers to sufficient storage capacity |

| **Noncompliant Code** |
| --- |
| This noncompliant code example ensures that the long is constructed into a buffer of sufficient size. However, it does not ensure that the alignment requirements are met for the pointer passed into placement new. To make this example clearer, an additional local variable c has also been declared. |
| #include <new>    **void** f() {  **char** c; // Used elsewhere in the function    unsigned **char** buffer[**sizeof**(**long**)];  **long** \*lp = ::**new** (buffer) **long**;      // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the alignas declaration specifier is used to ensure the buffer is appropriately aligned for a long. |
| #include <new>    **void** f() {  **char** c; // Used elsewhere in the function    alignas(**long**) unsigned **char** buffer[**sizeof**(**long**)];  **long** \*lp = ::**new** (buffer) **long**;      // ...  } |

**Principles(s):**

| *Use Effective Quality Assurance Techniques –* Making sure that pointers are tested to prevent buffer overflow.  *Validate Data Input –* Validate the data has enough storage to prevent buffer overflow. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM54 |  |
| CodeSonar | 7.0p0 | LANG.MEM.BO | Buffer Overrun |
| LDRA tool suite | 9.7.1 | 597 S | Enhanced Enforcement |
| Polyspace Bug Finder | R2022a | CERT C++:MEM54-CPP | Checks for placement new used with insufficient storage or misaligned pointers (rule fully covered) |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly: |
| #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** |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used, as in this compliant solution: |
| **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

**Principles(s):**

| *Use Effective Quality Assurance Techniques –* Static Assertion can help eliminate software defects. These defects make the program vulnerable at compile time.  *Keep it Simple –* Don’t need to overcomplicate it, run the assertion at compile time to avoid silent malfunctions or errors. This can in turn help catch any issues and avoid compile problems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus | 7.2.0 | CertC-DCL03 |  |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Honor exception specifications |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception 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** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

**Principles(s):**

| *Architect and Design for Security Policies –* Let the function allow all exceptions to catch anything. Design for security purposes.  *Heed Compiler Warnings –* Watch for any exceptions and address accordingly. |
| --- |

**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 |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR55 |  |
| Helix QAC | 2022.1 | C++4035, C++4036, C++4632 |  |
| LDRA tool suite | 9.7.1 | 56 D | Partially implemented |
| Parasoft C/C++Test | 2021.2 | 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) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming (OOP) | STD-008-CPP | Gracefully handle self-copy assignment |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the copy assignment operator does not protect against self-copy assignment. If self-copy assignment occurs, this->s1 is deleted, which results in rhs.s1 also being deleted. The invalidated memory for rhs.s1 is then passed into the copy constructor for S, which can result in dereferencing an invalid pointer. |
| #include <new>    **struct** S { S(**const** S &) noexcept; /\* ... \*/ };    **class** T {  **int** n;    S \*s1;    **public**:    T(**const** T &rhs) : n(rhs.n), s1(rhs.s1 ? **new** S(\*rhs.s1) : nullptr) {}    ~T() { **delete** s1; }      // ...      T& operator=(**const** T &rhs) {      n = rhs.n;  **delete** s1;      s1 = **new** S(\*rhs.s1);  **return** \***this**;    }  }; |

| **Compliant Code** |
| --- |
| This compliant solution guards against self-copy assignment by testing whether the given parameter is the same as this. If self-copy assignment occurs, then operator= does nothing; otherwise, the copy proceeds as in the original example. |
| #include <new>    **struct** S { S(**const** S &) noexcept; /\* ... \*/ };    **class** T {  **int** n;    S \*s1;    **public**:    T(**const** T &rhs) : n(rhs.n), s1(rhs.s1 ? **new** S(\*rhs.s1) : nullptr) {}    ~T() { **delete** s1; }      // ...      T& operator=(**const** T &rhs) {  **if** (**this** != &rhs) {        n = rhs.n;  **delete** s1;  **try** {          s1 = **new** S(\*rhs.s1);        } **catch** (std::bad\_alloc &) {          s1 = nullptr; // For basic exception guarantees  **throw**;        }      }  **return** \***this**;    }  }; |

**Principles(s):**

| *Architect and Design for Security Policies –* Guarding against self-copy assignment by testing designs the program to be more secure. Undefined behavior could occur otherwise. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use |  |
| Clang | 9.0 (r361550) | cert-oop54-cpp | Checked by clang-tidy. |
| CodeSonar | 7.0p0 | IO.DC  ALLOC.DF  ALLOC.LEAK  LANG.MEM.NPD  LANG.STRUCT.RC  IO.UAC  ALLOC.UAF | Double Close  Double Free  Leak  Null Pointer Dereference  Redundant Condition  Use After Close  Use After Free |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-OOP54-a | Check for assignment to self in operator= |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Signals | STD-009-C | Avoid using signals to implement normal functionality |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses signals as a means to pass state changes around in a multithreaded environment: |
| /\* THREAD 1 \*/  **int** do\_work(**void**) {    /\* ... \*/    kill(THR2\_PID, SIGUSR1);  }    /\* THREAD 2 \*/  **volatile** **sig\_atomic\_t** flag;    **void** sigusr1\_handler(**int** signum) {    flag = 1;  }    **int** wait\_and\_work(**void**) {    flag = 0;  **while** (!flag) {}    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| A better solution, in this case, is to use condition variables. This code example uses a condition variable from the POSIX pthread library [IEEE Std 1003.1:2013]: |
| #include <pthread.h>    pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;  pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER;    /\* THREAD 1 \*/  **int** do\_work(**void**) {  **int** result;    /\* ... \*/  **if** ((result = pthread\_mutex\_lock(&mut)) != 0) {      /\* Handle error condition \*/    }  **if** ((result = pthread\_cond\_signal(&cond,&mut)) != 0) {      /\* Handle error condition \*/    }  **if** ((result = pthread\_mutex\_unlock(&mut)) != 0) {      /\* Handle error condition \*/    }  }    /\* THREAD 2 \*/  **int** wait\_and\_work(**void**) {  **if** ((result = pthread\_mutex\_lock(&mut)) != 0) {      /\* Handle error condition \*/    }  **while** (/\* Condition does not hold \*/) {  **if** ((result = pthread\_cond\_wait(&cond, &mut)) != 0) {        /\* Handle error condition \*/      }      /\* ... \*/    }  **if** ((result = pthread\_mutex\_unlock(&mut)) != 0) {      /\* Handle error condition \*/    }    /\* ... \*/  } |

**Principles(s):**

| *Architect and Design for Security Policies –* Having the program/system have error-handling mechanisms to prevent issues occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.0p0 | BADFUNC.SIGNAL | Use of signal |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| Parasoft C/C++test | 2021.2 | CERT\_C-SIG02-a | The signal handling facilities of <signal.h> shall not be used |
| PC-lint Plus | 1.4 | 586 | Assistance provided: reports use of the signal function |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-C | Choose an appropriate termination strategy |

| **Noncompliant Code** |
| --- |
| The abort() function should not be called if it is important to perform application-specific cleanup before exiting. In this noncompliant code example, abort() is called after data is sent to an open file descriptor. The data may or may not be written to the file. |
| #include <stdlib.h>  #include <stdio.h>    **int** write\_data(**void**) {  **const** **char** \*filename = "hello.txt";  **FILE** \*f = **fopen**(filename, "w");  **if** (f == NULL) {      /\* Handle error \*/    }  **fprintf**(f, "Hello, World\n");    /\* ... \*/  **abort**(); /\* Oops! Data might not be written! \*/    /\* ... \*/  **return** 0;  }    **int** main(**void**) {    write\_data();  **return** EXIT\_SUCCESS;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to abort() is replaced with exit(), which guarantees that buffered I/O data is flushed to the file descriptor and the file descriptor is properly closed: |
| #include <stdlib.h>  #include <stdio.h>    **int** write\_data(**void**) {  **const** **char** \*filename = "hello.txt";  **FILE** \*f = **fopen**(filename, "w");  **if** (f == NULL) {      /\* Handle error \*/    }  **fprintf**(f, "Hello, World\n");    /\* ... \*/  **exit**(EXIT\_FAILURE); /\* Writes data and closes f \*/    /\* ... \*/  **return** 0;  }    **int** main(**void**) {    write\_data();  **return** EXIT\_SUCCESS;  } |

**Principles(s):**

| *Architect and Design for Security Policies –* Design the program to exit properly to avoid having files in an inconsistent state and/or leaving sensitive files on the file system.  *Use Effective Quality Assurances Techniques –* Test in a dummy system to verify files are fine before releasing the program. Make sure files are secure in testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.2 | CERT\_C-ERR04-a  CERT\_C-ERR04-b  CERT\_C-ERR04-c | The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  The 'exit()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |
| PC-lint Plus | 1.5 | 586 | Fully Supported |

### 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.

Code that is created in pre-production will be designed by following the coding standards and making sure it follows the security principles. This code will then be testing using built in IDE tools and also outside tools to catch any possible errors that makes the code/system vulnerable. These errors will be logged and addressed accordingly to help the current and future projects. Once verified and tested with analysis tools, it will then move on to production where penetration testing is issued and is monitored. If the code/project does not pass the verify and testing stage, it will not move on to production and will be analyzed until it passes.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-002-CPP | High | Likely | Medium | P18 | L1 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | Low | Likely | Low | P9 | L2 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Likely | Low | P9 | L2 |
| STD-008-CPP | Low | Probable | High | P2 | L3 |
| STD-009-C | High | Probable | Medium | P12 | L1 |
| STD-010-C | Medium | Probable | High | P4 | 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 | This is where data is being encrypted while the data is being stored. Where unauthorized users cannot decrypt it and thus preventing the data from falling into the wrong hands. It’s converting the data into another form of data through an algorithm that can only be understood by a person who has the key in order to decode it. This policy applies because it helps protect the data in the system. Helping defend from hackers getting their hands on sensitive information of customers and/or the company. |
| Encryption at flight | This is where data is being encrypted while it is transmitting over the network. This helps prevent getting data stolen over the network. Each side, sender and receiver (if authorized) will have an encryption key that is unique so that once they have the data, it can be de-encrypted. This policy applies because it helps from preventing transmitting data stolen. |
| Encryption in use | Data that is either at rest, in use, or in transit will never be left unsecured. It does not matter where it is being stored but the sensitive data will always be protected. This policy applies in order to keep sensitive data, regardless on what it is doing (at rest, transmitting, or being used) is always secured to assure the customers, partners, etc. that their information is safe. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is necessary to provide the business a way to allow certain users to access the system. This can be done using a user ID, password, and possibly a biometric signature. This policy applies because it prevents unauthorized users out of the system and accessing data. |
| Authorization | Authorization is needed to grant access/rights for the authenticated users. This allows for limiting users from accessing information that they do not need, and also granting users to accessing information they do need. Data that is being accessed by too many users when only a few should be accessing it can make the data less secured. |
| Accounting | Accounting is basically a checks and balances for a business. Making sure the system is being monitored and being aware of the data that is being used. It also helps with authentication and authorization to see who has what capabilities and if they should keep said capabilities. It can help in preventing data getting into the wrong hands. |

**\***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 |  |
| 1.1 | 05/22/2022 | Coding Standards Addressed | Chloe Ninefeldt | Trevor Hodde |
| 1.2 | 06/12/2022 | Finalized Assessment and Template Completed | Chloe Ninefeldt | Trevor Hodde |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

Resources

Das, P. (2022, May 11). *In-use encryption – what it is and how companies benefit*. Sotero. Retrieved June 8, 2022, from <https://www.soterosoft.com/blog/data-in-use-encryption-data-in-motion-encryption/#:~:text=In%2DUse%20encryption%20takes%20a,%2C%20cloud%2C%20or%20hybrid>).

Model, I. (2017, May 17). *What is encryption at rest, and why is it important for your business?* Brightline Technologies. Retrieved June 8, 2022, from <https://brightlineit.com/encryption-at-rest-important-business/>

Shacklett, M. E. (2021, September 27). *What is authentication?* SearchSecurity. Retrieved June 12, 2022, from <https://www.techtarget.com/searchsecurity/definition/authentication#:~:text=Authentication%20enables%20organizations%20to%20keep,network%2Dbased%20applications%20or%20services>.