**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 | Validating input data from all sources is essential in preventing software vulnerabilities and security breaches. This is particularly important when dealing with untrusted or external data sources. It is crucial to monitor and validate incoming data from entry points such as command line arguments, network interfaces, user-controlled files, and environmental variables. |
| 1. Heed Compiler Warnings | Compiler warnings are implemented to mitigate security risks in code. By combining static and dynamic analysis tools with compiler checks and manual code reviews, security flaws can be identified and minimized. |
| 1. Architect and Design for Security Policies | Developing a robust software architecture and implementation design that adheres to the highest safety standards is crucial. The architecture should ensure appropriate security at each level of authority, with distinct permissions and resource access granted based on the authentication tier, as an example. |
| 1. Keep It Simple | As the saying goes, the simpler the program or software, the easier it is to secure. It should be kept as simple as possible while meeting the client's needs and requirements with maximum efficiency. The more complex the software, the more challenging it becomes to maintain its security and ensure its integrity at the highest level. |
| 1. Default Deny | The philosophy of access decisions is based on granting permissions rather than restricting access. By default, all users are denied access unless the security model explicitly grants them permission. |
| 1. Adhere to the Principle of Least Privilege | Each user and process should receive only the minimum permissions necessary to complete their tasks. By default, no permissions are granted. As required functions are performed, permissions are gradually granted based on ascending levels of authority. |
| 1. Sanitize Data Sent to Other Systems | All data transferred to other systems must be properly sanitized. This includes systems such as command shells, relational databases, and commercial off-the-shelf components. To mitigate the security risks posed by unused or improperly handled security and data within these systems, sanitizing the data beforehand helps reduce the potential for security breaches. |
| 1. Practice Defense in Depth | The practice of employing multiple layers of defense, each with a specific focus, creates a multi-layered security strategy that protects against various types of threats. This approach ensures comprehensive protection, as any weaknesses in one layer are addressed by the other layers, preventing issues from compromising the system. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques help prevent potential security compromises by identifying issues before they occur. Tests such as fuzz testing, penetration testing, source code audits, and JUnit testing should be integrated into the development cycle and quality assurance processes. Additionally, involving external reviewers with a critical mindset ensures they approach the review with the expectation of finding flaws, free from bias, which helps ensure nothing is overlooked. |
| 1. Adopt a Secure Coding Standard | Establishing and following a secure coding standard enables the efficient and safe development of programs and software. A universal standard is easy to assess, review, and compare against current, past, and future product releases, ensuring that the safest and most efficient product is produced. |

### 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] | Ensure that unsigned integer operations do not wrap |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable vulnerability. |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap: |
| #include <limits.h>    **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum;  **if** (UINT\_MAX - ui\_a < ui\_b) {      /\* Handle error \*/    } **else** {      usum = ui\_a + ui\_b;    }    /\* ... \*/  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | Integer-overflow | Fully Checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 8.1p0 | 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 |
| Coverity | 2017.07 | INTEGER-OVERFLOW | Implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Close files when they are no longer needed |

| **Noncompliant Code** |
| --- |
| This code example is noncompliant because the file opened by the call to fopen() is not closed before function func() returns: |
| #include <stdio.h>    int func(const char \*filename) {  FILE \*f = fopen(filename, "r");  if (NULL == f) {  return -1;  }  /\* ... \*/  return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the file pointed to by f is closed before returning to the caller: |
| #include <stdio.h>    int func(const char \*filename) {  FILE \*f = fopen(filename, "r");  if (NULL == f) {  return -1;  }  /\* ... \*/  if (fclose(f) == EOF) {  return -1;  }  return 0;  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | N/A | Supported, but no explicit checker |
| CodeSonar | 8.1p0 | ALLOC.LEAK | Leak |
| Coverity | 2017.07 | RESOURCE\_LEAK (partial) | Partially Implemented |
| Cppcheck | 2.15 | resourceLeak | N/A |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| This noncompliant code example demonstrates an off-by-one error [Dowd 2006]. The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte 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 compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to 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 - 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 - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. Practice Defense in Depth - Practicing defense in depth helps mitigate malicious data effectively, as inputs like characters and strings are common entry points for attacks. |
| --- |

**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 |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Klockwork | 2024.4 | SV.FMT\_STR.BAD\_SCAN\_FORMAT  SV.UNBOUND\_STRING\_INPUT.FUNC | N/A |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Exclude user input from format strings |

| **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);  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended.  Practice Defense in Depth - Practicing defense in depth helps mitigate malicious data effectively, as inputs like characters and strings are common entry points for attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | N/A | Supported via stubbing/taint analysis |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 8.1p0 | IO.INJ.FMT  MISC.FMT | MISC.FMT  Format string injection  Format string |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Free dynamically allocated memory when no longer needed |

| **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;  } |

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

| **Principles(s):** Heed compiler warnings - Paying attention to compiler warnings and employing effective quality assurance techniques can aid in proper memory management by helping to test for correct memory usage, clearing, and allocation/deallocation within the program. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | N/A | Supported, but no explicit checker |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM31 | Can detect dynamically allocated resources that are not freed |
| Coverity | 2017.07 | RESOURCE\_LEAK  ALLOC\_FREE\_MISMATCH | Finds resource leaks from variables that go out of scope while owning a resource |
| Cppcheck | 2.15 | memleak  leakReturnValNotUsed  leakUnsafeArgAlloc  memleakOnRealloc | N/A |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do not form or use out-of-bounds pointers or array subscripts |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the function f() attempts to validate the index before using it as an offset to the statically allocated table of integers. However, the function fails to reject negative index values. When index is less than zero, the behavior of the addition expression in the return statement of the function is undefined behavior 46. On some implementations, the addition alone can trigger a hardware trap. On other implementations, the addition may produce a result that when dereferenced triggers a hardware trap. Other implementations still may produce a dereferenceable pointer that points to an object distinct from table. Using such a pointer to access the object may lead to information exposure or cause the wrong object to be modified. |
| enum { TABLESIZE = 100 };    static int table[TABLESIZE];    int \*f(int index) {  if (index < TABLESIZE) {  return table + index;  }  return NULL;  } |

| **Compliant Code** |
| --- |
| One compliant solution is to detect and reject invalid values of index if using them in pointer arithmetic would result in an invalid pointer: |
| enum { TABLESIZE = 100 };    static int table[TABLESIZE];    int \*f(int index) {  if (index >= 0 && index < TABLESIZE) {  return table + index;  }  return NULL;  } |

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

| **Principles(s):** Heed compiler warnings - Paying attention to compiler warnings and employing effective quality assurance techniques can aid in proper memory management by helping to test for correct memory usage, clearing, and allocation/deallocation within the program. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | array-index-range  array-index-range-constant  null-dereferencing  pointered-deallocation  return-reference-local | Partially checked  Can detect all accesses to invalid pointers as well as array index out-of-bounds accesses and prove their absence.  This rule is only partially checked as invalid but unused pointers may not be reported. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-ARR30 | Can detect out-of-bound access to array / buffer |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU  LANG.STRUCT.PARITH  LANG.STRUCT.PBB  LANG.STRUCT.PPE  BADFUNC.BO.\* | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun  Pointer Arithmetic  Pointer before beginning of object  Pointer past end of object  A collection of warning classes that report uses of library functions prone to internal buffer overflows. |
| Coverity | 2017.07 | OVERRUN  NEGATIVE\_RETURNS  ARRAY\_VS\_SINGLETON  BUFFER\_SIZE | Can detect the access of memory past the end of a memory buffer/array  Can detect when the loop bound may become negative  Can detect the out-of-bound read/write to array allocated statically or dynamically  Can detect buffer overflows |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Detect and handle standard library errors |

| **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;  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | error-information-unused  error-information-unused-computed | Partially checked |
| CodeSonar | 8.1p0 | LANG.FUNCS.IRV  LANG.ERRCODE.NOTEST  LANG.ERRCODE.NZ | Ignored return value  Missing Test of Error Code  Non-zero Error Code |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.8  MISRA C 2012 Rule 22.9  MISRA C 2012 Rule 22.10 | Implemented |
| LDRA tool suite | 9.7.1 | 80 D | Partially implemented |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Cryptography Coding | [STD-008-CPP] | Never hard code sensitive information |

| **Noncompliant Code** |
| --- |
| This noncompliant code example must authenticate to a remote service with a code, using the authenticate() function declared below. It passes the authentication code to this function as a string literal. |
| /\* 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 compliant solution requires the user to supply the authentication code, and securely erases it when done, using memset\_s(), an optional function provided by C11's Annex K. |
| /\* 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):** Adopt a Secure Coding Standard - Emphasizes the importance of consistent, secure practices when implementing cryptographic algorithms, libraries, and protocols. Cryptography plays a critical role in ensuring data confidentiality, integrity, and authenticity, and adhering to a secure coding standard helps achieve these goals efficiently and reliably. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | 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 | N/A |
| Klocwork | 2024.4 | HCC  HCC.PWD  HCC.USER  CXX.SV.PWD.PLAIN  CXX.SV.PWD.PLAIN.LENGTH  CXX.SV.PWD.PLAIN.ZERO | N/A |
| Parasoft C/C++test | 2023.1 | CERT\_C-MSC41-a | Do not hard code string literals |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Performance and Resource Efficiency | [STD-009-CPP] | Allocate sufficient memory for an object |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, inadequate space is allocated for a struct tm object because the size of the pointer is being used to determine the size of the pointed-to object: |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the correct amount of memory is allocated for the struct tm object. When allocating space for a single object, passing the (dereferenced) pointer type to the sizeof operator is a simple way to allocate sufficient memory. Because the sizeof operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior. |
| #include <stdlib.h>  #include <time.h>    struct tm \*make\_tm(int year, int mon, int day, int hour,  int min, int sec) {  struct tm \*tmb;  tmb = (struct tm \*)malloc(sizeof(\*tmb));  if (tmb == NULL) {  return NULL;  }  \*tmb = (struct tm) {  .tm\_sec = sec, .tm\_min = min, .tm\_hour = hour,  .tm\_mday = day, .tm\_mon = mon, .tm\_year = year  };  return tmb;  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | malloc-size-insufficient | Partially checked  Besides direct rule violations, all undefined behaviour resulting from invalid memory accesses is reported by Astrée. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM35 | N/A |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  ALLOC.SIZE.TRUNC  IO.TAINT.SIZE  MISC.MEM.SIZE.BAD  LANG.MEM.BO  LANG.MEM.BU  LANG.STRUCT.PARITH  LANG.STRUCT.PBB  LANG.STRUCT.PPE  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Addition overflow of allocation size  Addition overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Truncation of allocation size  Tainted allocation size  Unreasonable size argument  Buffer Overrun  Buffer Underrun  Pointer Arithmetic  Pointer Before Beginning of Object  Pointer Past End of Object  Tainted Buffer Access  Type Overrun  Type Underrun |
| Coverity | 2017.07 | BAD\_ALLOC\_STRLEN  SIZECHECK (deprecated) | Partially implemented  Can find instances where string length is miscalculated (length calculated may be one less than intended) for memory allocation purposes. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary  Finds memory allocations that are assigned to a pointer that reference objects larger than the allocated block |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Access Control and Authorization | [STD-010-CPP] | Declare objects shared between threads with appropriate storage durations |

| **Noncompliant Code** |
| --- |
| This noncompliant code example passes the address of a variable to a child thread, which prints it out. The variable has automatic storage duration. Depending on the execution order, the child thread might reference the variable after the variable's lifetime in the parent thread. This would cause the child thread to access an invalid memory location. |
| #include <threads.h>  #include <stdio.h>    int child\_thread(void \*val) {  int \*res = (int \*)val;  printf("Result: %d\n", \*res);  return 0;  }    void create\_thread(thrd\_t \*tid) {  int val = 1;  if (thrd\_success != thrd\_create(tid, child\_thread, &val)) {  /\* Handle error \*/  }  }    int main(void) {  thrd\_t tid;  create\_thread(&tid);    if (thrd\_success != thrd\_join(tid, NULL)) {  /\* Handle error \*/  }  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution stores the value in an object having static storage duration. The lifetime of this object is the entire execution of the program; consequently, it can be safely accessed by any thread. |
| #include <threads.h>  #include <stdio.h>    int child\_thread(void \*v) {  int \*result = (int \*)v;  printf("Result: %d\n", \*result); /\* Correctly prints 1 \*/  return 0;  }    void create\_thread(thrd\_t \*tid) {  static int val = 1;  if (thrd\_success != thrd\_create(tid, child\_thread, &val)) {  /\* Handle error \*/  }  }    int main(void) {  thrd\_t tid;  create\_thread(&tid);  if (thrd\_success != thrd\_join(tid, NULL)) {  /\* Handle error \*/  }  return 0;  } |

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

| **Principles(s):** Validate Input Data - Validating input data is essential, as ensuring the correct data type is used in the process allows the program to function as intended. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | N/A | Supported, resulting undefined behavior is reported by the runtime error analysis |
| CodeSonar | 8.1p0 | CONCURRENCY.LOCALARG  CONCURRENCY.C\_THREAD.ISD | Local Variable Passed to Thread  Inappropriate Storage Duration |
| Cppcheck Premium | 24.11.0 | premium-cert-con34-c | N/A |
| Helix QAC | 2024.4 | DF4926, DF4927, DF4928 | N/A |

### 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.

*Automation can be effectively implemented in both the build and monitor-and-detect sections of the diagram above. In the build phase, automating simple or repetitive manual processes can significantly reduce development time. For monitoring and detection, automated tests and processes can be utilized to identify common issues, provide basic diagnostic tools, and streamline troubleshooting and error management.*

### 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 | Likely | High | P9 | L2 |
| STD-002-CPP | Medium | Unlikely | Medium | P4 | L3 |
| 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 | High | Likely | High | P9 | L2 |
| STD-007-CPP | High | Likely | Medium | P18 | L1 |
| STD-008-CPP | High | Probably | Medium | P12 | L1 |
| STD-009-CPP | High | Probable | High | P6 | L2 |
| Medium | Medium | Probable | High | P4 | 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 | Data at rest will be encrypted using full-disk encryption at the server level, while server-wide encryption policies will be applied at the database level. Additionally, both will be supported by robust backup strategies to ensure recovery in the event of a major failure. |
| Encryption in flight | Data in transit will be encrypted using public key infrastructure (PKI) to secure incoming and outgoing exchanges, including messages, attachments, file transfers, web traffic, and other forms of communication. Additionally, SSH, password protection, and restricted communication channels will be employed to prevent malicious activity. |
| Encryption in use | Data in use will be protected through identity security protocols, ensuring access is granted only to authorized individuals or entities once all identification criteria are met. The principle of least privilege will be enforced, allowing access solely for necessary actions and functions. Additionally, a multi-tiered approval system for access change requests will be implemented to thoroughly review and verify access levels, maintaining the security of sensitive data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | The process of verifying a user's identity, typically through a username and password, determines whether the user is granted access to the system. This ensures that only authorized personnel can access the system. |
| Authorization | Authorization grants authenticated users access based on a least-privilege model, controlling the level of mobility and authority they have within the system. It also facilitates the addition of new users, modification of access levels, or complete removal of access. |
| Accounting | Accounting involves tracking user interactions with the system, including monitoring the files accessed by users and any changes made to the database through user actions. |

**\***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 | 11/10/2024 | First Edit | Chad Nadeau |  |
| 1.2 | 12/15/2024 | Second Edit | Chad Nadeau |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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