**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 all data coming from external or untrusted sources. This will remove the majority of vulnerabilities. |
| 1. Heed Compiler Warnings | While compiling code pay attention to the highest warning level and be sure to modify the code to remove these warning. |
| 1. Architect and Design for Security Policies | Create and use a software design document to ensure the implementation of security policies. This will also help enforce the use and need of said policies. |
| 1. Keep It Simple | Keep the design of your system simple, as the more complex the system becomes the more likely that bugs and vulnerabilities will be introduced. It also becomes much harder to detect and fix bugs and vulnerabilities that do appear in the system. |
| 1. Default Deny | Create the access to the system based on roles and permissions rather than exclusion lists. |
| 1. Adhere to the Principle of Least Privilege | Access should be given in a way that the requester only gets the amount of access necessary to complete the task. If elevated permission is necessary, it should be removed immediately after the completion of the task. |
| 1. Sanitize Data Sent to Other Systems | Remove special characters or known commands/actions when transferring data between systems. This can prevent incidents such as injection attacks. |
| 1. Practice Defense in Depth | Be sure to have multiple layers of security in case one layer turns out inadequate at protecting a specific attack. Evaluate environment to decide on how many layers are necessary. |
| 1. Use Effective Quality Assurance Techniques | By using external audits, pen testing, etc, it can lead to more secure code and systems and overall prevent intrusions and vulnerabilities. |
| 1. Adopt a Secure Coding Standard | Be sure to incorporate and use secure coding standards for the language being developed with. |

### 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** | INT00-C | Understand the data model used by your implementation(s)  If your code depends on assumptions that is not guaranteed by the standards, then you should provide static limits to ensure your assumptions are valid. |

| **Noncompliant Code** |
| --- |
| This code reads a long into an int and cause a buffer overflow. |
| int f(void) {    FILE \*fp;    int x;  /\* ... \*/    if (fscanf(fp, "%ld", &x) < 1) {      return -1; /\* Indicate failure \*/    }    /\* ... \*/    return 0;  } |

| **Compliant Code** |
| --- |
| This code uses the correct format for the type being used. |
| int f(void) {    FILE \*fp;    int x;  /\* Initialize fp \*/    if (fscanf(fp, "%d", &x) < 1) {      return -1; /\* Indicate failure \*/    }    /\* ... \*/    return 0;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT00 |  |
| PC-lint Plus | 1.4 | 559, 705, 706, 2403 | Assistance provided: Reports data type inconsistencies in format strings |
| Polyspace Bug Finder | R2022b | CERT C: Rec. INT00-C | Checks for:  Use of basic numerical types instead of typedef-s  Integer overflow or integer constant overflow  Format string specifiers and arguments mismatch  Rec. partially covered |
| PVS-Studio | 7.2.1 | V629, V5004 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT31-C | Ensure that integer conversions do not result in lost or misinterpreted data  Data must not be lost or misinterpreted, especially from untrusted sources. This rule applies to all the following scenarios:   * Integer operands of any pointer arithmetic, including array indexing * The assignment expression for the declaration of a variable length array * The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object * Function arguments of type size\_t or rsize\_t (for example, an argument to a memory allocation function)   This also applies to all libraries and functions that convert data types. |

| **Noncompliant Code** |
| --- |
| Data and sign loss can occur when converting from an unsigned integer to a signed integer. |
| #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 validates ranges when converting unsigned long int to a signed char. |
| #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):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| CodeSonar | 7.1p0 | 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 |
| Compass/ROSE |  |  | Can detect violations of this rule. However, false warnings may be raised if limits.h is included |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | FIO47-C | Use valid format strings  This standard is providing the correct way to use conversion specifiers within a String. |

| **Noncompliant Code** |
| --- |
| By using incorrect conversion specifier, the outcome may result in undefined behavior. |
| #include <stdio.h>    void func(void) {    const char \*error\_msg = "Resource not available to user.";    int error\_type = 3;    /\* ... \*/    printf("Error (type %s): %d\n", error\_type, error\_msg);    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The correct conversion specifiers are used with printf() |
| #include <stdio.h>    void func(void) {    const char \*error\_msg = "Resource not available to user.";    int error\_type = 3;    /\* ... \*/    printf("Error (type %d): %s\n", error\_type, error\_msg);      /\* ... \*/  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO47 | Fully implemented |
| CodeSonar | 7.1p0 | IO.INJ.FMT  MISC.FMT  MISC.FMTTYPE | Format string injection  Format string  Format string type error |
| Coverity | 2017.07 | PW | Reports when the number of arguments differs from the number of required arguments according to the format string |
| GCC | 4.3.5 |  | Can detect violations of this recommendation when the -Wformat flag is used |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STR02-C | Sanitize data passed to complex subsystems  Remove all data that may contain special characters or commands before passing it to a subsystem. |

| **Noncompliant Code** |
| --- |
| To sanitize data, it is required to understand what data is being passed and what the subsystem can do. This is an example to data being passed without being sanitized. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| This example shows code that understands what data is valid and what is not allowed to be passed to the subsystem. It implements a filter based on those parameters. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 7.1p0 | 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 |
| Klocwork | 2022.3 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM35-C | Allocate sufficient memory for an object  Integer expressions used as size arguments, specifically: malloc(), calloc(), realloc(), or aligned\_alloc(), must have enough memory to represent the size of the object being stored. If the size argument is incorrect or manipulable an attacker could cause a buffer overflow. |

| **Noncompliant Code** |
| --- |
| The incorrect space is allocated for struct tm |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,                     int min, int sec) {    struct tm \*tmb;    tmb = (struct tm \*)malloc(sizeof(tmb));    if (tmb == NULL) {      return NULL;    }    \*tmb = (struct tm) {      .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,      .tm\_mday = day, .tm\_mon = mon, .tm\_year = year    };    return tmb;  } |

| **Compliant Code** |
| --- |
| The correct amount of memory was allocated for struct tm. This is done by passing the pointer type of sizeof. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,                     int min, int sec) {    struct tm \*tmb;    tmb = (struct tm \*)malloc(sizeof(\*tmb));    if (tmb == NULL) {      return NULL;    }    \*tmb = (struct tm) {      .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,      .tm\_mday = day, .tm\_mon = mon, .tm\_year = year    };    return tmb;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | malloc-size-insufficient | Partially checked  Besides direct rule violations, all undefined behaviour resulting from invalid memory accesses is reported by Astrée. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM35 |  |
| CodeSonar | 7.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  ALLOC.SIZE.TRUNC  IO.TAINT.SIZE  MISC.MEM.SIZE.BAD  LANG.MEM.BO  LANG.MEM.BU  LANG.STRUCT.PARITH  LANG.STRUCT.PBB  LANG.STRUCT.PPE  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Addition overflow of allocation size  Addition overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Truncation of allocation size  Tainted allocation size  Unreasonable size argument  Buffer Overrun  Buffer Underrun  Pointer Arithmetic  Pointer Before Beginning of Object  Pointer Past End of Object  Tainted Buffer Access  Type Overrun  Type Underrun |
| Compass/ROSE |  |  | Could check violations of this rule by examining the size expression to malloc() or memcpy() functions. Specifically, the size argument should be bounded by 0, SIZE\_MAX, and, unless it is a variable of type size\_t or rsize\_t, it should be bounds-checked before the malloc() call. If the argument is of the expression a\*b, then an appropriate check is   |  | | --- | | if (a < SIZE\_MAX / b && a > 0) ... | |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | DCL03-C | Use a static assertion to test the value of a constant expression.  Useful for identifying incorrect assumptions and not for runtime errors. |

| **Noncompliant Code** |
| --- |
| Assert() is being used on a memory-mapped structure essential for the code to work 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** |
| --- |
| A conditional statement may be used instead of assert() |
| 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 |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | **CertC-DCL03** |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 7.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | Honor exception specifications  Expect that an exception may be thrown and correctly handle the exception. If it is not correctly handled the program may terminate. |

| **Noncompliant Code** |
| --- |
| A function is declared as nonthrowing, however the expression used may throw an exception. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) noexcept(true) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| This code allows exceptions to be thrown. |
| #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):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | unhandled-throw-noexcept | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR55 |  |
| CodeSonar | 7.1p0 | LANG.STRUCT.EXCP.THROW | Use of throw |
| Helix QAC | 2022.3 | C++4035, C++4036, C++4632 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Lifetime | DCL30-C | Declare objects with appropriate storage durations  Do not attempt to access an object outside of it’s lifetime. |

| **Noncompliant Code** |
| --- |
| The assignment of c\_str is correct, however is goes out of scope while p holds it’s address. |
| #include <stdio.h>    const char \*p;  void dont\_do\_this(void) {    const char c\_str[] = "This will change";    p = c\_str; /\* Dangerous \*/  }    void innocuous(void) {    printf("%s\n", p);  }    int main(void) {    dont\_do\_this();    innocuous();    return 0; |

| **Compliant Code** |
| --- |
| P is set to NULL before c\_str is destroyed. Preventing p from taking on a random value. |
| const char \*p;  void is\_this\_OK(void) {    const char c\_str[] = "Everything OK?";    p = c\_str;    /\* ... \*/    p = NULL;  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | pointered-deallocation  return-reference-local | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL30 | Fully implemented |
| CodeSonar | 7.1p0 | LANG.STRUCT.RPL | Returns pointer to local |
| Compass/ROSE |  |  | Can detect violations of this rule. It automatically detects returning pointers to local variables. Detecting more general cases, such as examples where static pointers are set to local variables which then go out of scope, would be difficult |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer Signs | INT32-C | Ensure that operations on signed integers do not result in overflow  Signed integer overflow is undefined behavior 36. It is important that overflow does not occur:   * Integer operands of any pointer arithmetic, including array indexing * The assignment expression for the declaration of a variable length array * The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object * Function arguments of type size\_t or rsize\_t (for example, an argument to a memory allocation function) |

| **Noncompliant Code** |
| --- |
| This code can result in a signed integer overflow during the addition of signed operands. |
| void func(signed int si\_a, signed int si\_b) {    signed int sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This shows that overflow cannot happen regardless of representation. |
| #include <limits.h>    void f(signed int si\_a, signed int si\_b) {    signed int sum;    if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||        ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {      /\* Handle error \*/    } else {      sum = si\_a + si\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | integer-overflow | Fully checked |
| CodeSonar | 7.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Helix QAC | 2022.3 | C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | EXP50-CPP | Do not depend on the order of evaluation for side effects  A way to modify the way of execution, modifying an object, etc are called side effects. Do not allow scalar objects to appear in side effects or value computations. |

| **Noncompliant Code** |
| --- |
| In this code i is evaluated more than once, causing an undefined expression. |
| void f(int i, const int \*b) {    int a = i + b[++i];    // ...  } |

| **Compliant Code** |
| --- |
| This code is independent of the order of operands and can not be misinterpreted. |
| void f(int i, const int \*b) {    ++i;    int a = i + b[i];    // ...  } |

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

| **Principles(s):** [Name the principle and explain how it maps to this standard.] |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-EXP50 |  |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required |
| CodeSonar | 7.1p0 | LANG.STRUCT.SE.DEC  LANG.STRUCT.SE.INC | Side Effects in Expression with Decrement  Side Effects in Expression with Increment |
| Compass/ROSE |  |  | Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator |

### 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.

To change a DevOps environment into a DevSecOps environment you must adapt a security first environment. Starting with the “Pre-production”phase of the environment; therat modeling, security tool training, and metrics/threat modeling are added to the “Access and Plan” step. Next, within the “Design” and “Build” steps Security Plug-Ins are added to the IDE. Moving to Verify and Test, adding application testing and vulnerability scans will assist with the DevSecOps transition.

Within the “Production” phase automated tests and monitoring are enabled. Starting with “Transition and health check” step, adding Signature Verity, Integrity Checks, and Defense In-Depth Measures will ensure the steps meets DevSecOps requirements. With the step “Monitor and detect” using Network Monitoring and Penetration Testing would secure it. “Respond” should implement the use of RASPH/WAF Shielding and Obfuscation. Finally, “Maintain and stabilize” should use Dev Consumables and Vulnerability Analysis.

### 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 | Unlikely | Medium | High | 2 |
| INT00-C | High | Unlikely | High | P3 | L2 |
| INT31-C | High | Probable | High | P6 | L2 |
| FIO47-C | High | Unlikely | Medium | P6 | L2 |
| STR02-C | High | Likely | Medium | P18 | L1 |
| MEM35-C | High | Probable | High | P6 | L2 |
| DCL03-C | Low | Unlikely | High | P1 | L3 |
| ERR55-CPP | Low | Likely | Low | P9 | L2 |
| DCL30-C | High | Probable | High | P6 | L2 |
| INT32-C | High | Likely | High | P9 | L2 |
| EXP50-CPP | Medium | Probable | Medium | P8 | L2 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This type of encryption provides protection for stored data, irrelevant of the device it’s on. This helps prevent unwanted data access, whether it be physically attempting to access the data or remotely. A common encryption algorithm used is AES 256 |
| Encryption at flight | This encryption protects data as it moves through a network, such as a WAN or larger LANs. Typically, a key that is unique is shared between the two devices and is decrypted once received. This is useful for several reasons, such as a bank send data to a brand, or a hospital transferring data with a patient. |
| Encryption in use | This type of encryption is typically data in use by one or more applications. This includes data being read, written, or executed. Generally, this form of encryption is difficult to implement at a software level and should be implemented via hardware. Manufacturers such as Intel and AMD include such features directly into their processors. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Checks who are attempting to access the resource by requiring a username and password, SSO systems, biometrics, digital certificates, and public key infrastructure. Each user will have a unique ID, with specific roles and access assigned.  This prevents unwanted access or malicious actors from access systems and using resources that shouldn’t. Such as IoT devices or servers. |
| Authorization | Checks to see what resources the user is authorized to use and permits that access. This can be decided by a role, rule, or policy and can be applied per user, group, or physical location.  This ensures that an authorized user can only see what they should. Such as a receptionist should not be able to access ESXI. |
| Accounting | Records logs and watches the network to see what users consume or do on their account. Stuff such as network logs, application usage, session duration, and even the amount of data used can fall into this category.  This can ensure that if a user tries to copy a file without permission it is recorded into the logs and can prevent a data leak. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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