**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

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

[Coding Standard 1 4](#_3dy6vkm)

[Coding Standard 2 5](#_1t3h5sf)

[Coding Standard 3 6](#_4d34og8)

[Coding Standard 4 7](#_2s8eyo1)

[Coding Standard 5 8](#_17dp8vu)

[Coding Standard 6 9](#_3rdcrjn)

[Coding Standard 7 10](#_26in1rg)

[Coding Standard 8 11](#_lnxbz9)

[Coding Standard 9 13](#_35nkun2)

[Coding Standard 10 14](#_1ksv4uv)

[Defense-in-Depth Illustration 15](#_44sinio)

[Project One 15](#_2jxsxqh)

[1.](#_z337ya) Revise the C/C++ Standards 15

[2.](#_3j2qqm3) Risk Assessment 15

[3.](#_1y810tw) Automated Detection 15

[4.](#_4i7ojhp) Automation 15

[5.](#_2xcytpi) Summary of Risk Assessments 16

[6.](#_1ci93xb) Create Policies for Encryption and Triple A 16

[7.](#_3whwml4) Map the Principles 17

[Audit Controls and Management 18](#_2bn6wsx)

[Enforcement 18](#_qsh70q)

[Exceptions Process 18](#_3as4poj)

[Distribution 19](#_1pxezwc)

[Policy Change Control 19](#_49x2ik5)

[Policy Version History 19](#_2p2csry)

[Appendix A Lookups 19](#_147n2zr)

[Approved C/C++ Language Acronyms 19](#_3o7alnk)

## 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 | Input validation is crucial for preventing malicious input from compromising the application. By verifying that input meets the expected format, type, and length before being processed, developers can safeguard against issues like buffer overflows, SQL injection, and cross-site scripting (XSS) attacks. |
| 1. Heed Compiler Warnings | Compiler warnings are signals of potential issues in code that could lead to vulnerabilities or performance problems. Ignoring these warnings may introduce security flaws. Developers should address warnings promptly to maintain secure and optimized code. |
| 1. Architect and Design for Security Policies | Secure software design starts with defining strong security policies that govern how the application will handle user data, authentication, and access control. By incorporating security principles into the architecture, developers can proactively identify and mitigate security risks. |
| 1. Keep It Simple | Simple code is easier to understand, maintain, and secure. By avoiding overly complex systems, developers reduce the risk of introducing vulnerabilities through misunderstandings or overlooked edge cases. Simple designs are also easier to audit for security flaws. |
| 1. Default Deny | The default deny principle means denying access unless explicitly granted. This approach minimizes unnecessary access rights and restricts user privileges to only what's necessary, reducing the potential attack surface and limiting the impact of any potential security breach. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege ensures that users, applications, and systems only have the minimum level of access necessary to perform their tasks. By minimizing unnecessary permissions, the damage from potential security breaches can be contained and mitigated. |
| 1. Sanitize Data Sent to Other Systems | When sending data to other systems, ensure that it is sanitized to remove any malicious or unsafe content. This prevents attacks like SQL injection, command injection, or cross-site scripting (XSS) from exploiting vulnerabilities in external systems. |
| 1. Practice Defense in Depth | Defense in depth is a layered security strategy where multiple security controls are implemented at different levels of the application. This approach ensures that if one layer fails, others will still protect the system, reducing the likelihood of a successful attack. |
| 1. Use Effective Quality Assurance Techniques | Robust quality assurance practices, such as automated testing, code reviews, and vulnerability scanning, help identify and fix security issues early in the development process. Regular testing and validation ensure the application is secure before deployment. |
| 1. Adopt a Secure Coding Standard | A secure coding standard outlines the best practices for writing code that minimizes vulnerabilities. By adopting these standards, developers can avoid common mistakes, implement secure coding practices consistently, and create safer software applications. |

### 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** | DCL52-CPP | DCL52-CPP advises against using const or volatile with reference types because references are already non-rebindable. Applying const or volatile to the reference doesn't add any useful behavior and can cause confusion. Instead, focus on using const or volatile for the object being referred to, not the reference. |

| **Noncompliant Code** |
| --- |
| In this example, a const qualifier is incorrectly applied to a reference type, which results in undefined behavior. C++ does not allow references to be const-qualified, which may lead to unexpected behavior if the code compiles without a fatal error. |
| #include <iostream>    void f(char c) {  char &const p = c;  p = 'p';  std::cout << c << std::endl;  } |

| **Compliant Code** |
| --- |
| This example correctly declares a reference to a char without using the const qualifier on the reference type. The value of p can now be modified as expected. |
| #include <iostream>    void f(char c) {  char &p = c;  p = 'p';  std::cout << c << std::endl;  } |

| **Principles(s):** Ensures that code is clear and easy to maintain by avoiding unnecessary qualifiers on reference types. |
| --- |

**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++-DCL52 |  |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | C++0014 |  |
| Klocwork | 2024.4 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |
| Parasoft C/C++test | 2024.2 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | MEM50-CPP | MEM50-CPP advises against hardcoding data values directly into the code, as this can make maintenance difficult and introduce risks. Hardcoded values can create problems when updates or changes are needed, as the values would need to be manually changed in multiple places. Instead, using variables, constants, or configuration files makes the code more flexible, readable, and easier to maintain, enhancing both security and long-term sustainability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, s is dereferenced after it has been deallocated. If this access results in a write-after-free, the vulnerability can be exploited to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Principles(s):** Promotes flexibility and security by avoiding hardcoded values, making code easier to maintain and update. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.3p0 | ALLOC.UAF | Use after free |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR50-CPP | STR50-CPP emphasizes the importance of ensuring that strings have enough memory allocated to store both the character data and the null terminator ('\0'). Failing to do so can result in buffer overflows, leading to memory corruption, crashes, or vulnerabilities that attackers can exploit. By guaranteeing sufficient space for both the string content and its null terminator, this rule helps maintain program stability and security. |

| **Noncompliant Code** |
| --- |
| To solve this problem, it may be tempting to use the std::ios\_base::width() method, but there still is a trap, as shown in this noncompliant code example. |
| #include <iostream>    void f() {  char bufOne[12];  char bufTwo[12];  std::cin.width(12);  std::cin >> bufOne;  std::cin >> bufTwo;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

| **Principles(s):** Prevents buffer overflows and ensures stability by allocating sufficient memory for strings, including the null terminator. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| CodeSonar | 8.3p0 | MISC.MEM.NTERM | No space for null terminator |
| Helix QAC | 2024.4 | C++5216 |  |
| Klocwork | 2024.4 | NNTS.MIGHT |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | FIO30-C | FIO30-C advises excluding user input from format strings to prevent SQL injection vulnerabilities. SQL injection occurs when user input is improperly included in SQL queries, allowing attackers to manipulate the query and potentially access, modify, or delete sensitive data. By sanitizing user input and excluding it from format strings, this risk is mitigated, ensuring that only valid input is processed and preventing unauthorized access or data manipulation. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect. The function accepts the name of the user as a string referenced by user. This is an exemplar of untrusted data that originates from an unauthenticated user. The function constructs an error message that is then output to stderr using the C Standard 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** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs msg directly to stderr without evaluating its contents: |
| #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);  } |

| **Principles(s):** Reduces the risk of SQL injection by preventing user input from being directly included in format strings. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 8.3p0 | IO.INJ.FMT | Format string injection  Format string |
| Compass/ROSE |  |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM31-C | Free dynamically allocated memory when no longer needed. This is critical for memory protection because failing to free memory can lead to memory leaks, where memory is consumed without being properly released, reducing system performance or causing crashes. Properly managing memory by freeing it when it's no longer needed ensures that the system remains efficient and protected from resource depletion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object: |
| #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** |
| --- |
| In this compliant solution, the pointer is deallocated with a call to free(): |
| #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;  } |

| **Principles(s):** Enhances memory protection and system efficiency by freeing dynamically allocated memory when it's no longer needed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported, but no explicit checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM31 | Can detect dynamically allocated resources that are not freed |
| CodeSonar | 8.3p0 | ALLOC.LEAK | Leak |
| Coverity | 2017.07 | RESOURCE\_LEAK | Finds resource leaks from variables that go out of scope while owning a resource |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | ERR33-C | Detect and handle standard library errors relates to assertions because both help identify problems in a program. Assertions check conditions during development, while error handling deals with issues that arise during execution. Both ensure the program runs correctly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function utf8\_to\_wcs() attempts to convert a sequence of UTF-8 characters to wide characters. It first invokes setlocale() to set the global locale to the implementation-defined en\_US.UTF-8 but does not check for failure. The setlocale() function will fail by returning a null pointer, for example, when the locale is not installed. The function may fail for other reasons as well, such as the lack of resources. Depending on the sequence of characters pointed to by utf8, the subsequent call to mbstowcs() may fail or result in the function storing an unexpected sequence of wide characters in the supplied buffer wcs. |
| #include <locale.h>  #include <stdlib.h>    int utf8\_to\_wcs(wchar\_t \*wcs, size\_t n, const char \*utf8,  size\_t \*size) {  if (NULL == size) {  return -1;  }  setlocale(LC\_CTYPE, "en\_US.UTF-8");  \*size = mbstowcs(wcs, utf8, n);  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution checks the value returned by setlocale() and avoids calling mbstowcs() if the function fails. The function also takes care to restore the locale to its initial setting before returning control to the caller. |
| #include <locale.h>  #include <stdlib.h>    int utf8\_to\_wcs(wchar\_t \*wcs, size\_t n, const char \*utf8,  size\_t \*size) {  if (NULL == size) {  return -1;  }  const char \*save = setlocale(LC\_CTYPE, "en\_US.UTF-8");  if (NULL == save) {  return -1;  }    \*size = mbstowcs(wcs, utf8, n);  if (NULL == setlocale(LC\_CTYPE, save)) {  return -1;  }  return 0;  } |

| **Principles(s):** Improves program reliability by detecting and handling standard library errors, ensuring correct execution. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | error-information-unused | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-ERR33 |  |
| CodeSonar | 8.3p0 | LANG.FUNCS.IRV | Ignored return value  Missing Test of Error Code  Non-zero Error Code |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.8 | Implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR30-C | In relation to exceptions, errno is only meaningful after a function sets it to indicate an error. Like exceptions, you should only check errno after a function call that may have failed, and you shouldn't rely on its value between function calls, as it can be overwritten. Just as exceptions should be handled in a clear context, errno should only be inspected when its value is set by a function signaling an error. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example fails to set errno to 0 before invoking strtoul(). If an error occurs, strtoul() returns a valid value (ULONG\_MAX), so errno is the only means of determining if strtoul() ran successfully. |
| #include <errno.h>  #include <limits.h>  #include <stdlib.h>    void func(const char \*c\_str) {  unsigned long number;  char \*endptr;    number = strtoul(c\_str, &endptr, 0);  if (endptr == c\_str || (number == ULONG\_MAX  && errno == ERANGE)) {  /\* Handle error \*/  } else {  /\* Computation succeeded \*/  }  } |

| **Compliant Code** |
| --- |
| This compliant solution sets errno to 0 before the call to strtoul() and inspects errno after the call: |
| #include <errno.h>  #include <limits.h>  #include <stdlib.h>    void func(const char \*c\_str) {  unsigned long number;  char \*endptr;    errno = 0;  number = strtoul(c\_str, &endptr, 0);  if (endptr == c\_str || (number == ULONG\_MAX  && errno == ERANGE)) {  /\* Handle error \*/  } else {  /\* Computation succeeded \*/  }  } |

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

| **Principles(s):** ERR30-C ensures errno is only checked after a function call that sets it, preventing reliance on its value between calls. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | errno-reset | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-ERR30 | Fully implemented |
| CodeSonar | 8.3p0 | LANG.STRUCT.RC | Redundant Condition |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.8 | Implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Management** | MEM31-C | Free dynamically allocated memory when no longer needed. Proper memory management is crucial to avoid memory leaks, where memory is reserved but not properly released. By ensuring that memory is freed after use, this standard helps prevent the program from consuming more memory than necessary, improving efficiency and stability. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object: |
| #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** |
| --- |
| In this compliant solution, the pointer is deallocated with a call to free(): |
| #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;  } |

| **Principles(s):** MEM31-C ensures that dynamically allocated memory is freed when no longer needed to prevent memory leaks and improve efficiency. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported, but no explicit checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM31 | Can detect dynamically allocated resources that are not freed |
| CodeSonar | 8.3p0 | ALLOC.LEAK | Leak |
| Coverity | 2017.07] | RESOURCE\_LEAK | Finds resource leaks from variables that go out of scope while owning a resource |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Environment** | ENV30-C | Do not modify the object referenced by the return value of certain functions . This standard helps ensure that the program interacts safely with system-level resources, preventing issues that could arise from unintended changes to these objects. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example modifies the string returned by getenv() by replacing all double quotation marks (") with underscores (\_): |
| #include <stdlib.h>    void trstr(char \*c\_str, char orig, char rep) {  while (\*c\_str != '\0') {  if (\*c\_str == orig) {  \*c\_str = rep;  }  ++c\_str;  }  }    void func(void) {  char \*env = getenv("TEST\_ENV");  if (env == NULL) {  /\* Handle error \*/  }  trstr(env,'"', '\_');  } |

| **Compliant Code** |
| --- |
| If the programmer does not intend to modify the environment, this compliant solution demonstrates how to modify a copy of the return value: |
| #include <stdlib.h>  #include <string.h>    void trstr(char \*c\_str, char orig, char rep) {  while (\*c\_str != '\0') {  if (\*c\_str == orig) {  \*c\_str = rep;  }  ++c\_str;  }  }    void func(void) {  const char \*env;  char \*copy\_of\_env;    env = getenv("TEST\_ENV");  if (env == NULL) {  /\* Handle error \*/  }    copy\_of\_env = (char \*)malloc(strlen(env) + 1);  if (copy\_of\_env == NULL) {  /\* Handle error \*/  }    strcpy(copy\_of\_env, env);  trstr(copy\_of\_env,'"', '\_');  /\* ... \*/  free(copy\_of\_env);  } |

| **Principles(s):** Minimize side effects and preserve object integrity by not modifying return values directly, ensuring stable and predictable behavior in system interactions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | stdlib-const-pointer-assign | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-ENV30 |  |
| CodeSonar | 8.3p0 | LANG.STRUCT.RPNTC | Use of getenv  Returned Pointer Not Treated as const |
| Cppcheck Premium | 24.11.0 | premium-cert-env30-c | [Insert text.] |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Preprocessor** | PRE30-C | When creating universal character names (UCNs), the preprocessor can perform string concatenation, but doing so can lead to issues where the intended character names are incorrectly formed or misinterpreted. The preprocessor might not properly handle concatenated strings that are meant to represent UCNs, leading to unexpected behavior or incorrect character mappings. To avoid such issues, it's important to use properly defined, non-concatenated UCNs in macros or constants to ensure the correct handling of character data during preprocessing. |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because it produces a universal character name by token concatenation: |
| #define assign(uc1, uc2, val) uc1##uc2 = val    void func(void) {  int \u0401;  /\* ... \*/  assign(\u04, 01, 4);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a universal character name but does not create it by using token concatenation: |
| #define assign(ucn, val) ucn = val    void func(void) {  int \u0401;  /\* ... \*/  assign(\u0401, 4);  /\* ... \*/  } |

| **Principles(s):** Ensure clarity and correctness by avoiding preprocessor concatenation of universal character names to prevent misinterpretation and ensure accurate character handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | universal-character-name-concatenation | Fully implemented |
| Axivion Bauhaus Suite | 7.2.0 | CertC-PRE30 | Fully implemented |
| CodeSonar | 8.3p0 | LANG.PREPROC.PASTE | Macro uses ## operator |
| Cppcheck | 2.15 | preprocessorErrorDirective |  |

### Defense-in-Depth Illustration

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



## Project One

### Automation



To automate the enforcement of the standards defined in the policy, Green Pace can enhance its existing DevOps process by integrating security throughout the DevSecOps lifecycle. In the Pre-Production phase, automation can be introduced to assess threats, enforce security-driven design principles, and conduct automated vulnerability scans during development and testing. In the Production phase, automated security configurations can be applied during deployment, while real-time monitoring tools such as SIEM can detect security incidents. Additionally, automated responses can be set up to block attacks, isolate compromised services, and restore the system to a secure state. This approach ensures that security is continuously embedded in both development and operational processes, reducing vulnerabilities and enabling a swift response to potential threats.

### 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 |
| DCL52-CPP | Low | Unlikely | Low | P3 | L3 |
| MEM50-CPP | High | Likely | Medium | P18 | L1 |
| STR50-CPP | High | Likely | Medium | P18 | L1 |
| FIO30-C | High | Likely | Medium | P18 | L1 |
| MEM31-C | Medium | Probable | Medium | P8 | L2 |
| ERR33-C | High | Likely | Medium | P18 | L1 |
| ERR30-C | Medium | Probable | Medium | P8 | L2 |
| MEM31-C | Medium | Probable | Medium | P8 | L2 |
| ENV30-C | Low | Probable | Medium | P3 | L3 |
| PRE30-C | Low | Unlikely | Medium | P3 | L3 |

### 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 | All stored sensitive data must be encrypted using AES-256 to prevent unauthorized access in case of theft or breach. |
| Encryption in flight | Data transmitted over networks must use secure protocols (TLS 1.2/1.3, SSH) to protect against interception and eavesdropping. |
| Encryption in use | Sensitive data being processed in memory must be protected using secure enclaves or hardware encryption to prevent exposure from attacks. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Users must authenticate via unique credentials and multi-factor authentication (MFA) to prevent unauthorized access. |
| Authorization | Access to systems and data must be restricted based on role-based access control (RBAC) to ensure users only have necessary permissions. |
| Accounting | All user logins, database changes, file accesses, and new user additions must be logged and monitored for security and compliance. |

### Map the Principles

| **Standard** | **Mapped Principle(s)** | **Justification** |
| --- | --- | --- |
| Encryption at Rest | 3, 5, 6, 8 | Secure design (3) ensures encryption is properly implemented, default deny (5) restricts unauthorized access, least privilege (6) limits who can decrypt data, and defense in depth (8) ensures multiple security layers. |
| Encryption in Flight | 1, 3, 5, 8 | Validating input (1) ensures transmitted data is secure, security architecture (3) mandates secure transmission protocols, default deny (5) restricts unauthorized network access, and defense in depth (8) ensures multiple encryption mechanisms. |
| Encryption in Use | 3, 6, 8 | Secure design (3) ensures encryption is active while processing data, least privilege (6) limits exposure, and defense in depth (8) ensures multiple protection layers. |
| Authentication | 3, 5, 6, 8, 10 | Secure design (3) ensures strong authentication policies, default deny (5) prevents unauthorized logins, least privilege (6) limits access, defense in depth (8) implements multiple authentication methods, and secure coding (10) ensures strong password and token handling. |
| Authorization | 3, 5, 6, 9, 10 | Secure architecture (3) ensures access control, default deny (5) blocks unauthorized access, least privilege (6) restricts user permissions, quality assurance (9) ensures security testing, and secure coding (10) enforces best access control practices. |
| Accounting | 3, 6, 8, 9, 10 | Secure architecture (3) ensures detailed logging, least privilege (6) restricts log access, defense in depth (8) provides multiple layers of tracking, quality assurance (9) ensures audit trails work correctly, and secure coding (10) follows logging best practices. |

## 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 | 01/23/2025 | Updated to project one | Jon Wickerd |  |
| 1.2 | 02/12/2025 | project one | Jon Wickerd |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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