**UGreen 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 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 12](#_Toc52464063)

[Coding Standard 6 14](#_Toc52464064)

[Coding Standard 7 16](#_Toc52464065)

[Coding Standard 8 18](#_Toc52464066)

[Coding Standard 9 20](#_Toc52464067)

[Coding Standard 10 22](#_Toc52464068)

[Defense-in-Depth Illustration 24](#_Toc52464069)

[Project One 24](#_Toc52464070)

[1. Revise the C/C++ Standards 24](#_Toc52464071)

[2. Risk Assessment 24](#_Toc52464072)

[3. Automated Detection 24](#_Toc52464073)

[4. Automation 24](#_Toc52464074)

[5. Summary of Risk Assessments 26](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 26](#_Toc52464076)

[7. Map the Principles 28](#_Toc52464077)

[Audit Controls and Management 29](#_Toc52464078)

[Enforcement 29](#_Toc52464079)

[Exceptions Process 29](#_Toc52464080)

[Distribution 30](#_Toc52464081)

[Policy Change Control 30](#_Toc52464082)

[Policy Version History 30](#_Toc52464083)

[Appendix A Lookups 30](#_Toc52464084)

[Approved C/C++ Language Acronyms 30](#_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** | Description |
| --- | --- |
| 1. ValidateInput Data | All data input to a computer system from an outside source is *untrusted* data and can assumed to be malicious. As the entry point for many system attacks, ALWAYS validate both syntax(correct structure) and semantics(correct values) of input data. |
| 1. Heed Compiler Warnings | Although compiler warnings may not prevent application build and execution, they always point to *potentially* unsafe or incorrect code structure. Compiler warnings must be investigated and addressed wherever possible. |
| 1. Architect and Design for Security Policies | Security as an extension or add-on is weak security. System security is integrated into our business processes and begins at application conception and design, through development, and to the rest of the life cycle. |
| 1. Keep It Simple | The larger a software system becomes, the greater the chance of introducing new bugs and vulnerabilities. This concept applies from the top-level down to individual modules and functions. Development work should prioritize clear, readable solutions. |
| 1. Default Deny | All traffic into the computer system is disallowed access by default. Only applications, ports, or specific networks that are necessary for the business operation will be granted access. |
| 1. Adhere to the Principle of Least Privilege | A single client session or other any other authentication must only be given the absolute minimum number of privileges needed to perform its intended function. This will prevent unintended access to other areas of the computer system. |
| 1. Sanitize Data Sent to Other Systems | Data must be passed *carefully* between modules. Even if the single module being worked on does not process a certain piece of data, issues can arise if sending poorly formatted data to a second system, such as a database. |
| 1. Practice Defense in Depth | Do not rely on a single layer of defense whenever it is possible. Any singular defensive method more than likely contains holes or blind spots. Layering will offset these weaknesses by combining two or more defensive efforts with vulnerabilities that do not overlap. |
| 1. Use Effective Quality Assurance Techniques | Never assume the software module you have built or modified, regardless of size, to be safe by default. Always use a combination of testing techniques including peer review, static testing, and unit and integration testing. |
| 1. Adopt a Secure Coding Standard | This policy is only an outline of the most important principles and techniques; there are many potentially vulnerable outcomes that can be created while developing software, especially using C/C++. OWASP and the SEI CERT C/C++ Coding Standards are standard inclusions for development at Green Pace. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Use the **size\_t** data type whenever dealing with loop counts, array indices, or lengths. |

| **Noncompliant Code** |
| --- |
| Integer data types are not guaranteed to be of any specific size in C/C++, and while they may in most cases be large enough for the purpose, may not *always* be. As an example, the below code may work incorrectly for sufficiently long input arrays and attempt to access oldArray at indeterminate indices, possibly locking operation altogether. |
| void copy(const char\* oldArray, std::string& newArray) {  int i;  for (i = 0; i < (sizeof(oldArray) / sizeof(char)); i++) {  newArray.push\_back(oldArray[i]);  } |

| **Compliant Code** |
| --- |
| size\_t is guaranteed to be able to store the maximum theoretical size of any type, including an array. It cannot contain a negative index value. It is also not implementation specific, resulting in better portability. |
| void copy(const char\* oldArray, std::string& newArray) {  **size\_t** i;  for (i = 0; i < (sizeof(oldArray) / sizeof(char)); i++) {  newArray.push\_back(oldArray[i]);  } |

| **Principles(s):** Architect and Design for Security / Validate Input Data  The system design should not be dependent on the exact hardware that the system runs on. The C++ standard does not define standard integer type sizes, so your code must be structured in a way to handle any variations in sizes. This coding standard further enforces the policy of checking data correctness. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | LANG.TYPE.BASIC | Static analysis tool for numerical types correctness |
| Splint | 3.1.1 |  | Lightweight static analysis tool which may run continuously in IDE |
| Axivion Suite | 7.2.0 | CERTC-INT01 | CI QA suite with static analysis, architecture analysis, and integration |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Always check that an operation involving unsigned integers will not exceed the data value limit of that type. |

| **Noncompliant Code** |
| --- |
| The behavior of a compiled C/C++ program is undefined when an operation exceeds the available size of an unsigned integer value. This undefined behavior can result in system vulnerability. The below code may overflow the maximum size of int when called with large values of a and/or b, resulting in undefined behavior. |
| int add(int a, int b) {  return a + b;  } |

| **Compliant Code** |
| --- |
| A precondition test can be used to check if the operation will overflow the used data type, such as the below for an addition operation. This will prevent undefined behavior from occurring. The <limits> library provides implementation specific data value maximums and minimums for reference. |
| #include <limits>  int add(int a, int b) {  if ((std::numeric\_limits<int>() – a) < b {  /\* handle numeric overflow \*/  }  else return a + b;  } |

| **Principles(s):** Validate Input Data / Defense in Depth  Even within a single program module, data we are receiving into another function should be checked for correctness. Adding a final layer of protection against unsigned integer overflow along with using correct typing elsewhere is a multilayered approach in line with practicing defense in depth. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Static analysis tool. Implements a number of standards to check and warn of integer overflow from allocation (conversion) and arithmetic. |
| Axivion Suite | 7.2.0 | CertC-INT30 | CI testing suite. Fully implements the  relevant CERT standard. |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Static code scanner for multiple languages. |
| Parasoft C/C++test | 2022.2 | CertC-INT30 (a/b/c) | CI solution which implements the relevant CERT standard. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CLG] | Ensure that the size of a string container is large enough for all characters *and* the null terminator. |

| **Noncompliant Code** |
| --- |
| C string literals are automatically null terminated with no further indication to the programmer. This makes each string literal larger than it appears (ignoring other issues that may be caused by wide character sets). The following code truncates a string literal, leading to possibly difficult to detect issues further downstream when this string may be passed to a function which requires the terminator to exist. |
| const char s[3] = “abc”;  // drops the null terminator |

| **Compliant Code** |
| --- |
| Allow the compiler to determine the appropriate container size when storing string literals. Alternatively, consider using C++ basic\_strings if possible. |
| const char s[] = “abc”;  or  std::string s = “abc”; |

| **Principles(s):** Keep It Simple / Heed Compiler Warnings  Wherever possible, it is best to let the compiler handle assignments. The compiler is extremely well tested and trusted to produce predictable and valid output. The resultant code is also simpler, which lends itself to being less error prone by virtue of having less places to go wrong. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | MISC.MEM.NTERM | Static analysis tool. The analyzer checks for this specific issue by default. |
| GCC | 8.1 | -Wstringop- truncation | Standard C/C++ compiler. Generates warnings for string truncation during copies using standard library functions. |
| Parasoft C/C++test | 2022.2 | CertC-STR03-a | CI solution which can analyze code against the specific CERT rule for inadvertently truncating strings. |
| Klockwork | 2022.4 | NNTS.MIGHT  NNTS.MUST | Multi-language static analysis which implements string truncation checks. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Never pass raw user input into a SQL statement without parameterization. |

| **Noncompliant Code** |
| --- |
| Raw user input designed to specify query data can be used to manipulate SQL commands directly. This can lead to a malicious user gaining access to unauthorized areas of a database. The below code asks the user for a name, but the user is able to instead get all names and IDs (example with Sqlite3). |
| #include “sqlite3.h”  std::string command = “SELECT NAME, ID FROM USERS WHERE NAME=”;  std::string query;  getline(std::cin, query) // user enters: “Bob OR 1=1”;  sqlite3\_exec(db, (command + query).c\_str(), …, …);  /\* executes SELECT NAME, ID FROM USERS WHERE NAME=BOB OR 1=1  and finds all names and ids \*/ |

| **Compliant Code** |
| --- |
| Use parameterization, which should be available in any modern SQL implementation. Do not rely on filters (blacklists / whitelists) as they may be circumvented more easily. Code is generalized for brevity. |
| #include “sqlite3.h”  **std::string command = “SELECT NAME, ID FROM USERS WHERE NAME=?”;**  **sqlite3\_stmt\* parameterizedCommand;**  std::string query;  getline(std::cin, query) // user enters: “Bob OR 1=1”;  **sqlite3\_prepare\_v2(db, command, parameterizedCommand);**  **sqlite3\_bind(parameterizedCommand, query);**  **sqlite3\_step(parameterizedCommand);**  /\* executes SELECT NAME, ID FROM USERS WHERE NAME= “BOB OR 1=1”  and finds nothing \*/ |

| **Principles(s):** Validate Input Data / Sanitize Data Sent to Other Systems  Input data from outside the system is *always* untrusted data. It is a requirement in all coding solutions to sanitize this type of data correctly. This concept extends to data leaving the system for elsewhere, such as a database in this case. Data must be handled responsibly and with great care at all levels. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2022.4 | SV.DATA.BOUND  SV.DATA.DB  SV.HTTP\_SPLIT  SV.PATH  SV.PATH.INJ  SV.SQL | Multi-language static analysis with general checks against multiple types of injection attacks. |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Static analysis tool. Implements checks for user input sent directly to SQL statements, among a few other conditions. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory. |

| **Noncompliant Code** |
| --- |
| Accessing memory locations that have already been freed is undefined behavior and can result in vulnerabilities. The following code will attempt to access a function in memory that has been technically deleted. The actual function code may or may not exist in actual memory anymore, and if so, may appear to work for a time. |
| class MyClass {  void myFunc();  }  void doSomething() {  MyClass myClass = new MyClass;  /\* work with the class \*/  delete myClass;  myClass->myFunc(); // results in undefined behavior  } |

| **Compliant Code** |
| --- |
| Consider allowing the compiler to determine when memory should be freed, either by declaring a single function bound object on stack (as below) or, if not possible, through use of a smart pointer. |
| class MyClass {  void myFunc();  }  void doSomething() {  MyClass myClass;  /\* work with the class \*/  myClass.myFunc();  } /\* memory is deallocated at the end of scope \*/  **or**  #include <memory>  void doSomething() {  auto myClass = std::make\_unique<MyClass>();  /\* work with the class \*/  myClass->myFunc();  } /\* memory is freed automatically when it is no longer needed \*/ |

| **Principles(s):** Least Privilege / Keep It Simple  Least privilege applies here as it is important to think about where exactly classes or other data types need to exist, and not declaring them outside of this scope. Doing so further allows us to keep things simple, such as implicitly declaring bound scope objects instead of assigning new memory locations manually. Doing so can prevent even severe vulnerabilities from presenting. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | ALLOC.UAF | Static analysis tool. Checks for attempted use of a memory location after the location has been freed. |
| Parasoft C/C++test | 2022.2 | CertC-MEM30-a | CI solution which implements the relevant standard from CERT, do not access freed memory. |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Multi-language analysis tool which checks for memory usage after free/delete has been called. Also checks for deallocation of previously freed locations. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to better qualify pre-release builds. The should be removed from production builds. |

| **Noncompliant Code** |
| --- |
| While building and debugging, it is normal to include some runtime checkpoints to verify conditions you expect to always be true in order to better detect bugs. However, it is important to use the best tools provided. The below example uses an if statement for one checkpoint, and no checkpoint for another, resulting in unneccessary obfuscation and difficulty removing. |
| int addValues(int\* intPtr1, int\* intPtr2) {  if (intPtr1 == NULL) {  /\* report error \*/  } else {  return \*intPtr1 + \*intPtr2;  }  int main {  int a = 5;  int b = 6;  addValues(&a, &b); // &a and &b are not expected to ever be null  } |

| **Compliant Code** |
| --- |
| Assert should be used to catch a possible condition during debugging, and then can be disabled for the final build once you are confident that the condition can no longer exist. Note that assert() should not be used for conditional tests that are *required* for proper and safe operation in the final build. |
| #include <cassert>  // assertions can be disabled at any time using #define NDEBUG  int addValues(int\* intPtr1, int\* intPtr2) {  assert(intPtr1 != NULL); // assert is a macro  assert(intPtr2 != NULL);  return \*intPtr1 + \*intPtr2;  }  int main {  int a = 5;  int b = 6;  addValues(&a, &b); // &a and &b are not expected to ever be null  } |

| **Principles(s):** Use Effective Quality Assurance Techniques  Although it is impractical to inserts value tests at every stage of code operation during development, adding checkpoints throughout will help assure that 1) less bugs are introduced into production and 2) otherwise more difficult to detect bugs can be found by performing assertions to detect inconsistencies in unexpected places. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | custom | Static analysis tool. A custom check can be created to warn the developer if assert() macros exist as a check before production. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Always handle exceptions before program termination. |

| **Noncompliant Code** |
| --- |
| The default C/C++ process if there is no exception handler is to terminate the problematic process completely, without a chance to clean up objects and any allocated resources. The below code may terminate abruptly, leaving some memory in an unknown state. |
| void thisFunctionThrows() noexcept(false);  int main() {  thisFunctionThrows();  }  /\* std::terminate() is called if no handlers match by the time the stack unwinds /\* |

| **Compliant Code** |
| --- |
| Catch exceptions, preferably specifically and as close as possible to the source. The below code has a catch all exception statement added and is able to properly deconstruct objects, close connections, etc. Note that the throwing function may be a function in the standard or third-party library. Please note that, although this is better than not catching all errors, absolute best practice is to catch errors by type, close to the source. |
| void thisFunctionThrows() noexcept(false);  int main() {  try {  thisFunctionThrows();  } catch (…) {  /\* perform error handling \*/  }  } |

| **Principles(s):** Design for Security / Adopt a Secure Coding Standard  Abrupt program termination is a leading vector for denial-of-service attacks, and the CERT standard specifically warns against allowing it for this reason. The module in development should consider security in all aspects of its design, including how it handles exceptions, and a proper exception pattern should be implemented from the start. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | LANG.STRUCT.UCTCH | Static analysis tool. Implements checks for exception structure including detection of unreachable catch statements. |
| Axivion Suite | 7.2.0 | CertC++-ERR51 | CI testing suite which conforms to CERT 51: handle all exceptions. |
| Parasoft C/C++test | 2022.2 | CertC++-ERR51 (a/b) | A CI testing solution which checks for all exceptions handled and for specific catches for each type that is thrown. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Arrays** | [STD-008-CLG] | Do not use out-of-bounds indices to access an array value. |

| **Noncompliant Code** |
| --- |
| Arbitrary data may be accessed or modified using a primitive array if proper index bounding is not enforced programatically. The below code attempts to access one further index than exists in an array, which may cause a system crash (off by one error). |
| void printArray(int\* array, size\_t size) {  size\_t i;  for (i = 0; i <= size, i++) {  printf(“%d”, &array[index]);  } /\* may fail silently \*/  } |

| **Compliant Code** |
| --- |
| Errors like this can be avoided completely with small changes to the program structure, as below. Alternatively, if using C++, use <vector> wherever possible. |
| void printElement(int\* array, size\_t size, size\_t index) {  if (index < size) {  printf(“%d”, &array[index]);  }  } |

| **Principles(s):** Validate Input Data / Defense in Depth  Functions that take arrays as input can receive varying sizes of data, therefore it is always important to validate that an array value exists before accessing it. There is no way to guarantee the value will exist prior to runtime. Even when using arrays of known lengths that do not change, this principal is important as a layer of defense in depth. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU  LANG.STRUCT.PARITH  LANG.STRUCT.PBB  LANG.STRUCT.PPE  BADFUNC.BO.\* | Static analysis tool with a collection of checks for arrays and indices. |
| Cppcheck | 1.66 | arrayIndexOutOfBounds, outOfBounds, negativeIndex, arrayIndexThenCheck, arrayIndexOutOfBoundsCond, possibleBufferAccessOutOfBounds | Static analysis tool that implements out of bounds checking of arrays. |
| Parasoft C/C++test | 2022.2 | CERT\_C-ARR30-a | CI testing tool. Specifically implements CERT standard ARR30: avoid accessing arrays out of bounds. |
| Parasoft Insure++ |  |  | Runtime detection of memory errors related to this standard along with others. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input** | [STD-009-CLG] | Always ensure proper bounds checking when copying from input. |

| **Noncompliant Code** |
| --- |
| Strings of characters from an input stream are of indeterminate length. If the contents are read directly into a container without concern for the size of the data, buffer overflow and occur resulting in the possibility of data writes to other memory locations in the compiled program. The below program can overflow if the user enters too many characters. |
| int main() {  char[12] inputBuffer;  gets(inputBuffer);  } |

| **Compliant Code** |
| --- |
| Dynamically allocate buffer space to prevent potential overflows. In C, this can be done using the getline() command as shown. If using C++, basic\_string can be used to simply allocate a dynamic amount of memory from the input stream. |
| int main() {  char\* inputBuffer = NULL;  size\_t bufferLength;  if (getline(&inputBuffer, &bufferLength, stdin) < 0) {  /\* handle any errors \*/  }  /\* the inputBuffer must be freed after use as it was allocated on heap \*/  } |

| **Principles(s):** Validate Input Data  As with all input data, user entered arrays of characters, especially from the standard input stream, can be of any length. It is required to check the length of these inputs against the size of the storage space allotted for it. There are multiple methods to implement this and no excuse not to handle this properly. Failure to do so can result in buffer overflow vulnerabilities, a leading cause of security breaches. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Fortify SCA | 5.0 | CERT C Rule Pack | Static code analyzer that can detect improper memory copies, with the exception of those involving the use of *sizeof()* |
| Coverity | 2017.07 | SIZECHECK | Multi-language analysis tool, with partial coverage of this standard. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Output** | [STD-010-CLG] | Do not use deprecated functions. |

| **Noncompliant Code** |
| --- |
| The common format function printf() may allow arbitrary writes to memory if fed malformed or directly malicious input from a user. Further, it may output sensitive data to the user in the same situation. The below code will intepret input as command by default and respond with unexpected output. |
| int main() {  char\* input = NULL;  size\_t len;  getline(&input, &len, stdin);  printf(input);  } /\* vulnerable to interpreting input as commands and leaking data to console \*/ |

| **Compliant Code** |
| --- |
| Always format output, specifying how the format function should treat the data input. In the below, the printf() function is told to treat input as a string only. |
| int main() {  char\* input = NULL;  size\_t len;  getline(&input, &len, stdin);  printf(“%s”, input);  } |

| **Principles(s):** Design for Security / Adopt a Secure Coding Standard  The printf() function with a single argument (among others) is deprecated and should not be used for security concerns. Using it in this way is a simple violation of secure coding standards, which are put in place for the protection of our software as well as others. It also illustrates how, in some cases, simple design choices can increase system security without much added effort. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.2p0 | BADFUNC.\*  custom | Static analysis tool. Can be configured to check for a variety of deprecated/obsolete functions. Custom checks can be added for specific functions. |
| Parasoft C/C++test | 2022.2 | CERT\_C-MSC24 (a/b/c/d) | CI testing tool. A variety of checks are implemented including checking for usage of unsafe string functions. |
| Axivion Suite | 7.2.0 | CertC-MSC24 | CI suite that fully implements CERT C MSC24: Do not use deprecated functions. |

### 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.

The main benefit of automating the enforcement of security policies is consistency. These automized systems can perform repeated checks faster and more accurately than a human. However, the use of these tools does not completely remove developer agency from the process.

Security in the development process should start with threat modeling with input from development and analysis teams. This provides the foundation for requiring security functionality throughout the rest of the development lifecycle. Next, during building stages, both IDE plugins and static analysis tools can be used to perform security checks. IDE plugins especially can enforce certain checks in the background with no extra burden on development and can prevent bugs and vulnerabilities before being written.

Automation of runtime testing using unit testing and integration testing tools/frameworks are also important. Randomized condition schemes such as Chaos Monkey can introduce further tests that the development team did not expect or did not think of, increasing the resilience of the application towards production.

Finally, automation continues through the lifetime of the software in use. Logging and monitoring of network traffic, and software authentication and signing techniques such as checksums can be used to verify operational security continuously and automatically, along with alerting the system maintainers to new or potentially unsafe use cases. Single sign-on systems can be used to authenticate developers and maintainers across all platform tools simply and automatically.

All of these automation techniques work together to create multiple layers of defensive strategies that results in a strong total security system. This lends itself to the concept of defense in depth.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-002-CPP | High | Likely | High | Medium | 2 |
| STD-003-CLG | Medium | Probable | Medium | Medium | 2 |
| STD-004-CPP | High | Probable | Medium | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 4 |
| STD-006-CPP | Low | Unlikely | High | Low | 1 |
| STD-007-CPP | Low | Probable | Medium | Low | 1 |
| STD-008-CLG | High | Likely | High | Medium | 2 |
| STD-009-CLG | High | Probable | High | Medium | 2 |
| STD-010-CLG | High | Probable | Medium | High | 3 |

### 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 | Data at rest is data that is not in use and that is stored in some location, for example in a database. It cannot be assumed that just because data is stored in a location that is not easily accessible, or even password protected, is therefore safe.  In line with the principle of defense in depth, all sensitive database information should be encrypted. Even if the data is breached, the attacker will not be able to read any of the information. With a strong encryption scheme, it may not be feasible for the attacker to access the data at all.  It is important to note that encryption is useless if the decryption keys are not stored properly. A reputable key management system should be used and stored in a separate location. Please do not create your own encryption scheme, instead use the standard solution for the framework you are using and pick a strong method. |
| Encryption in flight | Data in flight is data that has left one system and is in transit to another. Since data will typically travel over the widely accessible public tool known as the internet, data in flight is especially susceptible to attack from a malicious third party.  Here, encryption is a strong line of defense. Methods of encryption have been developed and are used with great success to keep private information secure over internet channels. Most commonly, the SSH/TLS protocol is used which implements RSA handshaking and strong cryptographic hashing algorithms that are constantly being improved.  Simply put, encryption of even remotely sensitive data over the internet is a requirement, and at this point there is little to no reduction in performance to do so. |
| Encryption in use | Often, data will neither be in transit or in rest, but will be actively in use by a program or system. In this case, the principle of least privilege should apply. Data should not be decrypted until absolutely necessary (or perhaps, never at all, such as in the case of password hashing).  When data does need to be used in plain format, other methods such as strong access control with multi-factor authentication should be used. It is necessary to remember that attackers are resourceful and, in some cases, understand a computer system better than the developer. They may gain access to a system in surprising locations, so no area of a computer system should be assumed safe enough to drop all data protections. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the ‘who are you’ of the Triple-A Framework. This principle applies to every user of the system, both from the public side and from within the business itself. Just because system access is requested from a known location, such as inside the business unit or a known IP address, does *not* make it safe.  The type of authentication necessary for a public user depends on what a typical user accesses; if they store any personal data or perform any sensitive actions with an area of the system, authentication is necessary and multi-factor is preferred.  Authentication of especially sensitive roles such as admins must involve multiple factors, such as the common username/password scheme used alongside a cellphone number, keeping in mind that passwords are often guessed or stolen. Sufficiently strong passwords that can withstand brute force attacks must be enforced along with incorrect attempt limits. |
| Authorization | Authorization refers to what any authenticated role is allowed to do. The principle of least privilege strongly applies here and is the main consideration for the policy in this area. Role access must be limited to what the role *needs* to do and *only* what they need to do. As an example, we should not assume that every system maintainer will need access to every single admin role to do their job.  By default, every role should be denied access to any area. After this, access is opened up as needed for that role. This ensures that no roles are accidentally given too much access. It also helps ensure that if an attacker were artificially elevate their privileges to another level, they would not necessarily be in a position to cause system harm. |
| Accounting | Accounting is the ‘what did you do?’ of the Triple-A Framework. A user may be properly authenticated and authorized for legitimate access to certain areas of the system. However, in line with defense in depth, it is important to maintain a simple record of what was done in the system and what files were accessed.  This is not for spying purposes. This is for security automation and detection of irregularities which can thwart successful security breaches before any sensitive information is breached. There are documented instances of system attack which went undetected for years simply for the lack of systems like this in place.  Suspicious actions could be, for example, a public user making changes to a database in unexpected location, or an admin assigning new roles to unknown parties, among others. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 01/22/2023 | Core security principles, standards 1-10 added | Craig O'Loughlin |  |
| 1.2 | 02/12/2023 | Completed Security Policy Document | Craig O'Loughlin |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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