**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 | This principle defines the process of verifying that input data is not corrupt, containing malicious scripts, and contains the appropriate credentials. Input data verification is conducted in several ways by testing for input data size, data type, entry exploit script keywords and other volatile data inputs. These tests can be conducted in multiple ways include if-else statements, assert functions, and try-catch probes. |
| 1. Heed Compiler Warnings | This principle is to remind developers not to ignore a compiler when warnings arise regarding the initialization of variables, referencing of pointers and other memory or logic complications. Often programmers will overlook uninitialized variables that generate warnings during compilation and can create logic complications if the user or system does not give a starting value to a control variable. When utilizing pointers there are also a variety of warnings that can take place since improperly referenced pointer introduce risks relating to unintentionally over-writing data with system memory and therefore compiler warnings must be rectified. |
| 1. Architect and Design for Security Policies | This principle focuses upon the need for system blueprints to integrate security activities and structures rather than developer implement security measures after system architecture is decided upon. During development it is therefore essential to decide the appropriate layers of security for an application and determine how they will manifest within the various user experiences. It is also wise to remember that security protocols may need to be expanded upon post-deployment and therefore the designs should ensure that these operations can be conducted with ease. |
| 1. Keep It Simple | This principle is a paramount reminder to developers that software systems are intricate in nature and should not be made unnecessarily dense to allow for ease of implementation and integration. Concise code will also ensure that other developers will be able to quickly review systems when preparing updates or designing the next iteration of a software package. Furthermore, maintaining simplicity in code structures ensure that faults in data as well as logic can be quickly found to reduce development times by weeks if not months. |
| 1. Default Deny | This principle is to direct system developers to deny users access by default, only permitting users access to elements explicitly stated by the user class’ credentials. Security protocols of this sort maintain strong differentiations between public/guest users, system moderator accounts and administrative/owner users’ level of entitlement to ensure that base level system operators are not allowed to access or alter privileged data as well as routines. As a standard methodology, persistent denial of access for users prior to credential validation protects all systems exposed to threats of exposed networks. |
| 1. Adhere to the Principle of Least Privilege | This principle focuses upon the requirements of the various system actors, specifically the minimum system privileges needed for the actors to perform their standard procedures. Similar to the Principle of Default Deny, least privilege promotes developers to deny users access to any system data or operations that are not necessary as for the user class’ standard system access and operation. To apply least privilege, system designers often develop privilege levels to separate software systems for guest access, moderator access and administrator access allowing each class the bare-minimum privileges required to perform their standard activities within the program. |
| 1. Sanitize Data Sent to Other Systems | This principle directs developers to check data for vulnerabilities before allowing applications to transfer the data to other systems. When transferring data between systems there is the potential for injection attacks as well as other hacking exploits and hence it is essential that data is checked for corruption, unanticipated script and other signs of exploitation. Sanitizing data is a vital network process to ensure that neither the client application nor server application is subject to hacking exploits due to corrupt data or hacker scripts. |
| 1. Practice Defense in Depth | This principle guides developers to consider the security requirements for an application. For example, some applications may only need one level of security at the point of data entry while other system may require multiple layers of security for the data entry system, data processing system and the remote server credential confirmation system. When developing security for large, multi-component systems it is essential to understand the appropriate layers of security necessitated by the types of data and processes active within the system. |
| 1. Use Effective Quality Assurance Techniques | This principle acts to remind developers to use proven techniques that will ensure software applications run appropriately under standard operation, boundary testing and hacking attacks. Quality assurance is often done through battery assertion testing, that is a series of automated condition tests that will confirm a programs reaction to standard and non-standard data. But quality assurance is also conducted through penetration testing to ensure that system security will hold up under pressure. |
| 1. Adopt a Secure Coding Standard | This principle reminds developers to focus on aligning their security practices with one of the current secure coding standards, such as OWASP, CVE, etc. When selecting a secure coding standard, it is essential to consider if the industry that the software will be applied to has dictated its own standards for secure coding that must be respected additionally to the general standards for secure coding adopted by the developer as multi-industry guidelines. |

### 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** | **Obey the one-definition rule** |
| --- | --- | --- |
| **Data Type** | DCL60-CPP | This rule instructs developers not to reuse the same variable or class name within the same namespace since this will create conflicts when referencing/calling the variable or data structure. |

| **Noncompliant Code** |
| --- |
| The following segment of code is non-complaint with coding standards since both struct A and class A occupy the same namespace with the same name that creates a collision when attempting to call ‘A’. |
| struct A { int anInteger; };  class A { public: int anInteger; }; |

| **Compliant Code** |
| --- |
| The following example shares a code compliant variant of the previous code that enforces unique name schemes when identifying the data structures and variables utilized in an application. |
| struct structA { int anInteger; };  class classA {public: int anInt; }; |

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

| **Principles(s):**  Validate Input Data relates to this standard since this rule describes a need to avoid data corrupt and the preservation of order during data procedures.  Keep It Simple relates to this policy as the one-definition rule prevents confusion between variables and functions of similar names. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | Type-compatibility  Definition-duplicate  Undefined-extern  Undefined-extern-pure-virtual  External-file-spreading  Type-file-spreading | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-DCL60-a | A class, union or enum name (including qualification, if any) shall be a unique identifier |
| Polyspace Bug Finder | R2024a | CERT C++: DCL60-CPP | Checks for inline constraints not respected (rule partially covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | INT31-C | This rule reminds developers to consider how integers will be utilized by a system, i.e. as control values, or representative of subject matter data. Developers must also consider how accurate the integer value must remain through processes as conversion from integer to an unsigned short variable or float variable could result in loss of data. |

| **Noncompliant Code** |
| --- |
| This segment of code represents a loss of data as the maximum unsigned integer value, 4294967295, is being passed to a signed short integer, with a max value of 32767. |
| unsigned Int value = UINT\_MAX;  short newValue = value; |

| **Compliant Code** |
| --- |
| The follow example shares a proper integer conversion in which no data is lost since the integer value of, 4294967295, is being passed to a long integer type variable that has a maximum value of 9223372036854775807. |
| unsigned int value = UINT\_MAX;  long newValue = value; |

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

| **Principles(s):**  Heed Compiler Warnings relates to this coding standard as integer conversions will offer produce compiler warning relating to data overrun or data loss.  Sanitize Data Sent to Other Systems relates to this standard since this improper integer conversion will produce a domino-effect of system corruption should the faulty data be passed to other systems. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity\* | 2017.07 | NEGATIVE\_RETURNS  REVERSE\_NEGATIVE  MISRA\_CAST | Can find array accesses, loop bounds, and other expressions that may contain dangerous implied integer conversions.  Can find instances where a negative check occurs after the negative value has been used for another task.  Can find instances where an integer expression is implicitly converted to an improper type. |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset () cannot be represented as unsigned char |
| Polyspace Bug Finder | R2024a | CERT C: Rule INT31-C | Checks for integer conversion overflow, improper call to memset, tainted sign change conversion. Rule partially covered. |
| TrustInSoft Analyzer | 1.38 | Signed\_downcast | Exhaustively verified. |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to modify string literals** |
| --- | --- | --- |
| **String Correctness** | STR30-C | This rule implies that errors of various kinds will take place should a developer attempt to improperly access or modify string literals. Instead, string literals are to be handled through character arrays with the addition of pointers for referencing. |

| **Noncompliant Code** |
| --- |
| This segment of code will generate a compilation error that will prevent execute since mkdir() requires string constant as a parameter |
| Void func(){ mkdir(“../newFolder”); } |

| **Compliant Code** |
| --- |
| The following example shares the proper integration of mkdir() with the string literal assigned to a char array before passing the new Folder pathway to mkdir(). |
| Void func(){ const char str[] = “../newFolder”; mkdir(str);} |

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

| **Principles(s):**  Validate Input Data relates to this rule since string literals are implemented to avoid corrupting system functions.  Heed Compiler Warnings relates to this standard as the compiler will often warning developers when the implementation of a string will result in potentially erratic behavior.  Default Deny relates to this standard since denying users the ability to modify string literals becomes an essential security feature allowing the system greater control over how users interact with software package. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to “char \*” |
| LDRA tool suite | 9.7.1 | 157 S | Partially implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified  Do not modify string literals |
| Polyspace Bug Finder | R2024a | CERT C: Rule STR30-C | Checks for writing to const qualified object (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **SQL Injection** | STR31-C | This rule is paramount to ensure that buffer overflow does not take place, greatly reducing the risk of hacking attacks such as SQL Injection. Sufficient space for buffers can often be determined by anticipating and restricting user inputs. |

| **Noncompliant Code** |
| --- |
| This segment of code will generate an error since the terminating null is not accounted for in this implementation of copy() and therefore may cause a memory overflow. |
| for (i = 0; src[i] && (i < n); ++i) {       dest[i] = src[i];     } |

| **Compliant Code** |
| --- |
| The following example shows the proper way to respect the boundaries of the variable dest by accounting for the terminating null value at the end of the dest character array. |
| for (i = 0; src[i] && (i < n - 1); ++i) {       dest[i] = src[i];     } |

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

| **Principles(s):**  Validate Input Data relates to this standard as there must be sufficient memory allocated to a string for input data to be secure and valid.  Keep It Simple relates to this standard since the simplest approach to defining strings involves restricting user input or a transfer buffers size to only what is acceptable within the string.  Sanitize Data Sent to Other Systems relates to this rule since strings size and null termination will need to be confirmed during the sanitation process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string functions that could result in buffer overflow.  Detects potential buffer overruns. |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Detects buffer overrun, type overrun, no space for null terminator.  Warns for usage of functions that are prone to internal buffer overflow. |
| Parasoft C/C++ test | 2023.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoids accessing out of bounds index for arrays, buffers overflow, buffer write overflow from tainted data, usage of unsafe string functions. |
| Polyspace Bug Finder | R2024a | CERT C: Rule STR31-C | Checks for usage of dangerous standard functions, missing null in string arrays, buffer overflow from improper string format, destination buffer overflow and insufficient transfer buffer size. Rule is partially covered. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Allocate sufficient memory for an object** |
| --- | --- | --- |
| **Memory Protection** | MEM35-C | This rule directs developer to consider the maximum amount of memory data objects and structures will need to perform operations. Should this rule not be respected systems will fail when limits are met and exceeded, increasing hacking vulnerabilities and failing to deliver a quality product for clients. |

| **Noncompliant Code** |
| --- |
| The following method shares an instance in which a long integer pointer has been allocated a chunk of memory only size for an standard integer, this will cause issues as the pointer will bee able to reference long integers but will experience buffer overflow due to undersized memory allocation. |
| void func (size\_t len){  long \*aPtr;  aPtr = (long \*)malloc(len \* sizeof(int));  free(aPtr);  } |

| **Compliant Code** |
| --- |
| This example shares a properly allocated memory block for the long pointer ‘aPtr’ since the size of the memory will be described by the long integer type rather than the standard integer type. |
| Void func(size\_t len){  Long \*aPtr;  aPtr = (long \*)malloc(len \* sizeof(long));  free(aPtr);  } |

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

| **Principles(s):**  Validate Input Data relates to this standard since adequate memory must be reserved for input data to be processed by a system.  Architect and Design for Security Policies relates to this rule as the portion of memory allocated for input objects, network objects, interface objects and database objects must be understood before security measures can be properly designed to test for proper object states.  Sanitize Data Sent to Other System relates to this standard since memory overrun can become a concern for post-encryption data being transfer across a network. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Malloc-size-insufficient | Partially checked.  Reports all undefined behavior resulting from invalid memory access |
| Coverity | 2017.07 | BAD\_ALLOC\_STRLEN  SIZECHECK (deprecated) | Can find miscalculations of string length.  Can detect memory allocations where the referenced object is larger than the allocated memory. |
| Polyspace Bug Finder | R2024a | CERT C: Rule MEM35-C | Checks for out of bounds pointer access, corrupt size memory allocation.  Rule is partially covered. |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively detects undefined behavior |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Incorporate diagnostic tests using assertions** |
| --- | --- | --- |
| **Assertions** | MSC11-C | This rule reminds developers to integrate assertion statements for automated testing of system procedures. The addition of assert() into program structure can easily provide insight of the condition of the data as it moves from input stream, to processing and finally to the output stream. |

| **Noncompliant Code** |
| --- |
| This method shows a function that improperly introduces an assert() for testing that a pointer has been properly allocated memory, since the assert function provides only a true or false return for the parameter statement this will not properly terminate operations. Instead, this assert must be replaced with a return statement or secondary memory allocation attempt. |
| char \*dupstring(const char \*c\_str) {    size\_t len;    char \*dup;      len = strlen(c\_str);    dup = (char \*)malloc(len + 1);    assert(NULL != dup);      memcpy(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| The following module share the proper integration of assertions that work to confirm the validity of existing error protection code rather than acting independently to correct errors within the system. |
| char[] inFunc(){  int maxLen = 10;  char[maxLen] inStr;    for(int i = 0; i < maxLen -1; i++){  inStr[i] = (char) \_getch();  if(inStr[i] < 33 || inStr[i] > 126)  inStr[i] = ‘!’;  assert(inStr[i] > 32);  assert(inStr[i] < 127);  }  } |

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

| **Principles(s):**  Validate Input Data relates to this standard as assertions can integrated to test the validity of user input.  Architect and Design for Security Policies relates to this rule since assertions are a strategic tool for checking the validity of data and the outcomes of key procedures.  Keep It Simple relates to this rule as battery assertion testing is a quick and easy way to test data and function outcomes without implementing lengthy procedures.  Use Effective Quality Assurance Techniques relates to this standard since assertions allow for automated system testing that will effortlessly allow developers to test the system for a multitude of user interactions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++ test | 2023.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Restore prior object state on method failure** |
| --- | --- | --- |
| **Exceptions** | ERR03-J | This rule focuses developers to undo any changes that a method may perform, or simply empty buffers, should an exception arise or failure in processing be detected. This is an essential consideration to protect data from becoming corrupt or hackers from passing malicious script into protected operations. |

| **Noncompliant Code** |
| --- |
| This try-catch statement code does not conform to this standard since the length, width and height variables will remain altered even the case an Exception has been throw, this could potentially cause additional errors in operation to occur later in procedures when users attempt to re-enter data or the program is directed to perform the operation a second time. |
| length += PADDING;      width  += PADDING;      height += PADDING;  try {        if (length <= PADDING || width <= PADDING || height <= PADDING ||          length > MAX\_DIMENSION + PADDING || width > MAX\_DIMENSION +  PADDING || height > MAX\_DIMENSION + PADDING || weight <= 0 ||  weight > 20) {          throw new IllegalArgumentException();        }          int volume = length \* width \* height;        length -= PADDING; width -= PADDING; height -= PADDING; // Revert        return volume;      } catch (Throwable t) {        MyExceptionReporter mer = new MyExceptionReporter();        mer.report(t); // Sanitize        return -1; // Non-positive error code      } |

| **Compliant Code** |
| --- |
| The following example shows the proper was to protect system data by reverting data buffers to their original states, or values, upon failure of an operation. The original state of the data may refer to the initial input value or the value upon entering the failed process depending upon the type of data and procedure conducted. |
| length += PADDING;    width  += PADDING;    height += PADDING;  try {      if (length <= PADDING || width <= PADDING || height <= PADDING ||        length > MAX\_DIMENSION + PADDING ||        width > MAX\_DIMENSION + PADDING ||        height > MAX\_DIMENSION + PADDING ||        weight <= 0 || weight > 20) {        throw new IllegalArgumentException();      }        int volume = length \* width \* height;      return volume;    } catch (Throwable t) {      MyExceptionReporter mer = new MyExceptionReporter();      mer.report(t); // Sanitize      return -1; // Non-positive error code    } finally {      // Revert      length -= PADDING; width -= PADDING; height -= PADDING;    } |

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

| **Principles(s):**  Architect and Design for Security Policies relates to this standard since the integration of this rule can prevent a catastrophic system failure and allow the user the opportunity to attempt other tasks within the system using the same object/instance, rather than restart the system with a new object/instance.  Keep It Simple relates to this rule as the system will ignore a failed procedure and return an unaltered object rather than an alter object with potentially corrupt data that will complicate the object’s interactions with the system.  Default Deny relates to this standard as the default behavior within the software package shall be to refuse a function the right to alter an object unless successful execution of the procedure has been confirmed.  Use Effective Quality Assurance Techniques relates to this standard since refusing to alter an object upon failure of an operation can prevent the software package from crashing and prevent the object from carrying corrupt data to other potentially more sensitive processes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft Jtest | 2024.1 | CERT.ERR03.REVOBJ | Restore prior object state on method failure |

#### Coding Standard 8

| **Coding Standard** | **Label** | **All exit handlers must return normally** |
| --- | --- | --- |
| **Termination Handling** | ENV32-C | This rule instructs developers to never allow for exit or termination as a standard part of error handling, or system termination. |

| **Noncompliant Code** |
| --- |
| This function highlights a failure to throw an exception, display an error code and a double call while immediately terminating the system should an error condition occur. These types of error handling behaviors can be highly compromising as it does not allow technical experts time to understand the error that has occurred and potentially correct it while the data is still active within the software system. |
| void exitFunc(void) {    extern int some\_condition;    if (some\_condition) {      exit(0);    }    return;  }  Int main(void){  If(atexit(exitFunc) != 0)  /\* Handle error \*/  Return 0;  } |

| **Compliant Code** |
| --- |
| The following code shares the proper implement of exit() for error handling since there is only one call at the termination of the program rather than an initial call to exit at execution of the exitFunc() as well as a call upon execution of the return statement. This code also allows error condition handling rather than immediate software package termination. |
| void exitFunc(void) {    extern int some\_condition;    if (some\_condition) {      /\* error mitigation code \*/    }    return;  }  int main(void){  if(atexit(exitFunc) != 0)  /\* Handle error \*/  return 0;  } |

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

| **Principles(s):**  Architect and Design for Security Policies relates to this standard since security measures must be designed properly to ensure that the software package does not terminate when encountering errors but rather responses in a fluid manner.  Practice Defense in Depth relates to this rule as a program that crashes under standard fault and error handling will be susceptible to hacking exploited that will require the program to had additional security measures to protect confidential data, or simply repair error handling structures as a first line of defense against attackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT  BADFUNC.LONGJMP | Checks usage of abort, exit and longjmp |
| Compass/ROSE | - | - | Detects violations to this rule, specifically that all functions registered to atexit() do not call other exit functions. |
| Parasoft C/C++ test | 2023.1 | CERT\_C-ENV32-a | Properly define exit handlers |
| Polyspace Bug Finder | R2024a | CERT C: Rule ENV32-C | Checks for abnormal termination of exit handler |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Restrict privileges when spawning child processes** |
| --- | --- | --- |
| **Least Privilege** | WIN02-C | This rule directs developers to restrict privileges for child classes, functions and other systems to only allow access to parent system operations and data that are essential to standard operations. Access restrictions are a simple and powerful way to protect sensitive data and processes from hacking attacks and corruption that could hinder overall system performance. |

| **Noncompliant Code** |
| --- |
| The launch\_notepad() highlights an unrestricted privilege level for access of the user that can result in the user accessing other user data, system setting and other applications. |
| void launch\_notepad(void) {    PROCESS\_INFORMATION pi;    STARTUPINFO si;      ZeroMemory(&si, sizeof(si));    si.cb = sizeof( si );    if (CreateProcess(TEXT("C:\\Windows\\Notepad.exe"), NULL, NULL, NULL,  TRUE,0, NULL, NULL, &si, &pi )) {      /\* Process has been created; work with the process and wait for it to         terminate. \*/      WaitForSingleObject(pi.hProcess, INFINITE);      CloseHandle(pi.hThread);      CloseHandle(pi.hProcess);    }  } |

| **Compliant Code** |
| --- |
| The following version of launch\_notepad() includes the necessary operations to control the user’s privilege level when launching the notepad application. A user’s privilege level is controlled in this function by copying the process’ security token then lowering the privilege level of the token copy before re-assigning this adjusted token to the process instance to ensure that the user will have the least security entitlement required for standard application usage. |
| void launch\_notepad(void) {    /\* Low level; see table for integrity level string names \*/    const char \*requested\_sid = "S-1-16-4096";    HANDLE token\_cur, token\_dup;    /\* Get the current process' security token as a starting point, then  Modify a duplicate so that it runs with a fixed integrity level. \*/    if (OpenProcessToken(GetCurrentProcess(), TOKEN\_DUPLICATE |                         TOKEN\_ADJUST\_DEFAULT | TOKEN\_QUERY |                         TOKEN\_ASSIGN\_PRIMARY, &token\_cur)) {      if (DuplicateTokenEx(token\_cur, 0, NULL, SecurityImpersonation,                           TokenPrimary, &token\_dup)) {        if (adjust\_token\_integrity\_level(token\_dup, requested\_sid))           launch\_notepad\_as\_user(token\_dup);        CloseHandle(token\_dup);      }      CloseHandle(token\_cur);    }  } |

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

| **Principles(s):**  Architect and Design for Security Policies relates to this standard since determining the levels of entitlement that child processes should require for standard operations will be determined during the software analysis and design stages of development.  Default Deny relates to this rule as the standard initialization of a child process should allow for no access privileges, ensuring that confidential data and processes are protected.  Adhere to the Principle of Least Privileges relates to this standard as restricting newly generated child process entitlements will defend against unwarranted public users from engaging will private user, moderator or admin specific of system interactions.  Practice Defense in Depth relates to this standard since restriction of newly initialized child processes will provide an additional layer of security to the software package by preventing the process from gaining undue access to system operations and information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | BADFUNC.CREATEPROCESS  BADFUNC.CREATETHREAD | Checks usage of CreateProcess and CreateThread |
| PC-lint Plus | 1.4 | 586 | Rule is fully supported |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Use sizeof to determine the size of a type or variable** |
| --- | --- | --- |
| **Sanitize data** | EXP09-C | This rule focuses developers upon the need to confirm the actual size of a type or variable, through the implementation of sizeof() rather than assuming the size based on a previously defined variable. When sanitizing data, it is vital to ensure that both client and server know the anticipated size of the data package transferred so that alteration or hick-hiker code can be isolated. |

| **Noncompliant Code** |
| --- |
| This segment of code shares an instance where data may be lost during a client/server transfer or unanticipated data may enter the transfer buffer due to the new transfer string variable being greater than or less than the system expectation of 240000000 characters. Although data length of 2400000000 characters may have been recorded before sanitation this value will have changed as the data object is made more concise and encrypted prior to transfer. |
| char \* datTransProc(const char[] newTransStr){  long dataTransLen = 240000000;  char transferBuff [dataTransLen];  for(int i; i < dataTransLen -1; i++)  if( newTransStr[i] == ‘\n’)  break;  transferBuff[i] = newTransStr[i];  return transferBuff;  } |

| **Compliant Code** |
| --- |
| The following function displays the proper variant of dataTransProc() where the data buffer is synchronized to the actual size of the new transfer string parameter variable that will prevent the loss of data and introduction of malicious script. |
| char \* datTransProc(const char[] newTransStr, long originalTransLen){  char transferBuff [sizeof(newTransStr)];  for(int i; i < sizeof(newTransStr) -1; i++)  if( newTransStr[i] == ‘\n’ || i == originalTransLen -1)  break;  transferBuff[i] = newTransStr[i];  return transferBuff;  } |

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

| **Principles(s):**  Keep It Simple relates to this standard since a function call to sizeof( ) will return the current size of the variable or type that present the most direct and simple way to determine the memory value of an object.  Sanitize Data Sent to Other Systems relates to this standard as the implementation of sizeof( ) during sanitation will ensure that pre- and post-encrypted data along with network packets are properly represented with the current sizes in memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ECLAIR | 1.2 | CC2.EXP09 | Can detect violations to this rule. Primary in malloc(), calloc() and realloc(). |
| Compass/ROSE | - | - | Can detect violations to this rule. Primary used to check size for malloc(), calloc() and realloc(). |
| Polyspace Bug Finder | R2024a | CERT C: Rec. EXP09-C | Checks for hard-coded object size used to manipulate memory |
| LDRA tool suite | 9.7.1 | 201 S | Partially implemented |

### 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 is a key feature that will verify the adherence to code standards for a variety of software structures. The phases of development where automation will be primary of interest are during the design, build, verify and test, monitor and detect, as well as the maintain and stabilize phases. The Pre-production phases are where developers will determine where automation and testing are not just essential but also most efficient for confirming the validity of data and testing fault behaviors. In Production phases will be the juncture where automation will provide added benefits through software self-testing, automated error-handling, and a variety of other procedure checks that will save developers time on code module revisions. The adherence to Coding Standards is verify through the fundamental features of automation.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL60-CPP | High | Likely | High | P3 | L3 |
| INT31-C | High | Probable | High | P6 | L2 |
| STR30-C | Low | Likely | Low | P9 | L2 |
| STR31-C | High | Likely | Medium | P18 | L1 |
| MEM35-C | High | Probable | High | P6 | L2 |
| MSC11-C | Low | Unlikely | High | P1 | L3 |
| ENV32-C | Medium | Likely | Medium | P12 | L1 |
| WIN02-C | High | Likely | High | P9 | L2 |
| EXP09-C | High | Unlikely | Medium | P6 | 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 at rest | This type of defense focuses on the encryption of stored data within a system. To apply this policy a system must encrypt new data before preserving it in a database and these protocols will ensure that information within the system is protected independently of access controls and other regulator functions. |
| Encryption in flight | This type of defense concentrates upon the encryption of information before transfer between systems, often across networks. To apply this policy developers must provide added encryption measures to data before transfer the data packet to an external system and the purpose of this added security measure is to prevent data hijacking, man-in-the-middle attacks, along with other hacking threats that common on public and private networks. |
| Encryption in use | This type of security measure refers to keeping data encrypted while in use by the local software system. To apply this policy a software package shall keep data encrypted even when the system is performing basic access and processing operations, the value of this protocol is that raw data cannot be easily dumped from the active software and used to manipulate a credential bypass or another form of brute-force function call. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | This principle refers to the requirement of a system to confirm the validity of a user’s identity before allowing the user to access any part of the software system. The principle of authentication is applied by providing a log-in procedure before allowing the user to access any privileged data or operations within the system and the benefit of authentication is that it is the simplest security measure utilized to prevent public-level and corrupt users from accessing sensitive information or operations. |
| Authorization | This principle refers to minimizing a user class’s level of entitlement allowing the users access to the minimal amount of data and operations required for their standard usage of the system to protect the system from exploitation. The principle of authorization is enforced by categorizing users into separate categories and defining the access/privilege requirements of each category of user, the advantage of authorization is that restricting users to the minimum entitlements will greatly increase the security and functionality of the software system. |
| Accounting | This principle refers to the system maintaining an awareness of who’s accessing data, how the users are interacting with the data and what data has been transferred to/from the system. The principle of accounting is applied through the integration of data logs produced by the software package that track events within the system’s operations along with error reporting procedures, the value of accounting is that when hacking attacks and fault codes occur within the software package there isn’t necessarily a need to recreate the events that led to the occurrence so long as the system logs or fault reports are detailed enough to explain how the event progressed to a system or security failure. |

**\***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 |  |
| 1.2 | 8/9/2024 | Project One | Christopher Anderson | [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 |