**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 6](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 14](#_Toc52464063)

[Coding Standard 6 17](#_Toc52464064)

[Coding Standard 7 19](#_Toc52464065)

[Coding Standard 8 21](#_Toc52464066)

[Coding Standard 9 23](#_Toc52464067)

[Coding Standard 10 26](#_Toc52464068)

[Defense-in-Depth Illustration 28](#_Toc52464069)

[Project One 28](#_Toc52464070)

[1. Revise the C/C++ Standards 28](#_Toc52464071)

[2. Risk Assessment 28](#_Toc52464072)

[3. Automated Detection 28](#_Toc52464073)

[4. Automation 28](#_Toc52464074)

[5. Summary of Risk Assessments 29](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 30](#_Toc52464076)

[7. Map the Principles 32](#_Toc52464077)

[Audit Controls and Management 33](#_Toc52464078)

[Enforcement 33](#_Toc52464079)

[Exceptions Process 33](#_Toc52464080)

[Distribution 34](#_Toc52464081)

[Policy Change Control 34](#_Toc52464082)

[Policy Version History 34](#_Toc52464083)

[Appendix A Lookups 34](#_Toc52464084)

[Approved C/C++ Language Acronyms 34](#_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 | Input from many data sources may have malicious intent and should be validated for both being in the correct format, and for being in the correct range of input values. Validating input should focus on server-side but may also be implemented client-side. Client-side validation is not as reliable or safe, as an attacker may bypass the client more easily. Using input validation can often make successfully attacking an application more difficult and eliminate software vulnerabilities. |
| 1. Heed Compiler Warnings | Compilers display errors which prevent the compiling of code, and warnings, which indicate there is a potential issue. Warnings can help identify hard to find bugs, or bugs before testing. These warnings should be eliminated to avoid potential issues and vulnerabilities. Compiler warnings help to find and eliminate bugs and issues as soon as possible and are often displayed by a compiler or other tool. |
| 1. Architect and Design for Security Policies | Security policies are used to protect confidentiality and integrity of data used, stored, and processed in a specific system. A software architecture represents the decisions related to the structure and behavior of a system. Creating an architecture and designing software to implement and enforce system security policies will help with protection of data involved in the system. |
| 1. Keep It Simple | Using complex code to create a system can increase the chance of errors, bugs, and vulnerabilities. Using only functions, features and code that are necessary can avoid the chance of having issues. Retaining simpler code can also help in the maintenance of the system; if a different developer cannot understand what has been written or why, they may have issues correcting the code to reduce or eliminate a vulnerability. |
| 1. Default Deny | Inbound and outbound traffic that is not permitted by a firewall should automatically be blocked. Blocking non-permitted traffic, including unnecessary services, can prevent malware from being spread to systems. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege restricts users, services, and applications to having a limited number of permissions. A specific amount of privilege is given; just enough to perform the task they require. Reducing the level of access can reduce security risks for users, as well as attackers. Most users do not require permission at an administrator level, so limiting the access can minimize the chance of having malware and rootkits installed. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data ensures that the data conforms to a system’s requirement, as well as security related requirements. Sanitization includes removing unwanted characters from input and can/should be done alongside input validation. Input should not be trusted as safe, and neither should client-side validation. Sanitizing this data can prevent exposure to security threats. GET requests, cookies and POST requests should all be sanitized. |
| 1. Practice Defense in Depth | Using a DoD approach, a series of defenses are layered to protect data and info. If an attack successfully gets through one mechanism of defense, a different layer steps up to attempt to stop the attack. There are intentional redundancies involved in this approach, and often, parts of the defenses will try to trick attackers. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance ensures the quality of software being released meets guidelines and requirements, including security requirements. QA should span along the entire SDLC (software development life cycle). Effective QA techniques help greatly reduce defects in software code. Techniques such as automated testing, allocating proper time limits for segments of development, prioritizing bugs, and releasing specific criteria carefully, and not too soon, can help a team of QA testers achieve the highest quality of a software system. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are formal coding practices used to prevent vulnerabilities, and  give developers guidance to deliver secure, high quality software systems. In a coding  standard, a few things that should be implemented include the principle of least privilege,  encrypted communications among the development team, protected passwords, and  other encryption, such as file and database. Also, specific language(s) should be  chosen at the creation of the standard, and which tools the team will be using, such as an  IDE (integrated development environment). Ensuring the team follows this standard  throughout development can reduce vulnerabilities, bugs, and many other issues. |

### 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** | **DCL07-C. Include the appropriate type information in function declarators.** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Function declarators should be declared with the appropriate type information. If the type information isn’t properly specified, the compiler can’t properly check function type information. Attempting to compile a program with a function declarator that doesn’t include the appropriate type information typically generates a warning but doesn’t prevent compilation. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example user the identifier-list form for parameter declarations: |
| **int** max(a, b)  **int** a, b;  {  **return** a > b ? a : b;  } |

| **Compliant Code** |
| --- |
| In this solution, “int” is the type specifier. “max(int a, int b)” is the function declarator, and the block within the curly braces is the function body: |
| **int** max(**int** a, **int** b) {  **return** a > b ? a : b;  } |

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

| **Principles(s):** 2 – Heed compiler warnings. Compiler warnings help find and eliminate bugs and issues. If the function declarator doesn’t include the correct type info, a warning will be generated which should be used to correct the type info error. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | Function-prototype implicit-function-declaration | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | LANG.FUNCS.PROT LANG.STRUCT.DECL.IMPT | Incomplete function prototype implicit type |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL07 | Fully implemented |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/c/RuleChecker) | 23.04 | Function-prototype implicit-function-declaration | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **INT04-C. Enforce limits on integer values originating from tainted sources.** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | All integer values originating from [tainted sources](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-taintedsource) should be evaluated to determine if they have identifiable upper and lower bounds. If so, these limits should be enforced by the interface. Restricting the input of excessively large or small integers helps prevent overflow, truncation, and other type range errors. Furthermore, it is easier to find and correct input problems than it is to trace internal errors back to faulty inputs. |

| **Noncompliant Code** |
| --- |
| In this example, length is the value of a user defined environment variable whose value is used to determine the size of a dynamically allocated array, table. In compliance with INT30-C. Ensure that unsigned integer operations do not wrap, the code prevents unsigned integer wrapping, but does not impose any upper bound on the size of the array, making it possible for the user to cause the program to use too much memory: |
| **char**\*\* create\_table(**void**) {  **const** **char**\* **const** lenstr = **getenv**("TABLE\_SIZE");  **const** **size\_t** length = lenstr ? **strtoul**(lenstr, NULL, 10) : 0;    **if** (length > SIZE\_MAX / **sizeof**(**char** \*))  **return** NULL;   /\* Indicate error to caller \*/    **const** **size\_t** table\_size = length \* **sizeof**(**char** \*);  **char**\*\* **const** table = (**char** \*\*)**malloc**(table\_size);    **if** (table == NULL)  **return** NULL;   /\* Indicate error to caller \*/      /\* Initialize table... \*/  **return** table;  } |

| **Compliant Code** |
| --- |
| This solution defines the acceptable range for “length” as [1, MAX\_TABLE\_LENGTH]. The length parameter is declared as size\_t, which is unsigned. The test for length == 0 ensures that a nonzero number of bytes is allocated. |
| **enum** { MAX\_TABLE\_LENGTH = 256 };    **char**\*\* create\_table(**void**) {  **const** **char**\* **const** lenstr = **getenv**("TABLE\_SIZE");  **const** **size\_t** length = lenstr ? **strtoul**(lenstr, NULL, 10) : 0;    **if** (length == 0 || length > MAX\_TABLE\_LENGTH)  **return** NULL;   /\* Indicate error to caller \*/    **const** **size\_t** table\_size = length \* **sizeof**(**char** \*);  **char**\*\* **const** table = (**char** \*\*)**malloc**(table\_size);    **if** (table == NULL)  **return** NULL;   /\* Indicate error to caller \*/      /\* Initialize table... \*/  **return** table;  } |

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

| **Principles(s):**   1. Validate input data – Tainted sources are locations where applications gain access to possibly tainted data. Tainted data refers to values that may be used for unauthorized or malicious operations to interact with a system. They can cause vulnerabilities and unauthorized access, SQL injections, etc. Validating the input data to check for upper and lower bounds on integers can prevent overflow and other range errors.   7. Sanitize data sent to other systems – Sanitizing data can remove unwanted characters from input and should be done while validating input. This can prevent errors, also. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probably | High | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.0 | NA | Supported by taint analysis |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | IO.TAINT.SIZE LANG.MEM.TBA IO.TAINT.ADDR IO.UT.HOST IO.UT.PORT  (general) | Tainted allocation size Tainted buffer access Tainted network address Untrusted Network Host Untrusted Network Port  CodeSonar will track the tainted value, along with any limits applied to it, and flag any problems caused by underconstraint. Warnings of a wide range of classes may be triggered, including tainted allocation size, buffer overrun, and division by zero |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | CERT\_C-INT04-a  CERT\_C-INT04-b CERT\_C-INT04-c | Protect against integer overflow/underflow from tainted data Avoid buffer read overflow from tainted data Avoid buffer write overflow from tainted data |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rec. INT04-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.int02c.html) | Checks for:   * Array access with tainted index * Loop bounded with tainted value * Memory allocation with tainted size * Tainted size of variable length array   Rec. partially supported. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **STR03-C. Do not inadvertently truncate a string** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Alternative functions that limit the number of bytes copies are often recommended to mitigate buffer overflow vulnerabilities. An example is:   * strncpy() instead of strcpy()   These functions truncate strings that exceed the specified limits. Also, some functions don’t guarantee that the resulting character sequence is null-terminated. Unintentional truncation results in a loss of data and in some cases, software vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The standard functions strncpy() and strncat() copy a specified number of character ‘n’ from a source string to a destination array. For strncpy(), if there is no null character in the first ‘n’ characters of the source array, the result will not be null-terminated and any remaining characters are truncated: |
| **char** \*string\_data;  **char** a[16];  /\* ... \*/  **strncpy**(a, string\_data, **sizeof**(a)); |

| **Compliant Code** |
| --- |
| The strcpy() and strncpy() functions can be used to copy a string and a null character to a destination buffer if there is enough space. The programmer must be careful to ensure the destination buffer is large enough to hold the string to be copied and the null byte to prevent errors, such as data truncation and buffer overflow. This solution requires that ‘string\_data’ is null-terminated (a null byte can be found within the bounds of the character array). Otherwise, ‘strlen()’ will stray into other objects before finding a null byte: |
| **char** \*string\_data = NULL;  **char** a[16];    /\* ... \*/    **if** (string\_data == NULL) {    /\* Handle null pointer error \*/  }  **else** **if** (**strlen**(string\_data) >= **sizeof**(a)) {    /\* Handle overlong string error \*/  }  **else** {  **strcpy**(a, string\_data);  } |

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

| **Principles(s):**  3. Architect and design for security policies – Creating an architecture and designing software to implement security policies can help prevent string truncation. Deciding how the structure and behavior of a system can help a development team understand what is expected and how to correctly code the software.  4. Keep it simple – String truncation is when an attempt to insert a string value into a specific location fails because the string exceeds the limits of the function. Using simple code and functions that insert string values correctly can prevent security vulnerabilities like truncation. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probably | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | MISC.MEM.NTERM | No Space For Null Terminator |
| [GCC](http://gcc.gnu.org/) | 8.1 | [-Wstringop-truncation](https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#index-Wstringop-truncation) | Detects string truncation by strncat and strncpy. |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/STR03-C.+Do+not+inadvertently+truncate+a+string#:~:text=Partially%20implemented-,Parasoft%20C/C%2B%2Btest,-2023.1) | 2023.1 | CERT\_C-STR03-a | Avoid overflow due to reading a not zero terminated string |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/STR03-C.+Do+not+inadvertently+truncate+a+string#:~:text=Polyspace%20Bug%20Finder) | R2023a | [CERT C: Rec. STR03-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str03c.html) | Checks for invalid use of standard library string routine (rec. partially supported) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **FIO30-C. Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Never call a formatted I/O function with a format string containing a tainted value. An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. The attacker can also execute arbitrary code with the permissions of the vulnerable process. Formatted output functions are particularly danger because many programmers do not know their capabilities. |

| **Noncompliant Code** |
| --- |
| The ‘incorrect\_password()’ function in this example is called during identification and authentication to display an error message if the specified user isn’t found or if the password is incorrect. The function accepts the users name as a string ‘user’. This is untrusted data that originates from an unauthenticated user. The function constructs an error message that is output to ‘stderr’.  The ‘incorrect\_password()’ function calculates the size of the message, allocates dynamic storage, and constructs the message in the allocated memory. The addition operations aren’t checked for integer overflow because the string referenced by ‘user’ is known to have a length of 256 or less. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| This solution fixes the problem by replacing the fprint() call with a call to fputs(), which outputs ‘msg’ directly to stderr without evaluating the contents. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

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

| **Principles(s):**   1. Validate input data – Input should not be trusted as it may contain a tainted value. Calling a function with possibly tainted values can cause crashes, data leak, etc. Validating this input can help with making sure the data is not tainted.   5. Default deny – Making sure that inbound and outbound traffic not permitted is blocked can prevent some issues with tainted user input that may be called through a function. Having a plan to block unwanted traffic before they can even enter malicious input is beneficial.  8. Practice defense in depth – having a multi-layered defense system can also prevent these tainted sources from entering user input. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | NA | Supported via stubbing/taint analysis |
| [GCC](https://wiki.sei.cmu.edu/confluence/display/c/GCC) | 4.3.5 | NA | Can detect violations of this rule when the -Wformat-security flag is used |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-FIO30-a** **CERT\_C-FIO30-b** **CERT\_C-FIO30-c** | Avoid calling functions printf/wprintf with only one argument other than string constant Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable Never use unfiltered data from an untrusted user as the format parameter |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 592 | Partially supported: reports non-literal format strings |

#### Coding Standard 5

| **Coding Standard** | **Label** | **MEM00-C. Allocate and free memory in the same module, at the same level of abstraction.** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Dynamic memory management is a common source of programming flaws that can lead to vulnerabilities. Poor memory management can lead to security issues such as heap buffer overflows, dangling pointers, and double free issues. Memory management involves allocating memory, reading, and writing to memory, and deallocating memory.  Allocating and freeing memory in different modules/levels of abstraction may make it difficult to determine when and if a block of memory has been freed, leading to defects such as memory leaks, double free vulnerabilities. To avoid these, memory should be allocated and freed at the same level of abstraction, and in the same code module. |

| **Noncompliant Code** |
| --- |
| This example shows a double-free vulnerability resulting from memory being allocated and freed at differing levels of abstraction. Memory for the ‘list’ array is allocated in the ‘process\_list()’ function, and the array is passed to the ‘verify\_size()’ function, which performs error checking on the size of the list. If the size of the list is below a minimum size, the memory allocated to the list is freed, and the function returns to the caller. The calling function then frees this same memory again, resulting in a double-free and possibly exploitable vulnerability.  The call to free memory in the ‘verify\_size()’ function takes place in a subroutine of the ‘process\_list()’ function, at a different level of abstraction from the allocation, resulting in a violation of this recommendation. The memory deallocation also occurs in error-handling code, which is frequently not as well tested as green paths through the code. |
| **enum** { MIN\_SIZE\_ALLOWED = 32 };    **int** verify\_size(**char** \*list, **size\_t** size) {  **if** (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/  **free**(list);  **return** -1;    }  **return** 0;  }    **void** process\_list(**size\_t** number) {  **char** \*list = (**char** \*)**malloc**(number);  **if** (list == NULL) {      /\* Handle allocation error \*/    }    **if** (verify\_size(list, number) == -1) {  **free**(list);  **return**;    }      /\* Continue processing list \*/    **free**(list);  } |

| **Compliant Code** |
| --- |
| To correct this problem, the error-handling code in ‘verify\_size()’ is modified so that it no longer frees ‘list’. This change ensures that ‘list’ is freed only once, at the same level of abstraction, in the ‘process\_list()’ function. |
| **enum** { MIN\_SIZE\_ALLOWED = 32 };    **int** verify\_size(**const** **char** \*list, **size\_t** size) {  **if** (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/  **return** -1;    }  **return** 0;  }    **void** process\_list(**size\_t** number) {  **char** \*list = (**char** \*)**malloc**(number);    **if** (list == NULL) {      /\* Handle allocation error \*/    }    **if** (verify\_size(list, number) == -1) {  **free**(list);  **return**;    }      /\* Continue processing list \*/    **free**(list);  } |

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

| **Principles(s):**  4. Keep it simple – Using complex code can increase the chance of vulnerabilities. Reducing this complex code and using more simple mannerisms can prevent these vulnerabilities, bugs, and errors from slipping through to the released product.  9. Use effective quality assurance techniques – Effective QA techniques can greatly reduce defects in code by using automated testing, prioritizing bugs, and carefully releasing specific criteria. A simple coding error of double freeing the same memory can lead to an exploitable vulnerability, so having effective QA techniques can prevent this from getting through. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probably | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | ALLOC.DF ALLOC.LEAK | Double free Leak |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-MEM00-a** **CERT\_C-MEM00-b** **CERT\_C-MEM00-c** **CERT\_C-MEM00-d** **CERT\_C-MEM00-e** | Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not use resources that have been freed Ensure resources are freed |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | 449, 2434 | Partially supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rec. MEM00-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.mem00c.html) | Checks for:   * Invalid free of pointer * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rec. partially covered. |

#### 

#### Coding Standard 6

| **Coding Standard** | **Label** | **DCL03-C. Use a static assertion to test the value of a constant expression.** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The runtine ‘assert()’ macro has some limitations, however, in that it incurs a runtime overhead and because it alls ‘abort()’. Consequently, the runtime ‘assert()’ macro is useful only for identifying incorrect assumptions and not for runtime error checking. Because of this, runtime assertions are usually unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This example uses the ‘assert()’ macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    **struct** timer {    unsigned **char** MODE;    unsigned **int** DATA;    unsigned **int** COUNT;  };    **int** func(**void**) {  **assert**(**sizeof**(**struct** timer) == **sizeof**(unsigned **char**) + **sizeof**(unsigned **int**) + **sizeof**(unsigned **int**));  } |

| **Compliant Code** |
| --- |
| In this solution, a preprocessor conditional statement is used for assertions involving only constant expressions. Using #error directives allows for clear diagnostic messages. Because this approach evaluated 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):**   1. Heed compiler warnings – Using static assertions can prevent issues that may happen by using ‘assert()’, which is only useful for identifying incorrect assumptions and not runtime error checking. Using static assertions allows for more clear diagnostic messages and is evaluated at runtime.   9. Use effective quality assurance techniques – Having effective QA techniques can eliminate uses of the assert() macro, using tools such as CodeSonar. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | Misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | (customization) | User can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL03 | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **ERR51-CPP. Handle all exceptions.** |
| --- | --- | --- |
| **Exceptions** | [STD-007-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 surrounding try blocks of the same thread.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception can’t be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| In this code example, both ‘f()’ and ‘main()’ fail to 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** |
| --- |
| In this solution, the main entry point handles all exceptions, which ensures the stack is unwound up to the ‘main()’ function and allows for graceful management of external resourcesL |
| **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):**  9. Use effective quality assurance techniques – Having the techniques to prevent, find, and eliminate exceptions that are NOT handled is key in preventing system crashes and normal operation of the software. If a malicious user finds an exception that is not properly handled, they may be able to cause some type of damage or gain unauthorized access.  10. Adopt a secure coding standard – Secure coding standards include exception handling. Adopting a secure standard throughout the development process can reduce vulnerabilities, exceptions not being handled (which can cause crashes and errors), etc. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probably | Medium | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 527 S | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **CTR53-CPP. Use valid iterator ranges** |
| --- | --- | --- |
| Containers | [STD-008-CPP] | When iterating over elements of a container, the iterators used must iterate over a valid range. An iterator range is a pair of iterators that refer to the first and past-the-end elements of the range. |

| **Noncompliant Code** |
| --- |
| In this example, the two iterators that delimit the range point into the same container, but the first iterator doesn’t precede the second. On each iteration if the internal loop, ‘std::for\_each()’ compares the first iterator with the second for equality. If they aren’t equal, it will continue to increment the first iterator. Incrementing the iterator representing the past-the-end element of the range results in undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this solution, the iterator values passed to ‘std::for\_each()’ are passed in the correct order: |
| #include <algorithm>  #include <iostream>  #include <vector>    **void** f(**const** std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

| **Principles(s):**   1. Heed compiler warnings – Using an incorrect range for iterating elements of a container will display errors during compilation. The range should not begin at the end and end at the beginning, for example.   9. Use effective quality assurance techniques – Using QA techniques can prevent incorrect iterator ranges and errors. There are automation tools that can be used, such as Polyspace Bug Finder, which can check for invalid iterator ranges. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probably | High | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | Overflow\_upon\_dereference | NA |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.4p0 | LANG.MEM.BO | Buffer Overrun |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CTR53-a** **CERT\_CPP-CTR53-b** | Do not use an iterator range that isn't really a range Do not compare iterators from different containers |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C++: CTR53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr53cpp.html) | Checks for invalid iterator range (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **MSC12-C. Detect and remove code that has no effect or is never executed.** |
| --- | --- | --- |
| Dead Code | [STD-009-CPP] | Code that has no effect or is never executed is typically the result of a coding error and can cause unexpected behavior. This code is usually optimized out of a program during compilation. To improve readability and ensure logic errors are resolved, this type of code should be identified, understood, and eliminated. |

| **Noncompliant Code** |
| --- |
| This example demonstrates how dead code can be introduced into a program. The second conditional statement, ‘if (s)’, will never evaluate to true because it requires that ‘s’ not be assigned ‘NULL’, and the only path where ‘s’ can be assigned a non-null value ends with a return statement. |
| **int** func(**int** condition) {  **char** \*s = NULL;  **if** (condition) {          s = (**char** \*)**malloc**(10);  **if** (s == NULL) {             /\* Handle Error \*/          }          /\* Process s \*/  **return** 0;      }      /\* Code that doesn't touch s \*/  **if** (s) {          /\* This code is unreachable \*/      }  **return** 0;  } |

| **Compliant Code** |
| --- |
| Remediation of dead code requires the programmer to determine why the code is never executed and then to resolve the situation appropriately. To correct the preceding noncompliant code the ‘return’ is removed from the body of the first conditional statement. |
| **int** func(**int** condition) {  **char** \*s = NULL;  **if** (condition) {          s = (**char** \*)**malloc**(10);  **if** (s == NULL) {             /\* Handle error \*/          }          /\* Process s \*/      }      /\* Code that doesn't touch s \*/  **if** (s) {          /\* This code is now reachable \*/      }  **return** 0;  } |

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

| **Principles(s):**  4. Keep it simple – Having code that is never executed can often be caused by complex code, or simply mistakenly adding things that will never work. Avoiding complex code can, at times, prevent these issues, making the program easier to read and reduce logic errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **dead-assignment** **dead-initializer** **expression-result-unused** **redundant-operation** **unreachable-code** **unreachable-code-after-jump** **unused-function** **statement-sideeffect** | Supported + partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **DIAG.UNEX.\* LANG.STRUCT.EBS LANG.STRUCT.RC MISC.NOEFFECT LANG.STRUCT.UC LANG.STRUCT.UA LANG.STRUCT.UULABEL** **LANG.STRUCT.UUMACRO** **LANG.STRUCT.UUPARAM** **LANG.STRUCT.UUTAG** **LANG.STRUCT.UUTYPE** | Code not exercised by analysis Empty branch statement checks Redundant condition Function call has no effect Unreachable code checks Useless assignment Unused Label Unused Macro Unused Parameter Unused Tag Unused Type |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **NO\_EFFECT**  **DEADCODE**    **UNREACHABLE** | Finds statements or expressions that do not accomplish anything or statements that perform an unintended action.  Can detect the specific instance where code can never be reached because of a logical contradiction or a dead "default" in switch statement  Can detect the instances where code block is unreachable because of the syntactic structure of the code |
| [GCC](https://wiki.sei.cmu.edu/confluence/display/c/GCC) | 3.0 | -Wunused-value  -Wunused-parameter | Options detect unused local variables, nonconstant static variables and unused function parameters, or unreachable code respectively. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **EXP08-C. Ensure pointer arithmetic is used correctly.** |
| --- | --- | --- |
| Pointer Arithmetic | [STD-010-CPP] | When performing pointer arithmetic, the size of the value to add to a pointer is automatically scaled to the size of the type of the pointed-to object. Failing to understand how pointer arithmetic works can lead to miscalculations that result in serious errors such as buffer overflow. |

| **Noncompliant Code** |
| --- |
| In this example, integer values returned by ‘parseint(getdata())’ are stored into an array of INTBUFSIZE elements of typ int called buf. If data is available for insertion into buf and buf\_ptr has not been incremented past buf + sizeof(buf), an integer value is stored at the address referenced by buf\_ptr. However, the sizeof operator returns the total number of bytes in buf, which is typically a multiple of the number of elements in buff. This value is scaled to the size of an integer and added to buf. As a result, the check to make sure integers aren’t written past the end of buf is incorrect, and a buffer overflow may occur. |
| **int** buf[INTBUFSIZE];  **int** \*buf\_ptr = buf;    **while** (havedata() && buf\_ptr < (buf + **sizeof**(buf))) {    \*buf\_ptr++ = parseint(getdata());  } |

| **Compliant Code** |
| --- |
| In this solution, the size of buf, INTBUFSIZE, is added to buf and used as an upper bound. The integer literal INTBUFSIZE is scald to the size of an integer, and the upper bound of buf is checked correctly: |
| **int** buf[INTBUFSIZE];  **int** \*buf\_ptr = buf;    **while** (havedata() && buf\_ptr < (buf + INTBUFSIZE)) {    \*buf\_ptr++ = parseint(getdata());  } |

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

| **Principles(s):**   1. Keep it simple – Having code that is too complicated can affect pointer arithmetic and can lead to miscalculations and errors, such as buffer overflow. Reducing the complexity of code allows developers to have a better understanding of the arithmetic and reduce errors such as these.   9. Use effective quality assurance techniques – Using techniques that can prevent incorrect pointer arithmetic, as well as automation tools such as Astree, CodeSonar, and Parasoft C/C++test, are all best practices. Testing for arithmetic errors with pointers is also a good idea with quality assurance. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probably | High | Medium | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astree](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | NA | Supported: Astrée reports potential runtime errors resulting from invalid pointer arithmetics. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.4p0 | **LANG.STRUCT.PARITH LANG.MEM.BO LANG.MEM.BU LANG.STRUCT.PBB LANG.STRUCT.PPE LANG.MEM.TBA LANG.MEM.TO LANG.MEM.TU LANG.STRUCT.CUP LANG.STRUCT.SUP** | Pointer arithmetic Buffer overrun Buffer underrun Pointer before beginning of object Pointer past end of object Tainted buffer access Type overrun Type underrun Comparison of Unrelated Pointers Subtraction of Unrelated Pointers |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **45 D** **53 D** **54 D** **438 S** **576 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-EXP08-a** **CERT\_C-EXP08-b** | Pointer arithmetic should not be used Avoid accessing arrays out of bounds |

### 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.

While it’s not exactly possible to automate every part of a DevOps or DevSecOps process, automation should be used whenever possible to minimize basic and repetitive tasks, which can save a team a great amount of time. From the assessing and planning phase all the way through production’s maintenance, automation is extremely helpful. Assertions and exceptions should always be handled, also, and automated testing can check to make sure they are included and handled correctly, in the design, build, verify and test phases.

To modify the current process to automate enforcement of many of the standards in our policy, there should be numerous automated tests in the ‘Verify and test’ phase. Checking for correct data type, data value, and string correctness should be done through various tests.

Other automation tools can be used for standards such as sql injection and memory protection, in te monitor and detect, and respond phases of this DevSecOps process. The monitoring and detection phase can include tests to detect dead code, which should be removed.

One automation tool, CodeSonar, was observed to be a tool included in all the standards in this policy. This product can identify programming bugs using a static analysis engine. It also includes automation features such as an API for custom integrations and support for extensions. There are numerous other tools that can be used throughout the DevSecOps process for automation, such as Pc-Lint, which can do a static exception analysis, etc. Picking the correct tools to use can be a process. Security tools should integrate into the development pipeline and enable the dev and security teams to work together. Products used should be easy to use and scan code.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Low | High | 2 |
| STD-002-CPP | High | Probable | High | Medium | 4 |
| STD-003-CPP | Medium | Probable | Medium | Medium | 3 |
| STD-004-CPP | High | Likely | Medium | High | 3 |
| STD-005-CPP | High | Probably | Medium | High | 4 |
| STD-006-CPP | Low | Unlikely | High | Low | 2 |
| STD-007-CPP | Low | Probably | Medium | Low | 1 |
| STD-008-CPP | High | Probably | High | Medium | 4 |
| STD-009-CPP | Low | Unlikely | Medium | Low | 2 |
| STD-010-CPP | High | Probably | High | Medium | 4 |

### 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 | Encryption in rest is the process of using algorithms to encode data on devices, such as hard drives, databases, etc. to protect it. This data can only be decrypted with a key and is often used to keep information confidential. Encryption will scramble data as ciphertext, which can only be decrypted with a decryption key. Even if an attacker obtains this data, they cannot use or decrypt it without the key.  Applying encryption in rest can be accomplished with encryption algorithms, such as the Advanced Encryption Standard (AES), which is one of the most common algorithms to encrypt data at rest. Data should always be encrypted at the storage layer.  This policy should be enforced to protect all data stored to protect from unauthorized attackers gaining access to it. Data breaches are a constant threat, so encrypting all data so that even if it IS obtained by a malicious user, it cannot be used or accessed without the encryption key. |
| Encryption at flight | Encryption at/in flight is when data moving over a network is encrypted. Using open internet connections, HTTP connections, public cloud implementations, cell phones, etc., all include data moving over a network, and should be encrypted.  Using this type of encryption requires two steps: encrypting the data that will be moving through a network and decrypting this data on an authorized user’s side. Each step includes a unique encryption key to authorize who can access the data.  Having a policy for encryption at flight is important to enforce whenever there is communication that includes sensitive data. Finances, passwords, credit card numbers, and any information that should be kept secure and private must be protected by in flight encryption to prevent unauthorized access. |
| Encryption in use | Encryption in use, also known as end-to-end encryption, takes place when data is being accessed and used. Some examples are databases, RAM data, and open files (even ones in the process of being created). In use encryption encrypts data through the entire data life cycle (in rest, at flight, in use).  To successfully use encryption in use, data should be encrypted at the source before it is even used. To decrypt the data, authentication mechanisms should be implemented to safely and correctly authorize users who can be trusted and who the data is intended for. This also requires that the user is only authorized to use the resources and data that they need, no more privileges should be granted. In use encryption adheres to the AES-256 standard.  This type of encryption should be implemented and used where and when users and applications require accessing data. It is important to enforce this policy because in use encryption keeps data encrypted during its lifespan, so data loss can be prevented during a breach. This type of encryption allows authenticated users to have access to specific data access and should analyze data requests in real time to block requests that are suspicious or unauthorized. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | In the Triple-A framework, authentication is the initial process, and involves identifying a user with credentials, such as a username and password, hardware tokens, biometrics, etc. Each user has a unique set of criteria to gain access, and the Triple-A server compares the given credentials to credentials in a database. The user is only given access to the network if the credentials they give are a match to a user in the database, otherwise authentication will fail.  Enforcing this policy and implementing authentication methods prevents unauthorized users and/or malicious users from accessing the network, database, and information. We do not want our software to let unauthorized users in, as they would be accessing other users’ information, which is a breach of security. When authentication fails, it means the user has no record of credentials on the Triple-A server, so they should not be granted access to perform any actions or access any data/resources. |
| Authorization | After authentication is successful, a user must be authorized to determine what activities, resources, and services they are permitted to use. Various authorization levels should exist, preventing lower-level users from higher up commands (such as admin permissions). There may be various types of authorization restrictions, such as frequency of logins, IP filtering, bandwidth traffic management, encryption, etc.  Having a policy of authorization can prevent unwanted changes to a database, access of files, creation/deletion of users, data leaks, password retrieval, along with other things. The hierarchy of authorization levels can also help by allowing a higher up level (such as an admin) to alter privileges of a user when necessary, so the user may be allowed access to things they were previously unable to access. |
| Accounting | In the Triple-A framework, accounting refers to the resources and files a specific user accesses, and which, if any, commands were issued and executed. After a user is authorized, their activity can be monitored and logged for admins to view. Accounting can even log statistics, such as session duration, data sent and received. This information can be used for authorization control, billing analysis and resource utilization activities.  It is important to ensure a policy of accounting is set into place. An audit can enable admins to login and view actions and the logged information that may have been accessed and which commands were executed. Admins can also see what, if any, changes have been made. If a user accesses a specific resource, an admin can look at what privileges that user has and see changes and if they were justified. Accounting can also allow admins to adjust the capacity of resources allowed based on activity trends.  Accounting also allowed for info-based decision making, like the capacity of resources allowed previously stated. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 07/16/2023 | Coding Standards – Milestone | David Varella | Prof. Mike Prasad |
| 3.0 | 08/02/2023 | Final Security Policy | David Varella | TBD |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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