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



# Green Pace Secure Development Policy

# Anthony Baratti

## 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 | It is vitally important to validate all data from untrusted sources including command line arguments, user-supplied files, network (online communication) interfaces, environmental variables, and other data from external sources. |
| 1. Heed Compiler Warnings | Use the highest level of compiler warnings. Review the code for errors and warnings to find potential flaws. Use static and dynamic testing tools to determine further flaws to rework the code and reduce potential vulnerabilities. |
| 1. Architect and Design for Security Policies | Plan the architecture and design according to the security needs of the project. Separation of elements can be a key component to developing secure coding and implementing best practices (such as grouping higher privilege functions away from other functions) |
| 1. Keep It Simple | Complexity of architecture or code increases the risk of flaws and vulnerabilities. Keep the code concise, separated, and simple. The more complex security measures have to be made, the more likely it is they will have flaws. |
| 1. Default Deny | Use expected permissions rather than comprehensive exclusion during decision-making in code. Access should only be permitted if appropriate conditions are met, rather than if inappropriate conditions have been used. Deny all except appropriate conditions. |
| 1. Adhere to the Principle of Least Privilege | Access should only be given as needed, to those who need it. Practice separation of access between secure levels of a program and practice removing privilege once the actions taken are completed to reduce the amount of time a user has access to sensitive data. Elevate privileges only as a necessity, then de-elevate once the need is no longer required. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data is a key component for security. Before passing data from a source to a sub-system, ensure that the data is being passed as it should be (appropriate data type, size, location, destination, and source). The invoked process does not understand the context, so the calling process must be responsible for ensuring data is passed safely. |
| 1. Practice Defense in Depth | Defense in depths prepares multiple layers of strategies used to mitigate security vulnerabilities. Combining techniques and practices allows a system to double down (or more) on preventing risks. This can also help mitigate and contain exploits as they happen (such as removing higher privilege). |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is also part of a layered defense, utilizing various forms of testing (such as fuzz and penetration testing) along with static code reviews to ensure that vulnerabilities and assumptions have been minimized. It is often the case to use a specialized professional or group (3rd party auditor) to review code before production. |
| 1. Adopt a Secure Coding Standard | Choose an appropriate coding standard to apply to C/C++ and practice implementations with every project. |

### 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** | [DCL-007-C] | Functions must be declared with appropriate type information. Parameter variables and return types must also be accurately informed on their data type. Failure to type specify the function and its parameter/returns will result in compiler errors. The preferred way to enforce function declarators is to include a well defined header file containing the constructed function. |

| **Noncompliant Code** |
| --- |
| The function “add” was declared with 3 integer parameters. The function pointer in the non-compliant code has been initialized to receive 2 integer parameters instead of 3, which could potentially lead to unexpected data access by the called function. |
| **int** add(**int** x, **int** y, **int** z) {  **return** x + y + z;  }    **int** main(**int** argc, **char** \*argv[]) {  **int** (\*fn\_ptr) (**int**, **int**);  **int** res;    fn\_ptr = add;    res = fn\_ptr(2, 3);  /\* Incorrect \*/    /\* ... \*/  **return** 0;  } |

| **Compliant Code** |
| --- |
| To correct the code to compliance, properly declaring a function call (and its pointer) will prevent unintended or unexpected behavior in the program. |
| **int** add(**int** x, **int** y, **int** z) {  **return** x + y + z;  }    **int** main(**int** argc, **char** \*argv[]) {  **int** (\*fn\_ptr) (**int**, **int**, **int**) ; /\* Correct initialization parameters. \*/  **int** res;    fn\_ptr = add;    res = fn\_ptr(2, 3, 4); /\* Correct passing of parameters \*/    /\* ... \*/  **return** 0;  } |

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

| **Principles(s):** Validate Input Data – Excepting appropriate data types is essential for a program to operate without bugs or flaws. Initializing the datatypes and return types helps keep the code concise, clean, and clear. It allows for developers to design quick and easy fail-safes to securely handle errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DL07 |  |
| CodeSonar | 9.0p0 | LANG.FUNCS.PROT  LANG.STRUCT.DECL.IMPT | Incomplete function prototype  Implicit Type |
| ÉCLAIR | 1.2 | CC2.DCL07 | Fully Implemented |
| GCC | 4.3.5 |  | Can detect violation of this recommendation when the -Wstrict- prototypes flag is used. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [INT-008-C] | Verify that all integer values are in range – Values resulting from integer operations must be within range of the integer type (long, long long, short, etc.). The operations also include iteration values (where iterations may become extremely high values). Verifiable in-range operations are preferred to error handling to prevent denial of service situations. |

| **Noncompliant Code** |
| --- |
| i + 1 in this scenario will overflow the 16-bit structure of int value. The if statement is designed to capture the error, but can be replaced with a proper data type to hold the data value. |
| **int** i = /\* Expression that evaluates to the value 32767 \*/;  /\* ... \*/  **if** (i + 1 <= i) {    /\* Handle overflow \*/  }  /\* Expression involving i + 1 \*/ |

| **Compliant Code** |
| --- |
| In this scenario where i is known to evaluate to a MAX 16-bit value, no error handling is required for the expression i + 1 to evaluate without an overflow. In this scenario, trusted data or sanitized data (that will ensure no overflow) should be used (such as restricting value input or number of operations performed). |
| **long** i = /\* Expression that evaluates to the value 32767 \*/;  /\* ... \*/  /\* No test is necessary; i is known not to overflow \*/  /\* Expression involving i + 1 \*/ |

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

| **Principles(s):** Validate Input Data – Ensuring that integers are within the range of the declared datatype will keep the system from overflow errors. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | High | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | Integer-overflow | Fully Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT08 |  |
| CodeSonar | 9.0p0 | 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 | Addition Overflow of Allocation Size  Integer Overflow of Allocation Size  Multiplication Overflow of Allocation Size  Subtraction Underflow of Allocation Size  Addition Overflow of Size  Unreasonable Size Argument  Multiplication Overflow of Size  Subtraction Underflow of Size |
| Compass/ROSE |  |  | Could detect violations of this recommendation by flagging any comparison expression involving addition that could potentially overflow. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STR-031-C] | Guarantee that storage for string has sufficient space for character data AND null terminator – copying data to a buffer without enough space will result in a buffer overflow and can lead to unexpected behavior or access to memory outside of the buffer. Limiting copies through truncation or ensuring sufficient size to hold the character data will prevent string buffer overflow. |

| **Noncompliant Code** |
| --- |
| This portion of code receives presents an “off-by-one” scenario where a copy is being made for each character but the loop does not account for the final null-termination character. This may allow writing data past the end of dest. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

| **Compliant Code** |
| --- |
| In this revised code, the for loop utilized the n (number of characters to copy) – 1, which leaves 1 space to write the null-terminated character to the end of the buffer which will prevent an overflow in this scenario. |
| #include <stddef.h>    **void** copy(**size\_t** n, **char** src[n], **char** dest[n]) {  **size\_t** i;    **for** (i = 0; src[i] && (i < n - 1); ++i) {       dest[i] = src[i];     }     dest[i] = '\0';  } |

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

| **Principles(s):** Validate Input Data – Ensuring that there is enough buffer space for string data to prevent memory leaks and overflows.  Practice Defense In Depth – Ensures that memory is not accessed beyond the controlled scope of the data in question. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported  Astree reports all buffer overflows resulting from copying data to a buffer that is not large enough |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string functions that may cause buffer overflow  Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 9.0p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO\* | Buffer Overrun  Type Overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Compass/Rose |  |  | Can detect violations of the rule. However, it is unable to handle cases involving strcpy\_s() or manual string copies. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STR-002-C] | Sanitize data passed to complex subsystems – String data may contain special characters that can be used to trigger behavior. This can result in vulnerabilities, especially if used maliciously. Sanitizing and passing the resulting string so it is not harmful or malicious is key to mitigation. |

| **Noncompliant Code** |
| --- |
| This portion of code receives input from an untrusted source which could allow and uses it directly within the login function. This can result in malicious code being inserted and the program receiving it as expected code (Sun Solaris TELNET system). |
| (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")),    0); |

| **Compliant Code** |
| --- |
| The “- -“ double dash symbol on line 6 causes the getopt() function to stop parsing command line arguments, which means that the user input for “USER” will not be parsed and will be read as an entire piece of data, rendering it innate and sanitized as a string argument. |
| (**void**) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")), 0); |

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

| **Principles(s):** Validate Input Data – Ensures that untrusted data will not have unintended behavior.  Sanitize Data Sent To Other Systems – Ensures that data is passed as it should be, such as datatype, size, range, etc. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 9.0p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully Implemented |
| Klocwork | 2024.4 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [MEM-030-C] | Do not access freed memory – Attempting to access deallocated memory results in undefined behavior. Pointers leading to freed memory result in dangling pointers which can lead to exploitable vulnerabilities. All pointers that lead to freed memory must become invalid/null. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code, free(buf) is called after it is allocated and written to, but then is copied to after it is freed. Generally, allocations and frees are further apart in code, so they are much harder to recognize than this simple example. |
| #include <stdlib.h>  #include <string.h>    **int** main(**int** argc, **char** \*argv[]) {  **char** \*return\_val = 0;  **const** **size\_t** bufsize = **strlen**(argv[0]) + 1;  **char** \*buf = (**char** \*)**malloc**(bufsize);  **if** (!buf) {  **return** EXIT\_FAILURE;    }    /\* ... \*/  **free**(buf);    /\* ... \*/  **strcpy**(buf, argv[0]);    /\* ... \*/  **return** EXIT\_SUCCESS;  } |

| **Compliant Code** |
| --- |
| Structuring the code to free the memory after all of its uses is essential at preventing this memory flaw. Static code reviews can be beneficial in reducing these risks. |
| #include <stdlib.h>  #include <string.h>    **int** main(**int** argc, **char** \*argv[]) {  **char** \*return\_val = 0;  **const** **size\_t** bufsize = **strlen**(argv[0]) + 1;  **char** \*buf = (**char** \*)**malloc**(bufsize);  **if** (!buf) {  **return** EXIT\_FAILURE;    }    /\* ... \*/  **strcpy**(buf, argv[0]);    /\* ... \*/  **free**(buf);  **return** EXIT\_SUCCESS;  } |

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

| **Principles(s):** Use Effective Quality Assurance – Static code reviews can help prevent accessing deallocated memory. Ensuring that all pointers are managed appropriately will help keep a system secure from accessing freed memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | dangling\_pointer\_use | Supported  Astree reports all accesses to freed allocated memory |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses after its deallocated and double memory deallocations |
| CodeSonar | 9.0p0 | ALLOC.UAF | Use after free |
| Compass/ROSE |  |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [MSC-011-C] | Incorporate diagnostic tests using assertions – When assertions execute, a false assertion should output information about the failed assertion (including arguments, name of source file, line number, and name of function). |

| **Noncompliant Code** |
| --- |
| Using an assert() call in this code is not able to gracefully handle (recover or fail safely) from memory exhaustion. Using the assert method might lead to an abrupt termination of the process which can lead to denial-of-service attacks. |
| **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 appropriate solution requires an if statement to detect and handle memory exhaustion. |
| **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 – Static and dynamic testing can reveal bugs and vulnerabilities in code. Be sure to test components and variables of software for faults.  Adopt A Secure Coding Standard – Regularly testing to ensure appropriately functionality (using positive and negative tests) can help reduce bugs and vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 9.0p0 | LANG.FUNCS.ASSERTS | Not enough assertions. |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect |
| Parasoft C/C++test | 2024.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |
|  |  |  |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [ERR-051-CPP] | Handle all exceptions – All exceptions thrown by an application must be caught by a matching exception handler, even if the exception can’t be gracefully recovered. Using proper handlers ensures the stack is properly unwound and the external resources are managed before termination. |

| **Noncompliant Code** |
| --- |
| Here, neither f() nor main() catch any exceptions that throwing\_func() might cause. As such, terminate() will be called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In the compliant code, a try/catch block is placed within main to attempt f() and handle all exceptions which ensures the stack is unwound up to main(). |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Appropriate exception handling is a vital part of standard coding practices. Handling errors and failing gracefully can help keep code secure.  Practice Defense in Depth – Exception handling is a layer of defense that can be applied to mitigate vulnerabilities and bugs in software. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | main-function-catch-all  early-catch-all | Partially Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 |  |
| CodeSonar | 9.0p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAX | 2024.4 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Operators | [OOP-055-CPP] | Do not use pointer-to-member operators to access nonexistent members – Do not user pointer-to-member expression (A ->\*a) where the dynamic type of the first operand does not contain the member to which the second operand refers. |

| **Noncompliant Code** |
| --- |
| In this example, a null pointer-to-member value is passed as the second operand in the pointer-to-member expression and results in undefined behavior because \*gptr was not explicitly initialized (resulting in null). |
| **struct** B {  **virtual** ~B() = **default**;  };    **struct** D : B {  **virtual** ~D() = **default**;  **virtual** **void** g() { /\* ... \*/ }  };    **static** **void** (D::\*gptr)(); // Not explicitly initialized, defaults to nullptr.  **void** call\_memptr(D \*ptr) {    (ptr->\*gptr)();  }    **void** f() {    D \*d = **new** D;    call\_memptr(d);  **delete** d;  } |

| **Compliant Code** |
| --- |
| Here, D::\*gptr is initialized referencing D and the g method (&D::g;). This initializes the D::\*gptr to a value, making it non-null (explicitly initialized). |
| **struct** B {  **virtual** ~B() = **default**;  };    **struct** D : B {  **virtual** ~D() = **default**;  **virtual** **void** g() { /\* ... \*/ }  };    **static** **void** (D::\*gptr)() = &D::g; // Explicitly initialized.  **void** call\_memptr(D \*ptr) {    (ptr->\*gptr)();  }    **void** f() {    D \*d = **new** D;    call\_memptr(d);  **delete** d;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Using coding best practices can help eliminate bugs and errors. Not using pointer-to-member operators to access non-existent members can mitigate these concerns.  Use Effective Quality Assurance Techniques – Static code analysis and unit testing can help ensure that OOP-055 is not violated. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Overflow\_upon\_dereference  Invalid\_function\_pointer |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP55 |  |
| CodeSonar | 9.0p0 | LANG.MEM.UVAR | Uninitialized Variable |
| Helix QAC | 2024.4 | DF2810, DF2811, DF2812, DF2813, DF2814 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input/Output | [FIO-030-C] | Exclude user input from format strings – Never call a formatted I/O function with a format string containing a tainted value (generally untrusted data). Failure to properly receive input and generate output can result in viewing stack contents, memory content, and writing to an arbitrary location. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password method is called during authentication to display an error message if the user or password is incorrect. The function accepts user name as a string referenced by the user (which is untrusted data). The function constructs an error message that outputs to stderr using fprintf() function. Because the resulting code contains untrusted user input, a format-string vulnerability is created when it is passed to the fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code** |
| --- |
| Using fputs() instead of fprintf() with the formatted message allows the message to be directly output instead of parsing/evaluating the contents. This removes the string formatting and removes the related vulnerability. \*NOTE Using fprintf() with 3 arguments: (msg, stderr, user) can also mitigate the format vulnerability. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    **void** incorrect\_password(**const** **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/  **static** **const** **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + **sizeof**(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);  **if** (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);  **if** (ret < 0) {      /\* Handle error \*/    } **else** **if** (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

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

| **Principles(s):** Keep it Simple – Do not allow user input in formatted strings. Only use hardcoded variables and strings to deliver formatted data. This can mitigate a multitude of problems such as crashing vulnerable processes, viewing stack or memory content, or write to a memory location.  Use Effective Quality Assurance Techniques – Using static code review can help prevent user input into formatted strings which will mitigate unintended program behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO030 | Partially Implemented |
| CodeSonar | 9.0p0 | IO.INJ.FMT  MISC.FMT | Format String Injection  Format String |
| Compass/ROSE |  |  |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Sensitive Information | [MSC-041-C] | Never hard code sensitive information – Hard coding sensitive data (passwords, keys, health records, etc.) can expose information to attackers. This is not limited to external threats, but could also create a compromise of systems or legalities internally (such as shared company software). |

| **Noncompliant Code** |
| --- |
| The noncompliant code must authenticate to a remote service using authenticate, and is passed the authentication code to the function via string literal. The authentication code exists in the program’s binary executable which can easily be discovered. |
| /\* Returns nonzero if authenticated \*/  **int** authenticate(**const** **char**\* code);    **int** main() {  **if** (!authenticate("correct code")) {  **printf**("Authentication error\n");  **return** -1;    }    **printf**("Authentication successful\n");    // ...Work with system...  **return** 0;  } |

| **Compliant Code** |
| --- |
| This solution requires the user to supply the code and then securely erases it using memset\_s(). The program can read |
| /\* Returns nonzero if authenticated \*/  **int** authenticate(**const** **char**\* code);    **int** main() {  #define CODE\_LEN 50  **char** code[CODE\_LEN];  **printf**("Please enter your authentication code:\n");  **fgets**(code, **sizeof**(code), stdin);  **int** flag = authenticate(code);    memset\_s(code, **sizeof**(code), 0, **sizeof**(code));  **if** (!flag) {  **printf**("Access denied\n");  **return** -1;    }  **printf**("Access granted\n");    // ...Work with system...  **return** 0;  } |

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

| **Principles(s):** Architect & Design for Security Policies – Hardcoding sensitive data can reveal to attackers, which can allow further access into the system (such as encryption keys or passwords). Securing and separating sensitive data is a vital step to mitigating this security threat.  Adopt a Secure Coding Standard – Practicing secure coding (separate and secure sensitive information) on every project will reduce the opportunity for attackers to gain access through normally accessed portions of software (such as obtaining actual credentials and validly entering the system). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported |
| CodeSonar | 9.0p0 | HARDCODED.AUTH HARDCODED.DNS HARDCODED.KEY HARDCODED.SALT HARDCODED.SEED | Hardcoded Authentication Hardcoded DNS Name Hardcoded Crypto Key Hardcoded Crypto Salt Hardcoded Seed in PRNG |
| Helix QAC | 2024.4 | C3122  C++3842 |  |
| Klocwork | 2024.4 | HCC  HCC.PWD  HCC.USER  CXX.SV.PWD.PLAIN  CXX.SV.PWD.PLAIN.LENGTH  CXX.SV.PWD.PLAIN.ZERO |  |

### 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.

Security can be applied at every step of the DevOps Cycle.

* Assessment & Planning: Automated threat modeling can analyze proposed designs & architecture, as well as process results from feedback fed through dashboards or alerts from the production environment (post-production security flaws input back to development). Some tools can even have security policies and requirements embedded into them during planning to help realize security goals being met.
* Development: Static analysis tools can scan code for security vulnerabilities as it is written or committed.
* Build/Integration (CI/CD): Dependency check tools can be used to scan open source and third-party libraries for vulnerabilities (such as OWASP Dependency Check). Automated policy enforcement tools can be used to prevent builds or deployments if security policies are violated.
* Testing: Dynamic testing tools can scan programs as they run for vulnerabilities or misconfigurations.
* Deployment & Operation (Production): Runtime Analysis tools can be used to monitor, report, or even block threats as an application is running. Security monitoring and logging activity can help detect anomalies and provide early detection of security threats or breaches.
* Respond/Maintain & Stabilize: Providing results from security tools back to development can help remove vulnerabilities. Some security tools can even block and report threats, allowing developers to roll back versions if necessary. Utilizing the feedback from automation can be crucial for early threat detection and mitigation.

### 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 |
| STR-002-C | High | Likely | Medium | High | 1 |
| DLC-007-C | Low | Unlikely | Low | Low | 3 |
| INT-008-C | Medium | Probable | High | Low | 3 |
| MSC-011-C | Low | Unlikely | High | Low | 3 |
| FIO-030-C | High | Likely | Medium | High | 1 |
| MEM-030-C | High | Likely | Medium | High | 1 |
| STR-031-C | High | Likely | Medium | High | 1 |
| MSC-041-C | High | Probable | Medium | Medium | 1 |
| ERR-051-CPP | Low | Probable | High | Medium | 3 |
| OOP-055-CPP | High | Probable | Medium | Medium | 3 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 that is stored either internally or externally (such as a hard drive or a database). This data can be particularly valuable because it generally contains sensitive information (passwords, keys, personally identifying information, etc.). Using encryption on data at rest helps keep the information protected by not revealing its contents to any unauthorized party. Data at rest should always be kept secure and away from the methods to unlock it, such as encryption algorithms or keys. Third-party companies can provide encryption at rest (such as cloud services). Encrypting data at rest should be part of a standard security policy and a defense-in-depth strategy. |
| Encryption in flight | Encryption in flight refers to data in motion. Data being transmitted from one endpoint to another is the most vulnerable data because it is susceptible to interception. Encrypting data in motion must be handled with care, using only the most secure and trusted encryption methods available. This is especially true for protected data such as banking transactions or medical records (including email and messaging devices, not just HTTP requests through webpage applications). Third-party companies or external libraries can help manage encryption in flight. Encrypting sensitive data in flight should be part of a standard security policy and a defense-in-depth strategy |
| Encryption in use | Encryption in use is protecting data that is currently being processed (actively being input, read, updated, or accessed). Encryption at rest strategies allow data to be processed without first decrypting it, which can expose sensitive data to attackers. This is particularly important during external processes (3rd party information processing), frequent transfers between processing components, and cloud computing. Encryption in use can be computationally expensive, so choosing which data is encrypted in use is also important. This process can be managed by a trusted execution environment, which privately processes data (either as hardware components in memory or algorithms implemented within the software) to protect plain text information. Encrypting data in use should be a part of a standard security policy and a defense-in-depth strategy. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a way of determining whether a user is who they claim to be. A simple form of authentication is a username and password, but it can include serialization, pass keys/security tokens, or even biometric ID such as fingerprint scans. (among other forms of authentication). Authentication helps ensure that the specific user is allowed access to the requested part of the system and provides a confirmation of such. Adding new users should be done with care, and complex passwords should be prompted or generated. It is also valuable to consider multi-factor authentication, such as SMS text codes or security questions.. This follows adopting secure coding practices and defense-in-depth principles. |
| Authorization | Authorization is a way of allowing (or restricting) access to components of the system. By using separation of concerns, data and functionality can be tiered based on need, and the principle of least privilege (user access level) can be applied to reduce unauthorized access to authenticated or even unauthenticated users. It is also a way to manage human error, should a portion of the system be restricted to those with proper rights (for example, those who have been trained or have the appropriate legal right to access). This also follows adopting secure coding practices and implementing defense-in-depth principles. |
| Accounting | Accounting is a way of measuring or tracking the use of the system. Whether by external requests or internal requests, accounting can be akin to auditing. Using proper data tracking (such as elements accessed, information sent or received, dates and times, as well as errors/unauthorized or excessive attempts) with logging can help alert software handlers to potential threats (or bugs for that matter). Accounting can provide other meaningful insights into how the system is accessed and used. This helps keep the system secure by adding another layer of defense-in-depth. |

**\***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 | 03/23/2025 | Updated Principles  Updated Standards | Anthony Baratti |  |
| 1.2 | 04/13/2025 | Updated Risk Assessment  Updated Automation Detection Tools  Updated Automation Process  Updated Mapping Principles  Updated Encryption & Triple A Policies | Anthony Baratti |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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