**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 | The understanding that input is harmful until proven otherwise. Do not automatically trust the source of the input without further authentication. Make input validation a consistent routine for development. Restrict user fields and validate client values. Ensure only proper data formats enter the software. Check integrity and accuracy of the data prior to its usage. |
| 1. Heed Compiler Warnings | When compiling code, use highest available standards for the compiler and eliminate the warnings completely. Consider all compiler warnings as potential catastrophic issues in the code with critical errors. Static and dynamic analysis tools are crucial for detection and isolating security issues. As these warnings do not interrupt the compilation process, treat them as critical failures even if they turn out to be minimal. |
| 1. Architect and Design for Security Policies | Develop and pattern architecture for optimum implementation and enforcement of security privileges. Advance your architecture to guard against malicious intent and vulnerabilities by writing secure code as opposed to adding security later as an additional layer. Have the architecture reflect security-relevant elements within domains and interconnections. |
| 1. Keep It Simple | Use simplicity as a strength in that complex design can lead to errors which are difficult to realize or track. With simpler code and design, further assurances can be made in determination of quality security measures. Security guarantees become drastically more difficult to maintain as programs become more complex. |
| 1. Default Deny | Base access on a decision identified with a condition and permission. Block traffic that is not permitted using specific condition as access to sensitive resources are excluded unless the individual is authorized. |
| 1. Adhere to the Principle of Least Privilege | In efforts in reduction of attack opportunities, elevated permission should be held for a minimum time if at all. A user should be isolated to access certain resources and task requirements necessary for the assignment. Determine and utilize measures and levels of security to give the user minimal access or permission to perform their job. |
| 1. Sanitize Data Sent to Other Systems | Potential attacks are mitigated and/or reduced by eliminated unwanted characters from input. Physical destruction, cryptographic erasure, and data erasure are methodical in data sanitization and reduce security risk before data is passed across a trust boundary. |
| 1. Practice Defense in Depth | Multiple defense strategies are necessary for proper security. The practice of implementing continuous layers of adequate prevention assurances which might limit consequences of an attack. Secure programming techniques and layered environments greatly reduce the likelihood that malicious attacks affect multiple layers if one barrier is compromised. |
| 1. Use Effective Quality Assurance Techniques | Employ productive approaches to eliminating vulnerabilities in your code such as source code audits, penetration testing, fuzz testing, and independent security review. External reviews bring more objective perspectives in identification. Actively advance practices in ensuring the end product conforms and meets agreed upon expectations. |
| 1. Adopt a Secure Coding Standard | Become familiar and utilize development practices specific to the individual language you are using which promotes the utmost security. These particular standards should become second nature and correlate with the platform of utilization. |

### 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** | **Use correct integer precisions** |
| --- | --- | --- |
| **Data Type** | STD- INT35-C | Integer types in C have both a size and a precision. The size indicates the number of bytes used by an object and can be retrieved for any object or type using the size of operator. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.  Padding bits contribute to the integer's size, but not to its precision. Consequently, inferring the precision of an integer type from its size may result in too large a value, which can then lead to incorrect assumptions about the numeric range of these types. Programmers should use correct integer precisions in their code, and in particular, should not use the sizeof operator to compute the precision of an integer type on architectures that use padding bits or in strictly conforming (that is, portable) programs. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. |
| #include <limits.h>    unsigned int pow2(unsigned int exp) {    if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {      /\* Handle error \*/    }    return 1 << exp;  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a popcount() function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num) {    size\_t precision = 0;    while (num != 0) {      if (num % 2 == 1) {        precision++;      }      num >>= 1;    }    return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):** *10*. *Adopt a Secure Coding* Standard is the principle correlated with this standard. Mistaking an integer's size for its precision can permit invalid precision arguments to operations such as bitwise shifts, resulting in undefined behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported: Astrée reports overflows due to insufficient precision |
| CodeSonar | 7.2p0 | LANG.ARITH. BIGSHIFT | Shift Amount Exceeds Bit Width |
| Helix QAC | 2022.4 | C0582  C++3115 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_C-INT35-a | Use correct integer precisions when checking the right-hand operand of the shift operator |
| Polyspace Bug Finder | R2022b | CERT C: Rule INT35-C | Checks for situations when integer precisions are exceeded (rule fully covered) |
| PRQA QA-C | 9.7 | 0582 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not use object representations to compare floating point values** |
| --- | --- | --- |
| **Data Value** | STD-FLP37-C | The object representation for floating-point values is implementation defined. However, an implementation that defines the \_\_STDC\_IEC\_559\_\_ macro shall conform to the IEC 60559 floating-point standard and uses what is frequently referred to as IEEE 754 floating-point arithmetic [ISO/IEC 9899:2011]. The floating-point object representation used by IEC 60559 is one of the most common floating-point object representations in use today.  All floating-point object representations use specific bit patterns to encode the value of the floating-point number being represented. However, equivalence of floating-point values is not encoded solely by the bit pattern used to represent the value. For instance, if the floating-point format supports negative zero values (as IEC 60559 does), the values -0.0 and 0.0 are equivalent and will compare as equal, but the bit patterns used in the object representation are not identical. Similarly, if two floating-point values are both (the same) NaN, they will not compare as equal, despite the bit patterns being identical, because they are not equivalent.  Do not compare floating-point object representations directly, such as by calling memcmp()or its moral equivalents. Instead, the equality operators (== and !=) should be used to determine if two floating-point values are equivalent. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, memcmp() is used to compare two structures for equality. Since the structure contains a floating-point object, this code may not behave as the programmer intended. |
| #include <stdbool.h>  #include <string.h>    struct S {    int i;    float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {    if (!s1 && !s2)      return true;    else if (!s1 || !s2)      return false;    return 0 == memcmp(s1, s2, sizeof(struct S));  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the structure members are compared individually: |
| #include <stdbool.h>  #include <string.h>    struct S {    int i;    float f;  };    bool are\_equal(const struct S \*s1, const struct S \*s2) {    if (!s1 && !s2)      return true;    else if (!s1 || !s2)      return false;    return s1->i == s2->i &&           s1->f == s2->f;  } |

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

| **Principles(s):** 7. *Sanitize Data Sent to Other Systems* correlates with this standard as using the object representation of a floating-point value for comparisons can lead to incorrect equality results, which can lead to unexpected behavior. 10. Adopt a Secure Coding Standard could also be closely correlated with this standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | memcmp-with-float | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FLP37 | Fully implemented |
| Helix QAC | 2022.4 | C5026  C++3118 |  |
| Klocwork | 2022.4 | MISRA.STDLIB.MEMCMP.PTR\_ARG\_TYPES CERT.MEMCMP.FLOAT\_MEMBER |  |
| LDRA tool suite | 9.7.1 | 618 S | Enhanced Enforcement |
| Parasoft C/C++ test | 2022.2 | CERT\_C-FLP37-c | Do not use object representations to compare floating point values |
| PC-lint Plus | 1.4 | 2498,2499 | Fully supported |
| Polyspace Bug Finder | R2022b | CERT C: Rule FLP37-C | Checks for memory comparison of floating-point values (rule fully covered) |
| PRQA QA-C | 9.7 | 5026 |  |
| PVS-Studio | 7.23 | V1014 |  |
| RuleChecker | 22.04 | memcmp-with-float | Partially checked |
| TrustInSoft Analyzer | 1.38 |  | Exhaustively verified |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use valid format strings** |
| --- | --- | --- |
| **String Correctness** | STD-FIO47-C | The formatted output functions (fprintf() and related functions) convert, format, and print their arguments under control of a format string. The C Standard, 7.21.6.1, paragraph 3 [ISO/IEC 9899:2011], specifies  The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream. |

| **Noncompliant Code** |
| --- |
| Mismatches between arguments and conversion specifications may result in undefined behavior. Compilers may diagnose type mismatches in formatted output function invocations. In this noncompliant code example, the error\_type argument to printf() is incorrectly matched with the s specifier rather than with the d specifier. Likewise, the error\_msg argument is incorrectly matched with the d specifier instead of the s specifier. These usages result in undefined behavior. One possible result of this invocation is that printf() will interpret the error\_type argument as a pointer and try to read a string from the address that error\_type contains, possibly resulting in an access violation. |
| #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** |
| --- |
| This compliant solution ensures that the arguments to the printf() function match their respective conversion specifications: |
| #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):** 1. Validate Input Data and *8. Practice Defense in Depth* could be correlated with this standard as incorrectly specified format strings can result in memory corruption or abnormal program termination. |
| --- |

**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 | 7.2p0 | 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 |
| Helix QAC | 2022.4 | C0161, C0162, C0163, C0164, C0165, C0166, C0167, C0168, C0169, C0170, C0171, C0172, C0173, C0174, C0175, C0176, C0177, C0178, C0179, C0180, C0184, C0185, C0190, C0191, C0192, C0193, C0194, C0195, C0196, C0197, C0198, C0199, C0200, C0201, C0202, C0204, C0206, C0209  C++3150, C++3151, C++3152, C++3153, C++3154, C++3155, C++3156, C++3157, C++3158, C++3159 |  |
| Klocwork | 2022.4 | SV.FMT\_STR.PRINT\_FORMAT\_MISMATCH.BAD  SV.FMT\_STR.PRINT\_FORMAT\_MISMATCH.UNDESIRED  SV.FMT\_STR.PRINT\_IMPROP\_LENGTH  SV.FMT\_STR.PRINT\_PARAMS\_WRONGNUM.FEW  SV.FMT\_STR.PRINT\_PARAMS\_WRONGNUM.MANY  SV.FMT\_STR.SCAN\_FORMAT\_MISMATCH.BAD  SV.FMT\_STR.SCAN\_FORMAT\_MISMATCH.UNDESIRED  SV.FMT\_STR.SCAN\_IMPROP\_LENGTH  SV.FMT\_STR.SCAN\_PARAMS\_WRONGNUM.FEW  SV.FMT\_STR.SCAN\_PARAMS\_WRONGNUM.MANY  SV.FMT\_STR.UNKWN\_FORMAT |  |
| LDRA tool suite | 9.7.1 | 486 S  589 S | Fully implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-FIO47-a  CERT\_C-FIO47-b  CERT\_C-FIO47-c  CERT\_C-FIO47-d  CERT\_C-FIO47-e  CERT\_C-FIO47-f | There should be no mismatch between the '%s' and '%c' format specifiers in the format string and their corresponding arguments in the invocation of a string formatting function  There should be no mismatch between the '%f' format specifier in the format string and its corresponding argument in the invocation of a string formatting function  There should be no mismatch between the '%i' and '%d' format specifiers in the string and their corresponding arguments in the invocation of a string formatting function  There should be no mismatch between the '%u' format specifier in the format string and its corresponding argument in the invocation of a string formatting function  There should be no mismatch between the '%p' format specifier in the format string and its corresponding argument in the invocation of a string formatting function  The number of format specifiers in the format string and the number of corresponding arguments in the invocation of a string formatting function should be equal |
| PC-lint Plus | 1.4 | 492, 493, 494, 499, 557,  558, 559, 566, 705, 706,  719, 816, 855, 2401, 2402,  2403, 2404, 2405, 2406, 2407 | Fully supported |
| Polyspace Bug Finder | R2022b | CERT C: Rule FIO47-C | Check for format string specifiers and arguments mismatch (rule fully covered) |
| PRQA QA-C | 9.7 | 0161, 0162, 0163, 0164, 0165, 0166, 0167, 0168, 0169,  0170, 0171, 0172, 0173, 0174, 0175, 0176, 0177, 0178,  0179 [U], 0180 [C99], 0184 [U], 0185 [U], 0190 [U],  0191 [U], 0192 [U], 0193 [U], 0194 [U], 0195 [U], 0196 [U],  0197 [U], 0198 [U], 0199 [U], 0200 [U], 0201 [U], 0202 [I],  0204 [U], 0206 [U] | Partially implemented |
| PVS-Studio | 7.23 | V510, V576 |  |
| TrustInSoft Analyzer | 1.38 | Match format and arguments | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Functions should validate their parameters** |
| --- | --- | --- |
| **SQL Injection** | STD-AAPI00-C | Redundant testing by caller and by callee as a style of defensive programming is largely discredited in the C and C++ communities, the main problem being performance. The usual discipline in C and C++ is to require validation on only one side of each interface.  Requiring the caller to validate arguments can result in faster code because the caller may understand certain invariants that prevent invalid values from being passed. Requiring the callee to validate arguments allows the validation code to be encapsulated in one location, reducing the size of the code and making it more likely that these checks are performed in a consistent and correct fashion.  For safety and security reasons, this standard recommends that the called function validate its parameters. Validity checks allow the function to survive at least some forms of improper usage, enabling an application using the function to likewise survive. Validity checks can also simplify the task of determining the condition that caused the invalid parameter. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, setfile() and usefile() do not validate their parameters. It is possible that an invalid file pointer can be used by the library, corrupting the library's internal state and exposing a vulnerability. The vulnerability can be more severe if the internal state references sensitive or system-critical data. |
| /\* Sets some internal state in the library \*/  extern int setfile(FILE \*file);    /\* Performs some action using the file passed earlier \*/  extern int usefile();    static FILE \*myFile;    void setfile(FILE \*file) {  myFile = file;  }    void usefile(void) {  /\* Perform some action here \*/  } |

| **Compliant Code** |
| --- |
| Validating the function parameters and verifying the internal state leads to consistency of program execution and may eliminate potential vulnerabilities. In addition, implementing commit or rollback semantics (leaving program state unchanged on error) is a desirable practice for error safety. |
| /\* Sets some internal state in the library \*/  extern errno\_t setfile(FILE \*file);    /\* Performs some action using the file passed earlier \*/  extern errno\_t usefile(void);    static FILE \*myFile;    errno\_t setfile(FILE \*file) {  if (file && !ferror(file) && !feof(file)) {  myFile = file;  return 0;  }    /\* Error safety: leave myFile unchanged \*/  return -1;  }    errno\_t usefile(void) {  if (!myFile) return -1;    /\*  \* Perform other checks if needed; return  \* error condition.  \*/    /\* Perform some action here \*/  return 0;  } |

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

| **Principles(s):** *Validate input data* correlates with this standard as failing to validate the parameters in library functions may result in an access violation or a data integrity violation. Such a scenario indicates a flaw in how the library is used by the calling code. However, the library itself may still be the vector by which the calling code's vulnerability is exploited |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | LANG.STRUCT.UPD | Unchecked parameter dereference |
| Parasoft C/C++ test | 2022.2 | CERT\_C-API00-a | The validity of parameters must be checked inside each function |
| PC-lint Plus | 1.4 | 413, 613, 668 | Partially supported: reports use of null pointers including function parameters which are assumed to have the potential to be null |
| PVS-Studio | 7.23 | V781 | detecting potential errors and security vulnerabilities in code |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Allocate sufficient memory for an object** |
| --- | --- | --- |
| **Memory Protection** | STD- MEM35-C | The types of integer expressions used as size arguments to malloc(), calloc(), realloc(), or aligned\_alloc() must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.  Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure. (See MEM33-C. Allocate and copy structures containing a flexible array member dynamically.) Use the correct type of the object when computing the size of memory to allocate. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, inadequate space is allocated for a struct tm object because the size of the pointer is being used to determine the size of the pointed-to object: |
| #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** |
| --- |
| In this compliant solution, the correct amount of memory is allocated for the struct tm object. When allocating space for a single object, passing the (dereferenced) pointer type to the sizeof operator is a simple way to allocate sufficient memory. Because the size of operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior. |
| #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):** 7. Sanitize Data Sent to Other Systems, 8. Practice Defense in Depth, and 1. Validate Input Data could all potentially be correlated with this standard as providing invalid size arguments to memory allocation functions can lead to buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process. |
| --- |

**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 behavior resulting from invalid memory accesses is reported by Astrée. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM35 |  |
| CodeSonar | 7.2p0 | 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) ... |
| Coverity | 2017.07 | BAD\_ALLOC\_STRLEN  SIZECHECK (deprecated) | Partially implemented  Can find instances where string length is miscalculated (length calculated may be one less than intended) for memory allocation purposes. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary  Finds memory allocations that are assigned to a pointer that reference objects larger than the allocated block |
| Helix QAC | 2022.4 | C0696, C0701, C1069, C1071, C1073, C2840  DF2840, DF2841, DF2842, DF2843, DF2935, DF2936, DF2937, DF2938 |  |
| Klocwork | 2022.4 | INCORRECT.ALLOC\_SIZE  SV.TAINTED.ALLOC\_SIZE |  |
| LDRA tool suite | 9.7.1 | 400 S, 487 S, 115 D | Enhanced enforcement |
| Splint | 3.1.1 |  |  |
| Parasoft C/C++ test | 2022.2 | CERT\_C-MEM35-a | Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via ‘malloc’, ‘calloc’, or’realloc’ function |
| PC-lint Plus | 1.4 | 433, 816 | Partially supported |
| Polyspace Bug Finder | R2022b | CERT C: Rule MEM35-C | Checks for:   * Pointer access out of bounds * Memory allocation with tainted size   Rule fully covered. |
| PRQA QA-C | 9.7 | 0696, 0701, 1069, 1071, 1073, 2840, 2841, 2842, 2843, 2935, 2936, 2937, 2938 |  |
| PRQA QA-C++ | 4.4 | 2840, 2841, 2842, 2843, 2935, 2936, 2937, 2938 |  |
| PVS -Studio | 7.23 | V531, V635, V781 |  |
| RuleChecker | 22.04 | malloc-size-insufficient | Partially checked |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively detects undefined behavior |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | STD-DCL03-C | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **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. Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. This means that it is usually far away from the definition of the actual structure to which it refers. The diagnostic occurs only at runtime and only if the code path containing the assertion is executed. |
| #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. Using #error directives allows for clear diagnostic messages. Because this approach evaluates assertions at compile time, there is no runtime penalty. |
| 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):** *2. Heed Compiler Warnings and 9. Use Effective Quality Assurance Techniques* correlates with this standard as static assertion is a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities at compile time. The absence of static assertions, however, does not mean that code is incorrect. |
| --- |

**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 | 7.2p0 | (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 |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-ERR51-CPP | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surround try blocks of the same thread. |

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    try {      f();    } catch (...) {      // Handle error    }  } |

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

| **Principles(s):** *3. Architect and Design for Security Policies* and *10. Adopt a Secure Coding Standard* correlate with this standard as allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial-of-service attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ - ERR51 |  |
| CodeSonar | 7.2p0 | LANG.STRUCT. UCTCH | Unreachable Catch |
| Helix QAC | 2022.4 | C++4035, C++4036, C++4037 |  |
| Klocwork | 2022.4 | MISRA.CATCH.ALL |  |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Polyspace Bug Finder | R2022b | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4037 |  |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Avoid information leakage when passing a class object across a trust boundary** |
| --- | --- | --- |
| Data Leak | STD-DCL55-CPP | Non static data members of a (non-union) class with the same access control are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified. Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions and virtual base classes. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example runs in kernel space and copies data from arg to user space. However, padding bits may be used within the object, for example, to ensure the proper alignment of class data members. These padding bits may contain sensitive information that may then be leaked when the data is copied to user space, regardless of how the data is copied. |
| #include <cstddef>    struct test {    int a;    char b;    int c;  };    // Safely copy bytes to user space  extern int copy\_to\_user(void \*dest, void \*src, std::size\_t size);    void do\_stuff(void \*usr\_buf) {    test arg{1, 2, 3};    copy\_to\_user(usr\_buf, &arg, sizeof(arg));  } |

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

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

| **Principles(s):** *7. Sanitize Data Sent to Other Systems* is correlated with this standard due to padding bits might inadvertently contain sensitive data such as pointers to kernel data structures or passwords. A pointer to such a structure could be passed to other functions, causing information leakage. |
| --- |

**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++ -DCL55 |  |
| CodeSonar | 7.2p0 | MISC.PADDING. POTB | Padding Passed Across a Trust Boundary |
| Helix QAC | 2022.4 | DF4941, DF4942, DF4943 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-DCL55-a | A pointer to a structure should not be passed to a function that can copy data to the user space |
| Polyspace Bug Finder | R2022b | CERT C++: DCL55-CPP | Checks for information leakage due to structure padding (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Detect and handle standard library errors** |
| --- | --- | --- |
| Error Handling | STD-ERR33-C | The majority of the standard library functions, including I/O functions and memory allocation functions, return either a valid value or a value of the correct return type that indicates an error (for example, −1 or a null pointer). Assuming that all calls to such functions will succeed and failing to check the return value for an indication of an error is a dangerous practice that may lead to unexpected or undefined behavior when an error occurs. It is essential that programs detect and appropriately handle all errors in accordance with an error-handling policy.  The successful completion or failure of each of the standard library functions listed in the following table shall be determined either by comparing the function’s return value with the value listed in the column labeled “Error Return” or by calling one of the library functions mentioned in the footnotes |

| **Noncompliant Code** |
| --- |
| The function utf8\_to\_wcs() attempts to convert a sequence of UTF-8 characters to wide characters. It first invokes setlocale() to set the global locale to the implementation-defined en\_US.UTF-8 but does not check for failure. The setlocale() function will fail by returning a null pointer, for example, when the locale is not installed. The function may fail for other reasons as well, such as the lack of resources. Depending on the sequence of characters pointed to by utf8, the subsequent call to mbstowcs() may fail or result in the function storing an unexpected sequence of wide characters in the supplied buffer wcs. |
| #include <locale.h>  #include <stdlib.h>    int utf8\_to\_wcs(wchar\_t \*wcs, size\_t n, const char \*utf8,                  size\_t \*size) {    if (NULL == size) {      return -1;    }    setlocale(LC\_CTYPE, "en\_US.UTF-8");    \*size = mbstowcs(wcs, utf8, n);    return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution checks the value returned by setlocale() and avoids calling mbstowcs() if the function fails. The function also takes care to restore the locale to its initial setting before returning control to the caller. |
| #include <locale.h>  #include <stdlib.h>    int utf8\_to\_wcs(wchar\_t \*wcs, size\_t n, const char \*utf8,                  size\_t \*size) {    if (NULL == size) {      return -1;    }    const char \*save = setlocale(LC\_CTYPE, "en\_US.UTF-8");    if (NULL == save) {      return -1;    }      \*size = mbstowcs(wcs, utf8, n);    if (NULL == setlocale(LC\_CTYPE, save)) {      return -1;    }    return 0;  } |

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

| **Principles(s):** *1. Validate Input Data, 7. Sanitize Data Sent to Other Systems, 10. Adopt a Secure Coding Standard* all could potentially correlate with this standard as failing to detect error conditions can lead to unpredictable results, including abnormal program termination and denial-of-service attacks or, in some situations, could even allow an attacker to run arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | error-information-unused  error-information-unused-computed | Partially checked |
| Axivion Bauhaus | 7.2.0 | CertC-ERR33 |  |
| CodeSonar | 7.2p0 | LANG.FUNCS.IRV  LANG.ERRCODE. NOTEST  LANG.ERRCODE.NZ | Ignored return value  Missing Test of Error Code  Non-zero Error Code |
| Compass/ROSE |  |  | Can detect violations of this recommendation when checking for violations of EXP12-C. Do not ignore values returned by functions and EXP34-C. Do not dereference null pointers |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.8  MISRA C 2012 Rule 22.9  MISRA C 2012 Rule 22.10 | Implemented |
| Helix QAC | 2022.4 | C3200  C++3802, C++3803, C++3804  DF2820, DF2821, DF2822, DF2823, DF2824, DF2930, DF2931, DF2932, DF2933, DF2934 |  |
| Klocwork | 2022.4 | NPD.CHECK. MUST  NPD.FUNC. MUST  SV.RVT.RETVAL\_NOTTESTED |  |
| LDRA tool suite | 9.7.1 | 80 D | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-ERR33-a  CERT\_C-ERR33-b  CERT\_C-ERR33-c  CERT\_C-ERR33-d | The value returned by a function having non-void return type shall be used  The value returned by a function having non-void return type shall be used  Avoid null pointer dereferencing  Always check the returned value of non-void function |
| Parasoft Insure++ |  |  | Runtime analysis |
| PC-lint Plus | 1.4 | 534 | Partially supported |
| Polyspace Bug Finder | R2022b | CERT C: Rule ERR33-C | Checks for:   * Errno not checked * Return value of a sensitive function not checked * Unprotected dynamic memory allocation   Rule partially covered. |
| PRQA QA-C | 9.7 | 3200 | Partially implemented |
| PRQA QA-C++ | 4.4 | 2820, 2821, 2822, 2823, 2824, 2930, 2931, 2932, 2933, 2934, 3802, 3803, 3804 |  |
| RuleChecker | 22.04 | Error-information-unused | Partially checked |
| TrustInSoft Analyzer | 1.38 | Pointer arithmetic | Exhaustively verified |

Coding Standard 10

| **Coding Standard** | **Label** | **Do not use pointer arithmetic on polymorphic objects** |
| --- | --- | --- |
| Pointer Arithmetic | STD-CTR56-CPP | For addition or subtraction, if the expressions P or Q have type “pointer to cv T”, where T is different from the cv-unqualified array element type, the behavior is undefined. In particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, f() accepts an array of S objects as its first parameter. However, main() passes an array of T objects as the first argument to f(), which results in undefined behavior due to the pointer arithmetic used within the for loop. |
| #include <iostream>    // ... definitions for S, T, globI, globD ...    void f(const S \*someSes, std::size\_t count) {    for (const S \*end = someSes + count; someSes != end; ++someSes) {      std::cout << someSes->i << std::endl;    }  }    int main() {    T test[5];    f(test, 5);  } |

| **Compliant Code** |
| --- |
| Instead of having an array of objects, an array of pointers solves the problem of the objects being of different sizes, as in this compliant solution. |
| #include <iostream>    // ... definitions for S, T, globI, globD ...    void f(const S \* const \*someSes, std::size\_t count) {    for (const S \* const \*end = someSes + count; someSes != end; ++someSes) {      std::cout << (\*someSes)->i << std::endl;    }  }    int main() {    S \*test[] = {new T, new T, new T, new T, new T};    f(test, 5);    for (auto v : test) {      delete v;    }  } |

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

| **Principles(s):** *8. Practice Defense in Depth and 10. Adopt a Secure Coding Standard* are principles that can be correlated with this standard as using arrays polymorphically can result in memory corruption, which could lead to an attacker being able to execute arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++ -CTR56 |  |
| CodeSonar | 7.2p0 | LANG.STRUCT. PARITH | Pointer Arithmetic |
| Helix QAC | 2022.4 | C++3073 |  |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-CTR56-a  CERT\_CPP-CTR56-b  CERT\_CPP-CTR56-c | Don't treat arrays polymorphically  A pointer to an array of derived class objects should not be converted to a base class pointer  Do not treat arrays polymorphically |
| LDRA tool suite | 9.7.1 | 567 S | Enhanced Enforcement |
| PRQA QA-C++ | 4.4 | 3073 |  |
| PVS-Studio | 7.23 | V777 |  |

### 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.

Automation will be utilized to ensure and enforce coding standards in several ways corresponding with the Green Pace DevOps process of development, security, and operations which will complement its infrastructure with respect to pre-production and production phases. Beginning in pre-production and formulation of a plan, the process will be analyzed for current processes, systems, and technology in determination of automation. The design phase will consist of defining the architecture and systems such as determining requirements, security design, performance design, and risk analysis. The build phase will include code development with testing, debugging, and particulars such as release management. This is followed by verification testing which will be implemented using the proper testing platform such as static analysis and vulnerability scanners such as Nessus or OpenVAS.

With an agile methodology in position, the production phase requires consistent collaboration with security settings and health checks from pre-production. Requirements will be designed and built with respect to authentication and authorization such as LDAP to manage user credentials and control access to resources. Tools will be utilized to monitor and detect issues and anomalies such as Application Performance Monitoring from New Relic or AppDynamics. Log Monitoring is a necessary option as well with Logstash to collect and analyze data from varying sources. Implementation of response criteria will include necessary tools from the design phase such as Web Application Firewalls or Cloud Security Tools to mitigate attacks on applications and infrastructure. The process then moves to maintaining and stabilizing the software making assurances it is reliable, stable, and performs as necessary.

### 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-INT35-C | Low | Unlikely | High | P1 | 3 |
| STD-FLP37-C | Low | Unlikely | Medium | P2 | 3 |
| STD-FIO47-C | High | Unlikely | Medium | P6 | 2 |
| STD-AAPI00-C | Medium | Unlikely | High | P2 | 3 |
| STD-MEM35-C | High | Probable | High | P6 | 2 |
| STD-DCL03-C | Low | Unlikely | High | P1 | 3 |
| STD-ERR51-CPP | Low | Probable | Medium | P4 | 3 |
| STD-DCL55-CPP | Low | Unlikely | High | P1 | 3 |
| STD-ERR33-C | High | Likely | Medium | P18 | 1 |
| STD-CTR56-CPP | High | Likely | High | P9 | 2 |

### 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 refers to data that is stored in an encrypted form when it is not in use. Process of encoding sensitive data transmitted over a network to prevent unauthorized access. In REST, encryption is used to secure the communication between a client and a server, and to protect sensitive data, such as passwords, credit card numbers, and other confidential information, from being intercepted and read by unauthorized parties even if the device is stolen. |
| Encryption at flight | When data is encrypted at flight, it helps to ensure that information remains confidential and secure during transmission. It is transformed into a code that is unreadable by anyone who does not have the key to decrypt it. This means that even if an attacker were to intercept the encrypted data, they would not be able to access the sensitive information contained within it. |
| Encryption in use | Data that is encrypted while it is being processed by an application or service. This ensures that the data remains protected even if the device or network it is being processed on is compromised. It is often implemented with other security measures, such as password protection, two-factor authentication, and firewalls, to provide multiple layers of protection for sensitive information |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Determines who is trying to access the system and to ensure that only authorized users are able. Process of verifying the identity of a user, device, or system. Authentication is usually accomplished by requiring the user to provide a username and password. |
| Authorization | Authorization helps to enforce security policies and to prevent unauthorized access to sensitive data. It is process of granting or denying access to resources or services based on the user's identity. After a user has been authenticated, authorization is used to determine what the user is allowed to do with respect to making changes or their level of access. |
| Accounting | Used to monitor resource usage, to detect security incidents, and to audit compliance with security policies. This process keeps track of user activity and resource usage such as files accessed. It provides an avenue to provide a record of who has accessed a system, what they did, and when they did it. |

**\***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 | 01/25/2023 | Coding Standards Implementation | Joshua Pardue |  |
| 1.2 | 02/11/2023 | Risk Assessment/Encryption Framework | Joshua Pardue |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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