**Green Pace Developer: Security Policy Guide Template**



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

# Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

# Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

# Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All input from untrusted sources should be validated. Validation can eliminate a hacker’s access to source code and eliminate most software vulnerabilities. Sources of untrusted input can be, but are not limited to user-controlled files, user input, command line arguments, network traffic/interfaces, and environment variables (Software Engineering Institute, 2006). |
| 1. Heed Compiler Warnings | Code should be compiled using the highest available warning level. Static and dynamic analysis tools should be used to detect security flaws. All code warnings should be eliminated by fixing and updating the code and recompiling (Software Engineering Institute, 2006). |
| 1. Architect and Design for Security Policies | Applications and programs should be designed according to the security and privilege principles as defined in the Green Pace security policies (Software Engineering Institute, 2006). |
| 1. Keep It Simple | Code (files, classes, methods) should be kept as small and as simple as possible. Overly complicated code can be more difficult to read and maintain and can increase the chance of errors within a program (Software Engineering Institute, 2006). |
| 1. Default Deny | By default, code should only ask for permissions it needs and not all permissions (admin) with exclusions. When permissions are needed, the minimum permissions needed should be added and requested (Software Engineering Institute, 2006). |
| 1. Adhere to the Principle of Least Privilege | Execute every process with the lowest set of privileges necessary to perform the task. Processes requiring higher level permissions should ask permission and use the higher permissions levels for the least amount of time necessary to perform the task. Maintaining higher permissions levels can allow an attacker to execute malicious code with higher privileges as well (Software Engineering Institute, 2006). |
| 1. Sanitize Data Sent to Other Systems | Sanitize any and all data sent to other systems. The subsystems being called are unaware of any context in which they are being called, whereas your system is aware of the context surrounding the call and the input. Sanitizing data sent to subsystems can prevent attackers from using injection attacks (Software Engineering Institute, 2006). |
| 1. Practice Defense in Depth | Combine multiple layers of security to protect your software and processes. Should one layer fail, it does not inherently compromise the system as there are multiple layers of defense. Secure runtimes combined with these secure programming principles can be combined into a layered defense against vulnerability exploitation (Software Engineering Institute, 2006). |
| 1. Use Effective Quality Assurance Techniques | Quality Assurance is the last step in the development process and one of the most important ones. Code review, usability testing, penetration testing, and fuzz testing should be implemented processes. These processes can protect systems by identifying vulnerabilities before they are released (Software Engineering Institute, 2006). |
| 1. Adopt a Secure Coding Standard | Create and adopt a strict coding standard to which all engineers must adhere. Coding standards should be defined on a per language basis and geared towards the targeted deployment environments (Software Engineering Institute, 2006). |

## 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 | Choose appropriate types of integers  Choose the most appropriate type of integer needed for a variable. Integer types provide a “finite subset of the mathematical set of integers” based on the specific type of integer. Using an incorrectly sized integer variable can cause a buffer overflow or numerical overflow (wraparound) and lead to exploitable vulnerabilities or undefined behavior (Seacord, 2013, pp 87-113). |

| **Noncompliant Code** |
| --- |
| The non-compliant code example below sets a variable my\_num equal to the maximum value of an integer which is 2147483647. Adding to this value, in this example by one, will result in a wrap around. The value of my\_num would now equal -2147483648. |
| int my\_num = INT\_MAX;  my\_num = my\_num + 1; |

| **Compliant Code** |
| --- |
| By changing the type of integer from implied signed to explicitly declaring an unsigned integer effectively doubles the positive numbers allowed by an integer. INT\_MAX still has a value of 2147483647, but by declaring the integer type to an unsigned int, the max of the variable is now 4294967295 (Microsoft, 2019). The result of the operation would update the variable my\_num equal to 2147483648. |
| unsigned int my\_num = INT\_MAX;  my\_num = my\_num + 1; |

|  |
| --- |
| **Principles(s):**  **8:**  This principle maps to this standard because using proper variable types is just one layer in the defense of an application and its security. Often times, the type of variable is the first line of defense especially if it’s used with user input data.  **10:** "Coding standards should be defined on a per language basis” – knowing the features of the language being used and the intended use of the variable types can add to the security of an application. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | LANG.MEM.BO | Guarantee that library functions do not overflow |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Ensure operations on signed integers do not wrap  Operations on signed integers should not wrap. As seen in standard STD-001-CPP, the data type can be updated to an unsigned integer which will help prevent wrap in some cases but not all. Prior to executing a mathematical operation on a signed integer, code should check for a potential overflow and act accordingly (Software Engineering Institute, 2006). |

| **Noncompliant Code** |
| --- |
| In the non-compliant example below, when the signed integers si\_a and si\_b are added together, if the result of that operation is greater than the value of INT\_MAX, the operation would result in an overflow or wraparound. |
| void func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** sum = si\_a + si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| The example code below first to see if an overflow could occur. If an overflow would happen from the addition operation, the error is handled. If no overflow would occur, then the two values are added together as expected. |
| #include <limits.h>    void f(signed int si\_a, signed int si\_b) {    signed int sum;    if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||        ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {      /\* Handle error \*/    } else {      sum = si\_a + si\_b;    }    /\* ... \*/  } |

|  |
| --- |
| **Principles(s):**  **8:** Integer wrapping can cause unexpected behavior. Integer wrapping when calculating the length of a buffer can cause a buffer overflow which can lead to an attacker being able to run malicious code (Nidecki, 2020).  **10:** "Coding standards should be defined on a per language basis” – The language used in the application should be understood. Part of this is knowing the available data types and their limitations. In the example above, the limitations of a signed int, and what happens when the max integer limit is passed, should be known and accounted for. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | LANG.MEM.BO | Guarantee that library functions do not overflow |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure that strings have enough storage space for the character data and a null terminator.  Data inserted or copied into an insufficiently sized buffer may result in a buffer overflow. Strings should be either truncated or measures should be in place to ensure sufficient space is available prior to assigning a value (Software Engineering Institute, 2008). |

| **Noncompliant Code** |
| --- |
| Input from the user has no size/limit bounds and therefore if it exceeds the size of the user\_input character array, a buffer overflow can occur. |
| char user\_input[20];  std::cin >> user\_input; |

| **Compliant Code** |
| --- |
| Rather than reading and immediately assigning user input into the character string user\_input, the cin function getline is called which reads the user’s input with a maximum size set. The value read by cin is checked for errors and failures and if a string larger than the size of the character array was input, an error message is displayed and the program exits. |
| char user\_input[20];  std::cin.getline(user\_input, sizeof(user\_input) + 1);  if (std::cin.fail() || std::cin.eof() || std::cin.bad()) {  std::cin.clear();  std::cout << "Buffer overflow detected" << std::endl;  return 1;  } |

|  |
| --- |
| **Principles(s):**  **1:** Input data (user or otherwise) can sometimes be longer than the buffer length declared which could cause a buffer overflow and a potential security risk.  **2:** Static analysis tools can find and report this issue thereby warning the developer of a potential issue.  **8:** Proper data type declarations and usage is the first line of defense in an application. The size and limitations of the declared data type should be known and accounted for to preserve security.  **9:** Automated testing such as unit and fuzz testing should be utilized to guard against bad input data.  **10:** "Coding standards should be defined on a per language basis” – The limitations and requirements of strings and character arrays should be known in the language used – C++ in the examples above. A character array with a predefined size will not grow dynamically and is subject to buffer overflow. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | LANG.MEM.BO, LANG.MEM.TO, MISC.MEM.NTERM |  |
| LDRA | 9.7.1 | 66 X, 70 X, 71 X | Rules catch stack-based buffer overflow (LDRA, 2019). |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Prevent SQL injection  SQL queries can be maliciously altered, if not sanitized and validated, resulting in information leaks and/or data modification. Stored procedures are the primary defense against SQL injection as no raw SQL string is being sent across the network. Parameterized queries should be used when an application is writing raw queries itself (Software Engineering Institute, 2009). |

| **Noncompliant Code** |
| --- |
| The example below is non-compliant because the name parameter’s value is not sanitized. The raw query is passed to the database. This could result in the value for name being sent as Fred’ OR 1=1 which would always evaluate to true and therefore return all records from the database instead of just people named Fred. |
| std::string name;  std::cin >> name;  std::string sql = “SELECT id, name, password FROM users WHERE name='” + name + “'”;  sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message); |

| **Compliant Code** |
| --- |
| Use prepared statements with SQL databases. This parameterizes queries and binds values to parameters. This prevents SQL injection as the input is essentially escaped an not inserted directly into the query string. The example below (jrok, 2020) creates a Sqlite prepared statement, prepares it, binds the text value of the first parameter, executes the statement, and finally destructs the prepared statement. |
| std::string name;  std::cin >> name;  std::string sql = “SELECT id, name, password FROM users WHERE name=?”;  sqlite3\_stmt\* stmt;  sqlite3\_prepare\_v2(db, sql.c\_str(), sql.length(), &stmt, nullptr);  sqlite3\_bind\_text(stmt, 1, name.c\_str(), name.length(), SQLITE\_STATIC);  sqlite3\_step(stmt);  sqlite3\_finalize(stmt); |

|  |
| --- |
| **Principles(s):**  **1:** Maps to this standard because input data in an application might get sent as parameters in a SQL query. Nefarious input data could include malicious code meant to attack a database.  **4:** SQL in an application should be simple. Overly complicated queries can become difficult to maintain and a potential security hole might be overlooked for more complicated queries. Complicated queries should be broken up or stored in the database as stored procedures.  **6:** Use only the appropriate privileges necessary to execute a query. Ability to drop tables, add users, change permissions, etc, are likely not needed by application SQL and therefore should use a dedicated user with the minimum database privileges needed.  **7:** Data sent from an application to a database (external system) should be sanitized prior to sending it. This will protect the external system from possible attack from within an application.  **8:** Preventing SQL injection is a single layer in defense for an application. Input data should be validated and sanitized prior to accepting it in the application, but once it’s stored in application memory, it should again be sanitized prior to sending to the database. That example shows two layers for added security.  **9:** Automated tests can run various known SQL injection attacks on all input variables in an attempt to alter the database. The database state can be checked prior to and after each test to ensure proper protection against SQL injection for a given attack vector.  **10:** "Coding standards should be defined on a per language basis” – The vectors of attack for SQL injection should be known as well as the features of the Structured Query Language itself so as to be aware of potential risks or vulnerabilities in query statements. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Checkmarx](https://www.checkmarx.com/) | 2020 | SAST | Supports many scripting languages and compiled languages, CPP included, and checks for SQL injection |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access memory that has already been freed  Avoid creating or using dangling pointers. Dangling pointers are pointers to memory that has been freed or deallocated by a memory management function. Dangling pointers can lead to unexpected behavior and result in vulnerabilities that can be exploited by a hacker (Software Engineering Institute, 2008) |

| **Noncompliant Code** |
| --- |
| In this example, a new instance of S is created as a pointer. Some code is executed, then S is deleted. Sometime after that, the code attempts to call a method on the pointer, which since it has been dereferenced and the memory deallocated, is now a dangling pointer. This can result in a vulnerability in which code can be exploited to access memory and run code with the permissions of the parent process (our process). |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In the example below of compliant code, the memory for the pointer s is not deallocated until the variable is no longer needed (at the end of the calling function). |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

|  |
| --- |
| **Principles(s):**  **2:** Applies to this principle because static analysis tools and compilers should be able to alert the developer of this behavior.  **10:** Depending on the language used, the developer should understand the correct usage of pointers and the effects of each use within said language. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use | Proves the absence of runtime errors (AbsInt, n.d.) |
| Clang | 3.9 | clang-analyzer-alpha.security.ArrayBoundV2, clang-analyzer-cplusplus.NewDelete |  |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use assertions liberally throughout the code  Assert statements can catch bugs early and make debugging easier and quicker. Check every precondition and assumption by including the <cassert> header and using the assert() method (LLVM, 2003).  Assertions should include a descriptive error message that is printed if the assert statement fails. This will help with debugging in being able to identify where the build failed and under what conditions (LLVM, 2003). |

| **Noncompliant Code** |
| --- |
| In this example of non-compliant code (“Assertions in C/C++”, 2020), the programmer sets an integer variable to the value of 7. For the rest of the code, the programmer is expecting the variable x to be equal to 7. Suppose a block of code accidentally sets x equal to 9. Later in the code when the programmer expects x to equal 7, an error will occur. |
| #include <stdio.h>  #include <assert.h>    int main()  {      int x = 7;        /\*  Some big code in between and let's say x         is accidentally changed to 9  \*/      x = 9;      /\* Rest of the code \*/      return 0;  } |

| **Compliant Code** |
| --- |
| By inserting an assertion that says we expect x to equal 7 prior to actually needing x to equal 7, the compiler can warn us that something has changed the value of x to 9 earlier in the code and that this will throw an error (“Assertions in C/C++”, 2020). |
| #include <stdio.h>  #include <assert.h>    int main()  {      int x = 7;        /\*  Some big code in between and let's say x         is accidentally changed to 9  \*/      x = 9;        // Programmer assumes x to be 7 in rest of the code      assert(x == 7, ‘Expected x to equal 7’);        /\* Rest of the code \*/        return 0;  } |

|  |
| --- |
| **Principles(s):**  **2:** Compilers and static analysis tools can catch failed assertions prior to compilation or release.  **9:** Testing with various types of data can ensure the assertions trigger when they should. Code review will also help in catching potentials for failure but also places where more assertions could be utilized. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N/A | N/A | N/A | Not applicable because tools can’t tell you to use more assert statements |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions  When an exception is thrown, control is given to the nearest handler. If no handler can be found within the immediate try block, the exception bubbles up in the call stack to the next nearest handler. If ultimately no handler is found, the program will abruptly terminated by a call to the std::abort() function via the default program exception handler. This abrupt termination means the stack is not rewound, destructors are not called, and objects and resources can be left in an indeterminate state causing unknown problems (Software Engineering Institute, 2015).  Catching all errors using the catch(…) must not be used without reason or proper error handling within the catch statement. This is otherwise considered “error hiding” and can cause unforeseen problems as the source of the error is not fixed or known. |

| **Noncompliant Code** |
| --- |
| The function throwing\_func throws an exception. When f() is called from the main function, an exception is thrown and neither f nor main catch the exception. This will result in an abrupt termination of the program. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| The compliant code below is basically the same as the non-compliant code above. A call to f() will still call throwing\_func which will still throw an exception. This time however, the main function wraps the call to f() within a try/catch block. When the exception is thrown and bubbled up through f into main, main handles the exception and the program can continue running normally. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    **int** main() {    try {      f();    } catch (...) {      // Handle error    }  } |

|  |
| --- |
| **Principles(s):**  **2:** Static analysis tools can alert a developer when an exception is thrown but not caught. This allows the developer to act accordingly and appropriately.  **8:** Uncaught exceptions will cause the abrupt termination of the program which could prevent application memory from being cleaned up. The protection of the application memory is imperative and properly handling exceptions is but one layer in the DiD.  **9:** Code reviews and automated tests such as unit tests can ensure proper exceptions are thrown and all exceptions are handled appropriately. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | early-catch-all, main-function-catch-all | Proves the absence of runtime errors (AbsInt, n.d.) |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-ERR51 | Covers 100% of CERT C++ coding guidelines (Axivion, 2021) |
| Parasoft | 2020.2 | CERT\_CPP-ERR51-a, CERT\_CPP-ERR51-b |  |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Expressions** | STD-008-CPP | Do not read uninitialized variables  Variables of type T must be initialized before being read. Attempting to read an uninitialized variable can result in a coding error and vulnerability. This happens because the variable can be removed during compilation as part of the compiler’s optimizations (Software Engineering Institute, 2008) |

| **Noncompliant Code** |
| --- |
| The variable i is declared but uninitialized. The variable is read and attempted to be displayed in the standard output. The uninitialized variable could be removed as part of the compiler’s compilation and therefore cause an error or possible vulnerability. |
| #include <iostream>    void f() {  **int** i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| The variable i has been initialized with the value of 0. Now when it is read, it will be properly output to the standard output and the variable won’t be removed by the compiler. |
| #include <iostream>    void f() {  **int** i = 0;    std::cout << i;  } |

|  |
| --- |
| **Principles(s):**  **2:** Compilers may remove an unused variable, but static analysis tools can alert the developer to their existence, thus preventing unintended errors.  **9:** Code review and usability testing can uncover areas where a compiler-removed unused variable caused an issue. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | uninitialized-read | Proves the absence of runtime errors (AbsInt, n.d.) |
| Clang | 3.9 | -Wuninitialized, clang-analyzer-core.UndefinedBinaryOperatorResult |  |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-009-CPP | Do not store an already owned pointer into an unrelated smart pointer  Smart pointers enclose a regular pointer and automatically manage its memory allocation via internal constructor and destructor methods in the smart pointer classes. When a smart pointer is constructed with a pointer, it owns said pointer. The original pointer should not be shared with another smart pointer as this can cause vulnerabilities and code issues (Software Engineering Institute, 2015). |

| **Noncompliant Code** |
| --- |
| Storing a pointer in multiple unrelated smart pointers can cause a double-free vulnerability where when one pointer is destructed which will deallocate the memory for the pointer. In the non-compliant example below, i is passed to the constructors of p1 and p2. If p2 is destroyed it deletes the pointer i. Then when p1 is destroyed, it will also try to delete i and result in the double-free vulnerability. |
| #include <memory>    void f() {  **int** \*i = new **int**;    std::shared\_ptr<**int**> p1(i);    std::shared\_ptr<**int**> p2(i);  } |

| **Compliant Code** |
| --- |
| The shared\_ptr objects are related to each other in the example below. When p2 is destroyed the reference count for the shared pointer is decremented. When p1 is destroyed, the underlying managed pointer no longer has references to it, so it is now destroyed as well. |
| #include <memory>    void f() {    std::shared\_ptr<**int**> p1 = std::make\_shared<**int**>();    std::shared\_ptr<**int**> p2(p1);  } |

|  |
| --- |
| **Principles(s):**  **2:** Static analysis tools can catch misuse of shared pointers and provide warnings prior to compilation.  **10:** "Coding standards should be defined on a per language basis” – Being aware of the features and limitations of C++ should be known so developers can be aware of vulnerabilities from misuse of shared pointers. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use | Proves the absence of runtime errors (AbsInt, n.d.) |
| PVS-Studio | 7.07 | [V1006](https://www.viva64.com/en/w/v1006/) | “Several shared\_ptr objects are initialized by the same pointer. A double memory deallocation will occur.” (PVS-Studio, n.d.) |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-010-CPP | Do not create strings from null pointers  The std::basic\_string data type is composed of traits that build up the methods and operations used on strings in C++. Passing a null pointer to some of the underlying methods in the string class can result in undefined behavior (Software Engineering Institute, 2014). |

| **Noncompliant Code** |
| --- |
| If std::getenv() fails to return a value, or fails for any other reason, it returns a null pointer. Assigning this to a string can result in unexpected behavior. |
| #include <cstdlib>  #include <string>    void f() {    std::string tmp(std::**getenv**("TMP"));    if (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The compliant code below creates a temporary pointer is created with the result of the std::getenv() operation. Then the contents are checked that it is not a null pointer prior to assigning the value to a string. If the temp pointer is a string its value is assigned, otherwise an empty string is assigned. |
| #include <cstdlib>  #include <string>    void f() {    const **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");    if (!tmp.empty()) {      // ...    }  } |

|  |
| --- |
| **Principles(s):**  **9:** Code reviews can be used to catch potential errors related to possible strings created from null pointers.  **10:** "Coding standards should be defined on a per language basis” – The features and limitations of the language (CPP) should be known and utilized |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | assert\_failure | Proves the absence of runtime errors (AbsInt, n.d.) |

## 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.

**Automated Detection Policy**

Automation used for the enforcement of and compliance to the standards in this policy will take place during the pre-production phase of the DevSecOps. During the Design phase, static analysis tools, along with other built-in IDE tools, will be used to catch and warn of errors or potential violations during development. The automatic analysis tools will hook into the Build phase and provide warnings of non-severe violations but will fail a build completely if severe violations are found. Finally, automated analysis and testing tools will be run on every successful build prior to publishing. Should automated tests fail or report a problem, the build is deemed invalid and cannot be moved into production.

## Summary of Risk Assessments

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

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

## 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 | Data at rest is data stored on a disk and not in memory (AndyMac, 2018). When data is being stored on disk, it should be encrypted prior to saving to disk. This is used to prevent other users or systems from accessing sensitive information from files created or used by a system or program. |
| Encryption at flight | Data sent between systems internally or through external means such as the internet is data in flight (AndyMac, 2018). This data should be should sent through secure encrypted channels such as HTTPS. Using encrypted channels means the data is encrypted prior to sending the information across the wire, and then decrypted on the other end, someone listening in cannot decipher or alter the request data. |
| Encryption in use | Data in use within the application should be handled securely and encrypted when necessary. For example, a user’s password should never be stored unencrypted and instead should be stored as a hashed password, then verified via a hashed password (AndyMac, 2018). This prevents the user’s raw password from being compared against within the application. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying someone is who they say they are (Bowman, 2019). This is done through the use of user logins. This can be done through the user of credentials such as usernames and passwords, biometrics such as fingerprints, or the addition of mobile based two-factor authentication using one-time passwords. This policy should be applied prior to giving the user access to any system or application. |
| Authorization | Authorization is the process of ensuring that users have appropriate access to systems based on their role (Bowman, 2019). There are three types of authorization frameworks:   * Mandatory Access Control (MAC) – access is granted based on the security of the content being accessed * Discretionary Access Control (DAC) – access is provided by the content owner. For example, a document that is created by someone else and is secure, but is shared with a specific individual * Role-Based Access Control (RBAC) – Access is granted based on an individual’s role. For example, regular users within an organization do not need administrative privileges for installing software, that privilege is restricted and granted only to staff within the IT Department.   This policy applies when determining which parts of a system or program a user should have access to, such as being able to make changes to a database or not. Multiple authorization frameworks may be used within the same system or program. |
| Accounting | After a user has been authenticated and authorized, their actions should be logged and recorded (Bowman, 2019). Audit logs and event monitoring should record a user’s actions throughout the system as this will aid in knowing whether the user needs more or less authorization, or whether the user is a malicious actor who was authenticated incorrectly. Examples of what events to log would be the addition of new users or when a user accesses a file. This policy applies to every action within the system from the moment the user attempts to log in to the point they exit, close, or log out of the system. |

**\***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 | 03/20/2020 | Coding standards added | Steven Wade | Dewin Andujar |
| 2.0 | 04/11/2020 | Template completed | Steven Wade | Dewin Andujar |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

**References**

AbsInt. (n.d.). Fast and sound static analysis. Retrieved April 11, 2021, from https://www.absint.com/astree/index.htm

AndyMac. (2018, March 28). Retrieved April 11, 2021, from https://security.stackexchange.com/a/182477

Assertions in C/C++. (2020, August 31). Retrieved March 21, 2021, from https://www.geeksforgeeks.org/assertions-cc

Axivion. (2021, March 26). Axivion suite - code quality for C, C++ and C#. Retrieved April 11, 2021, from https://www.axivion.com/en/products/axivion-suite/

Bowman, B. (2019, March 18). The AAA Framework for Identity Access Security. Retrieved April 11, 2021, from https://securityboulevard.com/2019/03/the-aaa-framework-for-identity-access-security/

Jrok. (2020, May 14). How does prepared statements in Sqlite C++ work. Retrieved March 21, 2021, from https://stackoverflow.com/a/61796041

LDRA (2019). *CWE Standards Model Summary for C / C++*. http://my.ldrasoftware.co.uk/repository//miscellaneous/Cwe\_compliance.pdf

LLVM. (2003) Assert Liberally. Retrieved March 21, 2021, from https://llvm.org/docs/CodingStandards.html#assert-liberally

Microsoft. (2019, October 21). C and C++ integer limits. Retrieved March 21, 2021, from https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits

Nidecki, T. (2020, January 06). What is integer overflow. Retrieved April 09, 2021, from https://www.acunetix.com/blog/web-security-zone/what-is-integer-overflow/

PVS-Studio. (n.d.). Studio analyzer. Retrieved April 11, 2021, from https://pvs-studio.com/

Seacord, R. C. (2013). *Secure Coding in C and C*. [MBS Direct]. (pp 87-113) Retrieved from https://mbsdirect.vitalsource.com/#/books/9780132981972/

Software Engineering Institute. (2015, February 19). ERR51-CPP. Handle all exceptions Skip to end of metadata. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

Software Engineering Institute. (2008, August 13). EXP53-CPP. Do not read uninitialized memory. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory

Software Engineering Institute. (2009, September 21). IDS00-J. Prevent SQL injection. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection

Software Engineering Institute. (2006, June 8). INT32-C. Ensure that operations on signed integers do not result in overflow. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow

Software Engineering Institute. (2008, August 13). MEM50-CPP. Do not access freed memory. Retrieved March 21, 2021, https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Software Engineering Institute. (2015, June 19). MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer. Retrieved March 21, 2021, https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM56-CPP.+Do+not+store+an+already-owned+pointer+value+in+an+unrelated+smart+pointer

Software Engineering Institute. (2008, August 13). STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Software Engineering Institute. (2014, September 8). STR51-CPP. Do not attempt to create a std::string from a null pointer. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer

Software Engineering Institute. (2006, September 21). Top 10 Secure Coding Practices. Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices?focusedCommentId=88044413