**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 | Always validate foreign data sources to minimize the possibility of input-related vulnerabilities. One should keep a healthy level of suspicion when receiving external data sources. |
| 1. Heed Compiler Warnings | When compiling code, use the highest warning level available. Any warnings detected by the compiler should be promptly eliminated through refactoring of the code. After fixing the issues detected by the compiler, use multiple other static and dynamic testing methods to ensure no security flaws exist. |
| 1. Architect and Design for Security Policies | Programmers must design their code with focus on security from the first line to the last. Proper code will be designed around the security policies predefined by the requirements gathering phase of development. |
| 1. Keep It Simple | Always strive to find the simplest solution to a problem. As code becomes more complex, the chances of an underlying security flaw increase exponentially. Simple and straightforward code leaves little chance for external manipulation. |
| 1. Default Deny | Never design code which tries to prevent certain entities from accessing data. The code should deny access to all entities by default and then define a specific set of conditions in which a user could access certain data. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege states that users should only be granted the bare minimum access privileges necessary to carry out the desired function. This means that users should only be able to access a very specific set of data or functions that will perform the tasks defined in the requirements gathering phase. |
| 1. Sanitize Data Sent to Other Systems | Data which is passed to complex subsystems should be further sanitized to prevent the unintentional invocation of functions. This should include checks to catch the various types of injection attacks. |
| 1. Practice Defense in Depth | Secure code will include multiple mitigation types to manage risk. This will ensure that if one defense mechanism fails, there are still multiple others which will prevent a breach of secure data. Never should a programmer rely on one security measure to protect against an identified risk. |
| 1. Use Effective Quality Assurance Techniques | All code should be thoroughly tested and checked for effectiveness through a dedicated quality assurance program. The findings of the tests will be documented, and any issues identified will be promptly fixed. After refactoring the code to solve the documented issues, the code should then be tested again using the same procedure. Only once the code meets a defined level of risk management should it pass a quality assessment. |
| 1. Adopt a Secure Coding Standard | Create a predefined security standard of which to measure the code against. This standard will be constant through the development process to ensure all aspects of the program follow the same guidelines. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | *Ensure that integer conversions do not result in lost or misinterpreted data*  Rationale: Converting integers can lead to lost or misinterpreted data especially when converting to a smaller type. This cast will result in truncation of the high-order bits and will result in data loss. |

| **Noncompliant Code** |
| --- |
| This code casts a signed int to an unsigned int. Since unsigned and signed integers have different value ranges, this code results in negative numbers being misinterpreted as large positive numbers. |
| #include <limits.h>    **void** func(**signed** **int** si) {    /\* Cast eliminates warning \*/    unsigned **int** ui = (unsigned **int**)si;      /\* ... \*/  }    /\* ... \*/    func(INT\_MIN); |

| **Compliant Code** |
| --- |
| The proper code to handle type conversions will include a check for negative values to catch errors before they occur. One should never directly cast an integer to a different type without first checking the value. |
| #include <limits.h>    **void** func(**signed** **int** si) {    unsigned **int** ui;  **if** (si < 0) {      /\* Handle error \*/    } **else** {      ui = (unsigned **int**)si;  /\* Cast eliminates warning \*/    }    /\* ... \*/  }  /\* ... \*/    func(INT\_MIN + 1); |

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

| **Principles(s):** Heed Compiler Warnings, Keep it Simple  Most compilers will generate a warning regarding integer conversion and properly addressing these warnings will create more resilient code. This also helps avoid unnecessary data type conversions and can simplify the development process |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Type Checks - signed integer overflow (only enabled when --platform is used)  Type Checks - dangerous sign conversion, when signed value can be negative | Open source utility which checks for several types of errors, including proper usage of the STL |
| Klocwork | 2021.4 | CXX.CAST.SIGNED\_CHAR\_TO\_INTEGER | Partial coverage pertaining to signed char to integer casts |
| clang-tidy | 15.0.0 | bugprone-signed-char-misuse  bugprone-misplaced-widening-cast  abseil-duration-conversion-cast |  |
| Coverity | 2021.12.0 | MISRA\_CAST (CWE-195) |  |
| SonarQube | 9.3 | Signed and unsigned types should not be mixed in expressions (CERT, INT31-C) |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | *Guarantee that storage for strings has sufficient space for character data and the null terminator*  Rationale: Strings are stored as character arrays and within this array exists the bit data of the characters. However, there also exists a null terminator which must be considered when defining the size of the array. Failure to do so will lead to buffer overflow errors and create instability/insecurity in the code. |

| **Noncompliant Code** |
| --- |
| This code will result in an error as it does not factor in the required null space in a character array. Therefore passing a 1024 byte sized data packet will result actually pass 1025 bytes which will overflow the 1024 character length array. |
| #include <stdio.h>    **enum** { BUF\_LENGTH = 1024 };    **void** get\_data(**void**) {  **char** buf[BUF\_LENGTH];  **if** (1 != **fscanf**(stdin, "%s", buf)) {      /\* Handle error \*/    }      /\* Rest of function \*/  } |

| **Compliant Code** |
| --- |
| This code correctly factors the required null space in a character array and only receives 1023 bytes in from the fscanf() function. Therefore the data received will be 1023 + 1 bytes which perfectly fits in the 1024 character array. |
| #include <stdio.h>    **enum** { BUF\_LENGTH = 1024 };    **void** get\_data(**void**) {  **char** buf[BUF\_LENGTH];  **if** (1 != **fscanf**(stdin, "%1023s", buf)) {      /\* Handle error \*/    }      /\* Rest of function \*/  } |

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

| **Principles(s):** Heed Compiler Warnings  Any warning presented by the compiler regarding the memory buffer should be immediately addressed. Developers who are mindful of the memory limitations of data types deliver the most robust and successful code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Bounds Checking - Partial string write that leads to buffer that is not zero terminated. |  |
| SonarQube | 9.3 | Memory access should be explicitly bounded to prevent buffer overflows (CWE-131, CWE-119, STR50-CPP) |  |
| Coverity | 2021.12.0 | BUFFER\_SIZE |  |
| Klocwork | 2021.4 | ABV.ANY\_SIZE\_ARRAY  ABV.GENERAL  ABV.ITERATOR  ABV.MEMBER  ABV.STACK  ABV.TAINTED  ABV.UNICODE.BOUND\_MAP  ABV.UNICODE.FAILED\_MAP  ABV.UNICODE.NNTS\_MAP  ABV.UNICODE.SELF\_MAP  ABV.UNKNOWN\_SIZE  NNTS.MIGHT  NNTS.MUST  NNTS.TAINTED  RABV.CHECK  RN.INDEX  SV.FMT\_STR.BAD\_SCAN\_FORMAT  SV.STRBO.BOUND\_COPY.OVERFLOW  SV.STRBO.BOUND\_COPY.UNTERM  SV.STRBO.BOUND\_SPRINTF  SV.STRBO.UNBOUND\_COPY  SV.STRBO.UNBOUND\_SPRINTF  SV.UNBOUND\_STRING\_INPUT.CIN  SV.UNBOUND\_STRING\_INPUT.FUNC | Complete coverage including file stream buffer overflows |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | *Do not attempt to create a std::string from a null pointer*  Rationale: As documented by the C++ Standard [char.traits.require], passing a null pointer to the std::char\_traits::length() function will result in undefined behavior. In this instance, the function will dereferencing a null pointer and while some libraries prevent this issue, for code portability one should avoid this behavior. |

| **Noncompliant Code** |
| --- |
| In this example, a string is created from the result of the getenv() function call. This function returns a null pointer on failure which will lead to undefined behavior. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| To alleviate this issue, a developer must check for a null pointer before the string object is constructed. The code below loads the results of the function call into a char pointer before passing to the string object. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard, ValidateInput Data  While some compilers can handle this issue, one can be certain of a code’s integrity by avoiding this entirely. As such, Green Pace can more successfully maintain a secure coding standard by ensuring developers are following this practice. This will also help ensure input data is validated through additional checks for null values. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Null pointers – null pointer dereferencing |  |
| SonarQube | 9.3 | Null pointers should not be dereferenced |  |
| Coverity | 2021.12.0 | STRING\_NULL |  |
| Klocwork | 2021.4 | NPD.CHECK.CALL.MIGHT  NPD.CHECK.CALL.MUST  NPD.CHECK.MIGHT  NPD.CHECK.MUST  NPD.CONST.CALL  NPD.CONST.DEREF  NPD.FUNC.CALL.MIGHT  NPD.FUNC.CALL.MUST  NPD.FUNC.MIGHT  NPD.FUNC.MUST  NPD.GEN.CALL.MIGHT  NPD.GEN.CALL.MUST  NPD.GEN.MIGHT  NPD.GEN.MUST  RNPD.CALL  RNPD.DEREF |  |
| clang-tidy | 15.0.0 | unix.cstrisng.NullArg (C) | Specific check for coding standard |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | *Sanitize data passed to complex subsystems*  Rationale: SQL injection attacks are among the most common ways malicious entities can access sensitive data. Improper sanitation of data can lead to the contents of entire databases leaking to attackers. This is especially concerning because formulating an injection statement is not particularly difficult to do meaning the system is likely to face multiple injection attacks in its lifespan. |

| **Noncompliant Code** |
| --- |
| The code below directly passes data received through the sprintf() function to the system() function. At best this will lead to erratic behavior and at worst it can compromise the entire system. |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| Using coding standards, this method filters whatever data is received to eliminate characters which could be used in an injection statement. Once these are filtered out, the statement can then be passed on to other subsystems. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* Cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems, Architect and Design for Security Policies, Adhere to the Principle of Least Privilege  Preventing SQL injection attacks requires that data sent and received by our system is properly sanitized and filtered for unexpected values. Developers must plan for this vector of attack when planning the structure of the system. By limiting the channels of which one can access data, Green Pace can better employ the Principle of Least Privilege. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Very High | Likely | High | Very High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.12.0 | TAINTED\_SCALAR  TAINTED\_STRING  USER\_POINTER | Partial SQL injection coverage pertaining to string values |
| Klocwork | 2021.4 | SV.STR\_PAR.UNDESIRED\_STRING\_PARAMETER  SV.TAINTED.ALLOC\_SIZE  SV.TAINTED.BINOP  SV.TAINTED.CALL.BINOP  SV.TAINTED.CALL.DEREF  SV.TAINTED.CALL.GLOBAL  SV.TAINTED.CALL.INDEX\_ACCESS  SV.TAINTED.CALL.LOOP\_BOUND  SV.TAINTED.DEREF  SV.TAINTED.FMTSTR  SV.TAINTED.GLOBAL  SV.TAINTED.INDEX\_ACCESS  SV.TAINTED.INJECTION  SV.TAINTED.LOOP\_BOUND  SV.TAINTED.PATH\_TRAVERSAL  SV.TAINTED.SECURITY\_DECISION  SV.TAINTED.XSS.REFLECTED | SV.TAINTED checks for all un-sanitized input data. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | *Avoid large stack allocations*  Rationale: The C language supports variable length arrays which can dynamically change the size of the array and reallocate the memory buffer. Failure to prevent instances where excessive data is passed to the stack can leave an application vulnerable to denial-of-service attacks and prolonged system downtime. |

| **Noncompliant Code** |
| --- |
| This code example passes data from a source file into a buffer with a size of bufsize. This variable can be manipulated to alter the size and increase the value to a point where the system can no longer handle it. |
| **int** copy\_file(**FILE** \*src, **FILE** \*dst, **size\_t** bufsize) {  **char** buf[bufsize];    **while** (**fgets**(buf, bufsize, src)) {  **if** (**fputs**(buf, dst) == EOF) {        /\* Handle error \*/      }    }    **return** 0;  } |

| **Compliant Code** |
| --- |
| In this compliant example, the malloc() function is called which returns a value upon failure that can be checked to prevent an unintended system shutdown. |
| **int** copy\_file(**FILE** \*src, **FILE** \*dst, **size\_t** bufsize) {  **if** (bufsize == 0) {      /\* Handle error \*/    }  **char** \*buf = (**char** \*)**malloc**(bufsize);  **if** (!buf) {      /\* Handle error \*/    }    **while** (**fgets**(buf, bufsize, src)) {  **if** (**fputs**(buf, dst) == EOF) {        /\* Handle error \*/      }    }    /\* ... \*/  **free**(buf);  **return** 0;  } |

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

| **Principles(s):** Architect and Design for Security Policies, Heed Compiler Warnings  Data structures must be designed with memory limitations under consideration and any issues with storage capacity can lead to compiler warnings. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Bounds Checking – Buffer Overflow |  |
| SonarQube | 9.3 | Memory access should be explicitly bounded to prevent buffer overflows |  |
| Coverity | 2021.12.0 | BUFFER\_SIZE  READLINK  SIZECHECK | Targeted check for destination size |
| Klocwork | 2021.4 | ABV.ANY\_SIZE\_ARRAY  ABV.GENERAL  ABV.ITERATOR  ABV.MEMBER  ABV.STACK  ABV.TAINTED  ABV.UNICODE.BOUND\_MAP  ABV.UNICODE.FAILED\_MAP  ABV.UNICODE.NNTS\_MAP  ABV.UNICODE.SELF\_MAP  ABV.UNKNOWN\_SIZE  NNTS.MIGHT  NNTS.MUST  NNTS.TAINTED  RABV.CHECK  RN.INDEX  SV.FMT\_STR.BAD\_SCAN\_FORMAT  SV.STRBO.BOUND\_COPY.OVERFLOW  SV.STRBO.BOUND\_COPY.UNTERM  SV.STRBO.BOUND\_SPRINTF  SV.STRBO.UNBOUND\_COPY  SV.STRBO.UNBOUND\_SPRINTF  SV.UNBOUND\_STRING\_INPUT.CIN  SV.UNBOUND\_STRING\_INPUT.FUNC | Complete coverage including file stream buffer overflows |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | *Incorporate diagnostic tests using assertions*  Rationale: When testing code for errors, assertions can be very useful to debug a system. However, these should always be turned off before deployment as some types of failed assertions can lead to denial-of-service attacks. When testing, this method will output information about a failed assertion and call the abort() function to prevent major system failure. |

| **Noncompliant Code** |
| --- |
| In this example, the assert function is used to verify if memory allocation succeeded. However, memory allocation can vary over the lifetime of the program and upon exhaustion, the program must handle it gracefully. The assert function will not gracefully handle memory exhaustion and will instead abruptly terminate the program. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char** \*)**malloc**(len + 1);  **assert**(NULL != dup);    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

| **Compliant Code** |
| --- |
| The compliant code below verifies memory allocation using the malloc() function much like the previous example which prevented inputs which exceeded the buffer size. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char**\*)**malloc**(len + 1);    /\* Detect and handle memory allocation error \*/  **if** (NULL == dup) {  **return** NULL;    }    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques, Adopt a Secure Coding Standard, Architect and Design for Security Policies  Assertions can provide a powerful tool for developers to test the effectiveness and behavior of their code. But it is important that the debug and release versions are separate and accounted for in the initial design of the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.6 | Warn if there are side effects in assert statements (since this cause different behavior in debug/release builds). |  |
| clang-tidy | 15.0.0 | bugprone-assert-side-effect  misc-static-assert | Comprehensive checks for assert in release builds |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | *Handle all exceptions thrown before main() begins executing*  Rationale: When a program throws an exception, it can be caught and handled within a try-catch block. However, if an exception is thrown after the main() method begins executing, there is no longer an opportunity to handle it and can result in the program terminating abruptly. |

| **Noncompliant Code** |
| --- |
| The below code can throw an exception after the globalS object is constructed during program startup. |
| **struct** S {    S() noexcept(**false**);  };    **static** S globalS; |

| **Compliant Code** |
| --- |
| The compliant way to handle this exception is to create a try-catch block within a local globalS object so any exceptions are caught at the function call and not at program startup. |
| **struct** S {    S() noexcept(**false**);  };    S &globalS() {  **try** {  **static** S s;  **return** s;    } **catch** (...) {      // Handle error, perhaps by logging it and gracefully terminating the application.    }    // Unreachable.  } |

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

| **Principles(s):** Practice Defense in Depth, Adopt a Secure Coding Standard, ValidateInput Data  This standard prevents any unexpected outcomes or values from running within the main() portion of the code. As such, this gives depth to the code by ensuring no statement blindly outputs data and it is instead validated prior to finalization. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2021.12.0 | UNCAUGHT\_EXCEPT |  |
| SonarQube | 9.3 | "std::uncaught\_exception" should not be used |  |
| clang-tidy | 15.0.0 | bugprone-unhandled-exception-at-new  bugprone-exception-escape  hicpp-exception-baseclass  modernize-use-uncaught-exceptions | Coverage of unhandled exceptions and proper throw type |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | *Do not invoke virtual functions from constructors or destructors*  Rationale: Virtual functions allow for member functions to be called at run time based on the dynamic type of the object of which it is being called on. This allows developers to better design applications around the object-oriented standard. However, according to the C++ Standard, if a virtual function is called within a constructor or a destructor of a class’s data members it will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class seizes an objects resources through a virtual call in the constructor. However, the constructor for B() calls B::sieze() which will lead to undefined behavior. |
| **struct** B {    B() { seize(); }  **virtual** ~B() { release(); }    **protected**:  **virtual** **void** seize();  **virtual** **void** release();  };    **struct** D : B {  **virtual** ~D() = **default**;    **protected**:  **void** seize() override {      B::seize();      // Get derived resources...    }    **void** release() override {      // Release derived resources...      B::release();    }  }; |

| **Compliant Code** |
| --- |
| The code in this example fixes the previous issue by instead calling non virtual private functions in the constructor class. |
| **class** B {  **void** seize\_mine();  **void** release\_mine();    **public**:    B() { seize\_mine(); }  **virtual** ~B() { release\_mine(); }    **protected**:  **virtual** **void** seize() { seize\_mine(); }  **virtual** **void** release() { release\_mine(); }  };    **class** D : **public** B {  **void** seize\_mine();  **void** release\_mine();    **public**:    D() { seize\_mine(); }  **virtual** ~D() { release\_mine(); }    **protected**:  **void** seize() override {      B::seize();      seize\_mine();    }    **void** release() override {      release\_mine();      B::release();    }  }; |

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

| **Principles(s):** Adopt a Secure Coding Standard, Keep It Simple, Heed Compiler Warnings  Developers following this standard avoid unnecessarily complicated code and avoid compiler warnings thrown by improper use of virtual functions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | Constructors and destructors should only use defined methods and fields | Explicit detection of coding standard |
| Coverity | 2021.12.0 | VIRTUAL\_DTOR | Partial coverage of standard with respect to virtual destructors |
| Cppcheck | 2.6 | Class - Call of pure virtual function in constructor/destructor |  |
| Klocwork | 2021.4 | CL.MLK.VIRTUAL  CWARN.DTOR.NONVIRT.DELETE  CWARN.DTOR.NONVIRT.NOTEMPTY | Partial coverage of standard with respect to virtual destructors |
| clang-tidy | 15.0.0 | cppcoreguidelines-virtual-class-destructor  cppcoreguidelines-explicit-virtual-functions  bugprone-virtual-near-miss | Near miss detects similar naming conventions. Good for preventing human error. |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-009-CPP] | *Do not call a function with a mismatched language linkage*  Rationale: With modern programs containing many dependencies in multiple programming languages, it is important to properly access data according to the C++ Standard. Linking other languages to C++ is provided in the standard documentation, but one must ensure the function point which it links to is of the same type. If the language is mismatched, then undefined behavior will occur. |

| **Noncompliant Code** |
| --- |
| The below code attempts to call a java language linkage, however the function is instead passed a C++ language callback func(). |
| **extern** "java" **typedef** **void** (\*java\_callback)(**int**);    **extern** **void** call\_java\_fn\_ptr(java\_callback callback);  **void** callback\_func(**int**);    **void** f() {    call\_java\_fn\_ptr(callback\_func);  } |

| **Compliant Code** |
| --- |
| This compliant code example properly declares callback func() as a java language pointer which matches the function type of which it is being passed to. |
| **extern** "java" **typedef** **void** (\*java\_callback)(**int**);    **extern** **void** call\_java\_fn\_ptr(java\_callback callback);  **extern** "java" **void** callback\_func(**int**);    **void** f() {    call\_java\_fn\_ptr(callback\_func);  } |

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

| **Principles(s):** Architect and Design for Security Policies, Keep It Simple, Adopt a Secure Coding Standard  Often programs will require certain API’s to work properly and developers should plan on which ones will be required in a project. Once determined, the development process should strictly follow these guidelines to maintain a secure and simple code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | "extern" shouldn't be used on member definitions | Checks for variable linkage |
| clang-tidy | 15.0.0 | mpi-type-mismatch | Only checks MPI datatype pairs match |
| Cppcheck | 2.6 | STL usage – mismatching containers in calls |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| API’s | [STD-010-CPP] | *Functions should validate their parameters*  Rationale: Much like how many different languages can be utilized in a single application, API’s are often used to add additional functionality through extensively tested libraries. However, this creates a large stream of data being both sent and received externally. The API function calls often require certain predefined parameters, and one must ensure the proper data is being passed on to avoid invalid function returns. |

| **Noncompliant Code** |
| --- |
| Below, the code shows an example of a pointer file being passed to the setfile() function with no validation. Like injection attacks, one could pass an invalid file which could corrupt the internal library and create security flaws. |
| /\* Sets some internal state in the library \*/  **extern** **int** setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** **int** usefile();    **static** **FILE** \*myFile;    **void** setfile(**FILE** \*file) {      myFile = file;  }    **void** usefile(**void**) {      /\* Perform some action here \*/  } |

| **Compliant Code** |
| --- |
| This example corrects the previous issues through validation of the function parameters and the internal state. Additionally, the program will leave the state unchanged upon any errors that are returned. |
| /\* Sets some internal state in the library \*/  **extern** errno\_t setfile(**FILE** \*file);    /\* Performs some action using the file passed earlier \*/  **extern** errno\_t usefile(**void**);    **static** **FILE** \*myFile;    errno\_t setfile(**FILE** \*file) {  **if** (file && !**ferror**(file) && !**feof**(file)) {      myFile = file;  **return** 0;    }      /\* Error safety: leave myFile unchanged \*/  **return** -1;  }    errno\_t usefile(**void**) {  **if** (!myFile) **return** -1;        /\*       \* Perform other checks if needed; return       \* error condition.       \*/        /\* Perform some action here \*/  **return** 0;  } |

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

| **Principles(s):** Default Deny, ValidateInput Data, Sanitize Data Sent to Other Systems  Proper implementation of this standard will specify certain conditions to be met for data to move to the next location in the system. Therefore it will by default, deny entry to the next location unless these specific conditions are met. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 9.3 | Appropriate arguments should be passed to stream functions | Verifies correct values are passed to functions |
| Cppcheck | 2.6 | IO Using Format String - File input/output without positioning results in undefined behavior | Partial coverage of file IO preventing undefined behavior |
| Klocwork | 2021.4 | CXX.SQL.INJECT  SV.STR\_PAR.UNDESIRED\_STRING\_PARAMETER  SV.TAINTED | Explicit SQL injection checker |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



The diagram presented above represents the standard DevSecOps security pipeline which should be followed throughout the development process. However, the automation tools discussed in this document will prove most useful and relevant during the span between the Build stage and the Respond stage. The developers at Green Pace should actively build data structures and code with integration of static testing tools. These tools will then be used extensively to verify and validate the code before deployment. After deployment, some of the automation tools can be transitioned into monitoring tools that your team will use to detect any issues which may arise. Should any unforeseen security issues present themselves after deployment, these automations can assist the team in responding to the threat in a timely manner.

### Summary of Risk Assessments

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

### Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Any data being stored at any point within the system shall be considered at rest. Sensitive information relating to clients, logins, or financials should always be encrypted when at rest. Data which is stored as plaintext is at an extreme risk of becoming compromised leading to legal, financial, and moral liabilities as well as a loss of trust. It is recommended to use the Advanced Encryption Standard (AES) for any confidential data at rest. |
| Encryption at flight | Data at flight or data in transit refers to data which is being moved from one point to another. This can make it vulnerable to interception by malicious entities and a secure defense must be enacted to mitigate this risk. One must not only consider data in transit within their system, but also data which is leaving/arriving to or from their system. Secure transmission channels must be established to protect the integrity of the data and several encryption protocols exist for this purpose. For example, Transport Layer Security (TLS) is a common encryption method for data in transit. |
| Encryption in use | Often overlooked, data in use references data which is currently being created, worked on, or viewed by users. This state is also at risk from many of the same threats as the other data states, but one must also consider the human aspect. For example, if an employee is working on a document with sensitive information and steps away to get coffee without locking their PC, a malicious entity could compromise the information or modify it during the creation of the data. Therefore, confidential computing practices should be strictly enforced using multiple vectors of defense including real-time encryption of new data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The first ‘A’ in the Triple-A Framework is the process of ensuring that only those who are authorized to access specific data are allowed on the system. Commonly, usernames and passwords are used to verify entities on the system, but this is rapidly becoming insufficient. Best practice includes some form of Multi-Factor Authentication (MFA) where an additional uniquely generated code is required for each login. This can mitigate the risk of compromised login credentials allowing unauthorized entities into the system. An active and robust monitoring program should be enacted, where all logons are tracked, and suspicious attempts are reported. |
| Authorization | While similar to authentication, the next step of authorization refers to granting users access to different categories of data. One must ensure that only those who are authorized to view and modify data have those permissions. It is critical that the principle of Least Privilege is followed here which translates to giving users access to only the bare minimum data that is necessary for their job. An efficient system will categorize data based on levels of sensitivity and an Administrator will personally review, authorize, and assign users the correct level of access. |
| Accounting | The final step in the framework maintains the effectiveness of the previous steps. Accounting within data security includes the monitoring of system activities to verify that policies are followed. System administrators shall be at the head of this process, and they can be aided by automated utilities to generate reports and logs for review. These documents should track any changes to the database, user logons, user access levels, and data accessed by users. The Administrator must then review this information and be vigilant for any suspicious activity. An efficient system may also employ some automated security measures to block access if the entity is unfamiliar. When storing these records, it is mandatory to use an external security server to prevent unauthorized modifications in an attempt to hide certain activities. |

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 | 01/31/2022 | Preliminary security outline | Gary Clark |  |
| 2.0 | 02/12/2022 | Comprehensive Security Policy | Gary Clark |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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