**`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 | Validation of input data is an essential aspect of security in software. This principle ensures that input data is acceptable and safe for the program to process, protecting the system from malicious attacks or unknown behavior. |
| 1. Heed Compiler Warnings | Compiler warnings provide great insight into certain behaviors of software such as implicit conversions or unreachable code. Warnings should be reviewed even though they do not always require action. |
| 1. Architect and Design for Security Policies | A strong approach when in the process of designing software is to think about security policies to factor into the design and architecture from the beginning instead of a make-shift solution in the future. This way the program will have a natural safeguard built into it, making it safer in general. |
| 1. Keep It Simple | Simple code and simple solutions are often the best because they are easy to read and as a result, easy to maintain. Overcomplicating the code base/solution can lead to confusion when security issues arise, so it is best to keep it simple to avoid this confusion. |
| 1. Default Deny | Default deny denies requests by default. This approach assumes that all actions are a security risk and only permits those that have been predefined. By implementing it in this fashion, it provides a conservative layer of security for the application that allows for actions/permissions can be granted later. |
| 1. Adhere to the Principle of Least Privilege | Least privilege refers to the permission granted to a user within the system. It is best to be conservative with permissions, so it is ideal to set the user’s privilege to the lowest level. This promotes safety within the code and reduces the number of unauthorized actions that could compromise the system. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing of data is the process of removing the sent data from the device it was sent from. Sanitization is done to prevent an unwarranted party from recovering the data that is no longer needed but contains private information. In regards to the data being sent out, sanitization can refer to encrypting said data or following security measures to ensure the sent data is only readable to the parties involved. |
| 1. Practice Defense in Depth | Defense in Depth incorporates many layers of security to protect a system. Each tool (layer) has their niche that they are really good at defending against, but are poor defenders against other threats. When each tool is combined into one, it poses a great challenge to potential threats where the sum is greater than its parts. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance is a process that aims to assure quality in a system through various means such as following standards, guidelines, or techniques. Techniques such as failure testing, process and product quality assurance, and statistical process control are effective ways to provide quality assurance and provide confidence in the security of the software. |
| 1. Adopt a Secure Coding Standard | A set coding standard is paramount for any software project, especially in regards to security. A secure coding standard has a set of rules and best practices that make the code base clean and uniform, which will inevitably reduce the number of vulnerabilities within the software. This is because each developer is writing code in a similar style, and following up-to-date best practices that are in line with the latest security measures. |

### 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] | FLP34-C. Ensure that floating-point conversions are within range of the new type.  Freely converting between data types can result in undefined behaviors, loss of data, and inaccurate values. |

| **Noncompliant Code** |
| --- |
| The floating-point function argument is set as the value of the integer function variable. The floating-point failing to properly convert will have undefined behavior. |
| void func(float f\_a) {    int i\_a;      /\* Undefined if the integral part of f\_a cannot be represented. \*/    i\_a = f\_a; |

| **Compliant Code** |
| --- |
| The function validates that the floating-point value will properly fit within the integer variable. |
| #include <float.h>  #include <limits.h>  #include <math.h>  #include <stddef.h>  #include <stdint.h>    extern size\_t popcount(uintmax\_t); /\* See INT35-C \*/  #define PRECISION(umax\_value) popcount(umax\_value)    void func(float f\_a) {    int i\_a;      if (isnan(f\_a) ||        PRECISION(INT\_MAX) < log2f(fabsf(f\_a)) ||        (f\_a != 0.0F && fabsf(f\_a) < FLT\_MIN)) {      /\* Handle error \*/    } else {      i\_a = f\_a;    }  } |

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

| **Principles(s):** 1) Heed Compiler Warnings – C++ is statically-typed which means the compiler will throw an error when it catches an incorrect data type. The compiler will throw a message explaining what is going wrong.  2) Use Effective Quality Assurance Techniques – QA increases the chance that incorrect data type conversions are found, e.g., testing can show undefined behavior when converting float to int to char. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.TYPE.IAT | Inappropriate Assignment Type |
| LDRA tool suite | 9.7.1 | 435 S, 93 S | Partially implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_C-FLP34-a CERT\_C-FLP34-b | Avoid implicit conversions from wider to narrower floating type Avoid implicit conversions of floating point numbers from wider to narrower floating type |
| Polyspace Bug Finder | R2024a | CERT CL Rule FLP34-C | Checks for float conversion overflow |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | INT31-C. Ensure that integer conversions do not result in loss of misinterpreted data.  Conversions will lose data accuracy when done poorly. Validating before conversion will ensure that no data value is lost. |

| **Noncompliant Code** |
| --- |
| This function casts the unsigned long integer into a signed character which causes truncation. This happens when the unsigned integer is greater than the value range for a signed character. |
| #include <limits.h>    void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    sc = (signed char)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This function checks to see if the unsinged long is small enough to be store in a signed char. If it is not, the code properly handles the error. |
| #include <limits.h>    void func(void) {    unsigned long int u\_a = ULONG\_MAX;    signed char sc;    if (u\_a <= SCHAR\_MAX) {      sc = (signed char)u\_a;  /\* Cast eliminates warning \*/    } else {      /\* Handle error \*/    }  } |

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

| **Principles(s):** 1) Validate Input Data – If requiring user input, one must ensure that the input data and its stored location is the same data type. This ensures the data value is maintained through any conversions.  2) Use Effective Quality Assurance Techniques – QA can help the developer find logic errors or undefined behavior leading to weird values, e.g., testing can show if a function returns an incorrect value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | NA | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6 and 10.7 |
| Cppcheck | 2.15 | memsetValueOutOfRange | Partially implemented  The second argument to memset() cannot be represented as unsigned char |
| TrustInSoft Analyzer | 1.38 | Signed\_downcast | Exhaustively verified |
| Klocwork | 2024.3 | PORTING.CAST.SIZE | NA |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.  Buffer overflow will occur when the space for a string is not sufficient. Ensuring the string accounts for character data and the null terminator is vital. |

| **Noncompliant Code** |
| --- |
| After the loop terminates, the null character is added one spot too far, which results in an off-by-one error. |
| #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** |
| --- |
| The for loop terminates one spot sooner so that the null terminator character can be added correctly to the end of the string, resulting in a valid string. |
| #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):** 1) Validate Input Data – When asking a user for input, the string that is storing this input needs to be of adequate size. Input data that is not validated and stored in a string of inadequate length causes vulnerabilities.  2) Use Effective Quality Assurance Techniques – QA can help the developer find run-time errors like this where the allocated length is less than the provided string length. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow. Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 8.1p0 | 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. |
| Astrée | 24.04 | NA | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| LDRA tool suite | 9.7.1 | 489 S, 109 D, 66 X, 70 X, 71 X | Partially implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | FIO30-C. Exclude user input from format strings.  User input can be used to alter format strings, which makes the program vulnerable and open to attacks. |

| **Noncompliant Code** |
| --- |
| A format string is printed as the result of the code, but the print function is passed a string that contains untrusted user input. |
| #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** |
| --- |
| Altering the print function from fprintf() to fputs() outputs the message without evaluating it, skipping over the potential bad user input. |
| #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):** Adopt a Secure Coding Standard – A common exploit with SQL injections involves the use of user input that is used for a SQL statement. Having a secure coding standard eliminates this issue by setting the standard to be that no SQL statements can contain user input; a workaround can be to prompt the user to choose a premade option so the data fed to the statement is developer produced only.  Architect and Design for Security Policies – Design the program with security in mind to handle SQL injection attacks if using SQL. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 8.1p0 | IO.INJ.FMT MISC.FMT | Format string injection Format string |
| Polyspace Bug Finder | R2024a | |  |  | | --- | --- | |  | CERT C: Rule FIO30-C | | Checks for tainted string format (rule partially covered) |
| GCC | 4.3.5 | NA | Can detect violations of this rule when the -Wformat-security flag is used |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | MEM31-C. Free dynamically allocated memory when no longer needed.  Ensures that memory is handled properly so the application can run as intended without memory leaks, vulnerabilities, and high memory usage. |

| **Noncompliant Code** |
| --- |
| The allocated memory is not freed which will lead to memory leaks or vulnerabilities in longer running programs. |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    int f(void) {    char \*text\_buffer = (char \*)malloc(BUFFER\_SIZE);    if (text\_buffer == NULL) {      return -1;    }    return 0;  } |

| **Compliant Code** |
| --- |
| Here, the pointer text\_buffer is freed. This is proper memory management that prevents memory leaks and vulnerabilities from occurring. |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    int f(void) {    char \*text\_buffer = (char \*)malloc(BUFFER\_SIZE);    if (text\_buffer == NULL) {      return -1;    }      free(text\_buffer);    return 0;  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Properly handling memory is vital to the security of the system. QA will assist in the effort to ensure that this is done properly.  Adopt a Secure Coding Standard – Setting a standard gives developers a guideline for how to handle memory such as when to delete it or how to allocate. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM31 | Can detect dynamically allocated resources that are not freed |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak |
| Cppcheck | 2.15 | memleak leakReturnValNotUsed leakUnsafeArgAlloc memleakOnRealloc | Doesn't use return value of memory allocation function |
| Parasoft C/C++test | 2023.1 | CERT\_C-MEM31-a | Ensure resources are freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | MSC11-C. Incorporate diagnostic tests using assertions.  Helps validate assumptions and protect against invalid assumptions during the program. |

| **Noncompliant Code** |
| --- |
| The function asserts that dup will not equal NULL. This runtime condition check using assert can cause a process to terminate, resulting in a vulnerability. |
| 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 function features proper checking and handling of the runtime condition instead of using assert. In the event the expression evaluates to True, the program will return the proper value. |
| 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):** Architect and Design for Security Policies – Assertions help reinforce security built into the design of software by validating assumptions.  Adopt a Secure Coding Standard – Assertions are one facet of a secure coding standard. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | 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 | 2023.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | ERR51-CPP. Handle all exceptions.  Exceptions thrown need to be caught with the matching exception handler to prevent abrupt termination and potential undefined behavior. |

| **Noncompliant Code** |
| --- |
| The exceptions thrown by the function are not caught so the terminate function is called. |
| void throwing\_func() noexcept(false);    void f() {    throwing\_func();  }    int main() {    f();  } |

| **Compliant Code** |
| --- |
| Main function properly handles the exceptions thrown which allows for proper program execution. |
| 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):** Keep It Simple – Simple programs are the easiest to provide security for and using exceptions is no different. Exceptions keep the code base simple and secure.  Architect and Design for Security Policies – Exceptions included in the design help reinforce security policies.  Adopt a Secure Coding Standard – Exceptions are a secure coding standard staple. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| RuleChecker | 22.10 | main-function-catch-all early-catch-all | Partially checked |
| Polyspace Bug Finder | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Environment | [STD-008-CPP] | ENV33-C. Do not call system().  The system function creates potential vulnerabilities, mainly through execution of system commands. |

| **Noncompliant Code** |
| --- |
| The system function is called with a specific string that can be exploited to essentially tell the shell to create an account with the specified values. |
| #include <string.h>  #include <stdlib.h>  #include <stdio.h>    enum { BUFFERSIZE = 512 };    void func(const char \*input) {    char cmdbuf[BUFFERSIZE];    int len\_wanted = snprintf(cmdbuf, BUFFERSIZE,                              "any\_cmd '%s'", input);    if (len\_wanted >= BUFFERSIZE) {      /\* Handle error \*/    } else if (len\_wanted < 0) {      /\* Handle error \*/    } else if (system(cmdbuf) == -1) {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| The execve function replaces the system function which is known to not use the full shell like the system function. Because of this, the program is not susceptible to the command-injection from the noncompliant code example. |
| #include <sys/types.h>  #include <sys/wait.h>  #include <unistd.h>  #include <errno.h>  #include <stdlib.h>    void func(char \*input) {    pid\_t pid;    int status;    pid\_t ret;    char \*const args[3] = {"any\_exe", input, NULL};    char \*\*env;    extern char \*\*environ;      /\* ... Sanitize arguments ... \*/      pid = fork();    if (pid == -1) {      /\* Handle error \*/    } else if (pid != 0) {      while ((ret = waitpid(pid, &status, 0)) == -1) {        if (errno != EINTR) {          /\* Handle error \*/          break;        }      }      if ((ret == 0) ||          !(WIFEXITED(status) && !WEXITSTATUS(status))) {        /\* Report unexpected child status \*/      }    } else {      /\* ... Initialize env as a sanitized copy of environ ... \*/      if (execve("/usr/bin/any\_cmd", args, env) == -1) {        /\* Handle error \*/        \_Exit(127);      }    }  } |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – The system call is a powerful command so it must be sanitized before using because there is a risk of exploitation if not sanitized.  Default Deny – Denying by default could prevent the system call from causing damage; the caveat is it must be turned off when a regular user wants to use it. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | cert-env33-c | |  |  | | --- | --- | |  | Checked by clang-tidy | |
| CodeSonar | 8.1p0 | BADFUNC.PATH.SYSTEM IO.INJ.COMMAND | Use of system Command injection |
| Polyspace Bug Finder | R2024a | CERT C: Rule ENV33-C | Checks for unsafe call to a system function (rule fully covered) |
| |  |  | | --- | --- | | SonarQube C/C++ Plugin |  | | 3.11 | S990 | Detects uses of "abort", "exit", "getenv" and "system" from <stdlib.h> |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arrays | [STD-009-CPP] | ARR32-C. Ensure size argument for variable length arrays are in a valid range.  Not doing so will result in undefined behavior and cause vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The array size argument is valid, but it will cause problems if the size were to be 0 or too high, which is certainly possible. |
| #include <stddef.h>    extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {    int vla[size];    do\_work(vla, size);  } |

| **Compliant Code** |
| --- |
| The size of the array is limited from 1 to 1024 by the programmer. This prevents the array from being 0 or too large. |
| #include <stdint.h>  #include <stdlib.h>    enum { MAX\_ARRAY = 1024 };  extern void do\_work(int \*array, size\_t size);    void func(size\_t size) {    if (0 == size || SIZE\_MAX / sizeof(int) < size) {      /\* Handle error \*/      return;    }    if (size < MAX\_ARRAY) {      int vla[size];      do\_work(vla, size);    } else {      int \*array = (int \*)malloc(size \* sizeof(int));      if (array == NULL) {        /\* Handle error \*/      }      do\_work(array, size);      free(array);    }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Having a standard way for properly assigning variable lengths for arrays is a good way to prevent against a probable error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.15 | negativeArraySize | Context sensitive analysis Will warn only if given size is negative |
| LDRA tool suite | 9.7.1 | 621 S | Enhanced enforcement |
| PC-lint Plus | 1.4 | 9035 | Assistance provided |
| TrustInSoft Analyzer | 1.38 | Alloca\_bounds | Exhaustively verified |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory | [STD-010-CPP] | MEM30-C. Do not access freed memory.  Memory must not be access once it is freed because the behavior is undefined and can lead to exploits. |

| **Noncompliant Code** |
| --- |
| P is freed before p->next is executed so the result is that freed memory is accessed and the behavior is undefined. |
| #include <stdlib.h>    struct node {    int value;    struct node \*next;  };    void free\_list(struct node \*head) {    for (struct node \*p = head; p != NULL; p = p->next) {      free(p);    }  } |

| **Compliant Code** |
| --- |
| The expression at the end of each for loop cycle is stored temporarily so that the program can properly free memory and still properly access the next item in the linked list. |
| #include <stdlib.h>    struct node {    int value;    struct node \*next;  };    void free\_list(struct node \*head) {    struct node \*q;    for (struct node \*p = head; p != NULL; p = q) {      q = p->next;      free(p);    }  } |

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

| **Principles(s):** Keep It Simple – Accessing freed memory has dire consequences so a good way to prevent it from happening is by keeping the code simple so freed memory can be tracked and managed easily.  Adopt a Secure Coding Standard – Setting a standard for how to handle the freeing of memory is vital. There are many ways to format it so having a uniform way will reduce occurrences of this error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | dangling\_pointer\_use | Supported  Astrée reports all accesses to freed allocated memory |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses after its deallocation and double memory deallocations |
| Parasoft C/C++test | 2023.1 | CERT\_C-MEM30-a | Do not use resources that have been freed |
| PC-lint Plus | 1.4 | 449, 2434 | Fully Supported |

### 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.

There are four phases of the DevSecOps pipeline that can utilize automation tools: Build, Verify and Test, Transition and Health Check, and Monitor and Detect. The build and Verify and test phases of pre-production can utilize tools such as dependency checkers or source code analyzers such as Cppcheck or Polyspace Bug Finder to either finds potential errors in the source code or to acknowledge potentially vulnerable dependencies. On the production side with the Transition and health check and Monitor and detect phases, tools catering to penetration testing and reporting can be utilized. In this phase of the pipeline, automation tools such as LDRA tool suite can be used to automate testing of the code and to provide reports for audits and quality assurance. These tools could be seamlessly integrated into the pipeline to provide quality security automation.

### 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 | Low | P3 | L3 |
| STD-002-CPP | High | Probable | High | P6 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | Medium | Probable | Medium | P8 | L2 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | Medium | P12 | L1 |
| STD-009-CPP | High | Probable | High | P6 | L2 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### 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 | Encryption at rest protects stored information by using an encryption algorithm on the stored data, turning it into encrypted text that can only be deciphered with the hash key. This is important because it protects the sensitive data from being utilized upon being stolen because the data is encrypted. Password hashing is a common application of encryption at rest. |
| Encryption in flight | Encryption in flight encrypts data in transit, typically occurring from client to server or over a network. Data is encrypted before being sent so that if it were to be intercepted, it would be of no use to the interceptor. Upon reaching the destination, the data is then decrypted with the agreed upon hash key. |
| Encryption in use | Encryption in use is the practice of using encrypted data as if it was deciphered plain-text. When encrypted data is decrypted it poses a risk to that information so encryption in use is implemented to allow for the data to be used without the risk of decryption. This is the most secure encryption method but it is the most computationally intensive. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication confirms the identity of the user. It asks, “who are you?” and matches credentials to determine whether the user is valid or not. Authentication is applied with user logins where the user enters a username and password and these login credentials are authenticated with the username and password stored on the system. |
| Authorization | Authorization determines what a user is allowed to do. Once a user is authenticated, authorization takes over and permits what the user can and cannot do. Common actions such as files accessed by users or the level of access users are allowed is handled by authorization. A general user cannot run system administrator tasks and certain files are only allowed to be accessed by certain users are examples of authorization. |
| Accounting | Accounting tracks the resources used by users, session information, and data sent and received. Accounting is not a proactive security measure such as authentication and authorization, but it is just as important because it creates a history of all that has happened on the system or application. In the event that an issue arises, the accounting system will provide the information needed to understand what happened. Actions such as changes to the database or addition of new users are examples of useful information that will be recorded and monitored of the life 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 |  |
| 1.5 | 11/16/2024 | Coding Standards | Kevin Bristow |  |
| 2.0 | 12/07/2024 | Completed Policy | Kevin Bristow |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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