**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 | Input validation is important to prevent malicious and malformed data inputs from entering the database. This is the prime defense for injection attacks and if succeeded, can cause leaks in data. Validating data can be ensuring the correct data types are used or scanning to see if data is formatted to not contain dangerous characters or strings such as “ ` ” or 1 = 1 |
| 1. Heed Compiler Warnings | Understand the warnings and suggestions from compilers. They are not only very helpful in debugging, but offer suggestions on vulnerabilities that might be present in the code. Try to take all of the warnings into account and fix as much as possible. |
| 1. Architect and Design for Security Policies | Design and build software based on proper security guidelines. This will keep the software security much more consistent. Along with that, it helps uphold and maintain the security principles and practices. |
| 1. Keep It Simple | Keep software designs simple and minimal. More complex systems are prone to encountering errors and have a much more difficult time debugging and locating the source of error. By keeping it simple, we allow the system to be easier to understand and maintain. |
| 1. Default Deny | Permissions should automatically be defaulted to deny. This is to limit access to only those who should have access. Therefore, it is based on giving permission rather than excluding. |
| 1. Adhere to the Principle of Least Privilege | Any of the processes should be done with the minimal number of permissions. This is to prevent freely enabling permissions when it is not necessary. Every individual with access to those permissions is prone to being attacked by a hacker. |
| 1. Sanitize Data Sent to Other Systems | When data is sent out, it must be cleaned to prevent any residual functions from being used later. Attackers can capitalize this through SQL commands or injection attacks. Important to prevent this as it can be spread and infect many systems. |
| 1. Practice Defense in Depth | Defense in depth is a practice which defense is build up through many repetitive layers. This is so that if one layer becomes compromised, there will be another layer that will catch it. Through this practice, we can significantly slow down exploits and allow time to fix it. |
| 1. Use Effective Quality Assurance Techniques | Security systems must be efficiently tested to ensure that security is up to standards. By testing our security systems, we allow us the chance to catch and patch exploits before attackers do. |
| 1. Adopt a Secure Coding Standard | Create standards that is followed by all contributors to the software. This creates consistency within the system. With everyone following the same standards, we can develop stronger security. Along with that, debugging will be significantly easier as principles will be more organized. |

### 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** | **Include the appropriate type information in function declarators** |
| --- | --- | --- |
| **Data Type** | [STD-001-cpp] | Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. When using standard library calls, the easiest (and preferred) way to obtain function declarators with appropriate type information is to include the appropriate header file. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the identifier-list form for parameter declarations: |
| **int** max(a, b)  **int** a, b;  {  **return** a > b ? a : b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, int is the type specifier, max(int a, int b) is the function declarator, and the block within the curly braces is the function body: |
| **int** max(**int** a, **int** b) {  **return** a > b ? a : b;  } |

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

| **Principles(s):**  1) ValidateInput Data: This is to ensure that data sent in is of the correct type. Ex: making sure an integer is passed in to perform operations as opposed to a string.  8) Practice Defense in Depth: Simple form of defense that builds up defensive layers |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description** |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **function-prototype**  **implicit-function-declaration** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL07** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **LANG.FUNCS.PROT LANG.STRUCT.DECL.IMPT** | Incomplete function prototype Implicit Type |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL07** | Fully implemented |
| [GCC](https://wiki.sei.cmu.edu/confluence/display/c/GCC) | 4.3.5 |  | Can detect violation of this recommendation when the -Wstrict-prototypes flag is used |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **C1304, C2050, C3331, C3335, C3408, C3450** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **MISRA.FUNC.PROT\_FORM.KR.2012** **MISRA.FUNC.NOPROT.DEF** **MISRA.CAST.FUNC\_PTR.2012** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **21 S** **135 S** **170 S** | Fully implemented |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **718, 746, 936, 9074** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rec. DCL07-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.dcl07c.html) | Checks for:   * Cast between function pointers with different types * Function declared implicitly.   Rec. fully covered. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/c/RuleChecker) | 23.04 | **function-prototype**  **implicit-function-declaration** | Partially checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use correct integer precisions** |
| --- | --- | --- |
| **Data Value** | [STD-002-cpp] | Integer types in C have both a size and a precision. The size indicates the number of bytes used by an object and can be retrieved for any object or type using the sizeof operator. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.  Padding bits contribute to the integer's size, but not to its precision. Consequently, inferring the precision of an integer type from its size may result in too large a value, which can then lead to incorrect assumptions about the numeric range of these types. Programmers should use correct integer precisions in their code, and in particular, should not use the sizeof operator to compute the precision of an integer type on architectures that use padding bits or in strictly conforming (that is, portable) programs. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type unsigned int. |
| #include <limits.h>    unsigned **int** pow2(unsigned **int** **exp**) {  **if** (**exp** >= **sizeof**(unsigned **int**) \* CHAR\_BIT) {      /\* Handle error \*/    }  **return** 1 << **exp**;  } |

| **Compliant Code** |
| --- |
| This compliant solution uses a popcount() function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned. |
| #include <stddef.h>  #include <stdint.h>    /\* Returns the number of set bits \*/  **size\_t** popcount(uintmax\_t num) {  **size\_t** precision = 0;  **while** (num != 0) {  **if** (num % 2 == 1) {        precision++;      }      num >>= 1;    }  **return** precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):**   1. Validate Input Data: Ensure that the integers are within their range and do not overflow. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 |  | Supported: Astrée reports overflows due to insufficient precision. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **LANG.ARITH.BIGSHIFT** | Shift Amount Exceeds Bit Width |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **C0582**  **C++3115** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-INT35-a** | Use correct integer precisions when checking the right hand operand of the shift operator |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule INT35-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint35c.html) | Checks for situations when integer precisions are exceeded (rule fully covered) |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | [STD-003-cpp] | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflows occur frequently when manipulating strings [Seacord 2013b]. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the character data to be copied and the null-termination character. (See STR03-C. Do not inadvertently truncate a string.)  When strings live on the heap, this rule is a specific instance of MEM35-C. Allocate sufficient memory for an object. Because strings are represented as arrays of characters, this rule is related to both ARR30-C. Do not form or use out-of-bounds pointers or array subscripts and ARR38-C. Guarantee that library functions do not form invalid pointers. |

| **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):**  10) Adopt a Secure Coding Standard: Make this a standard that developers should follow to avoid this risk and ensure there is enough space. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **stream-input-char-array** | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | **MISC.MEM.NTERM**  **LANG.MEM.BO LANG.MEM.TO** | No space for null terminator  Buffer overrun Type overrun |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++5216**  **DF2835, DF2836, DF2839,** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **NNTS.MIGHT** **NNTS.TAINTED** **NNTS.MUST** **SV.UNBOUND\_STRING\_INPUT.CIN** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **489 S, 66 X, 70 X, 71 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-STR50-b** **CERT\_CPP-STR50-c** **CERT\_CPP-STR50-e** **CERT\_CPP-STR50-f** **CERT\_CPP-STR50-g** | Avoid overflow due to reading a not zero terminated string Avoid overflow when writing to a buffer Prevent buffer overflows from tainted data Avoid buffer write overflow from tainted data Do not use the 'char' buffer to store input from 'std::cin' |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: STR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr50cpp.html) | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **stream-input-char-array** | Partially checked |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046388) | 4.10 | [**S3519**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-cpp.html#RSPEC-3519) |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-cpp] | Never call a formatted I/O function with a format string containing a tainted value. An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Consequently, the attacker can execute arbitrary code with the permissions of the vulnerable process [Seacord 2013b]. Formatted output functions are particularly dangerous because many programmers are unaware of their capabilities. For example, formatted output functions can be used to write an integer value to a specified address using the %n conversion specifier. |

| **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.  The incorrect\_password() function calculates the size of the message, allocates dynamic storage, and then constructs the message in the allocated memory using the snprintf() function. The addition operations are not checked for integer overflow because the string referenced by user is known to have a length of 256 or less. Because the %s characters are replaced by the string referenced by user in the call to snprintf(), the resulting string needs 1 byte less than is allocated. The snprintf() function is commonly used for messages that are displayed in multiple locations or messages that are difficult to build. However, the resulting code contains a format-string vulnerability because the msg includes untrusted user input and is passed as the format-string argument in the call to fprintf(). |
| #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):**  7) Sanitize Data Sent to Other Systems: User’s input data should be sanitized to ensure that there are no risky inputs that might lead to SQL injections. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 |  | Supported via stubbing/taint analysis |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-FIO30** | Partially implemented |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **IO.INJ.FMT MISC.FMT** | Format string injection Format string |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **TAINTED\_STRING** | Implemented |
| [GCC](https://wiki.sei.cmu.edu/confluence/display/c/GCC) | 4.3.5 |  | Can detect violations of this rule when the -Wformat-security flag is used |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **DF4916, DF4917, DF4918** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **SV.FMTSTR.GENERIC SV.TAINTED.FMTSTR** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **86 D** | Partially Implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-FIO30-a** **CERT\_C-FIO30-b** **CERT\_C-FIO30-c** | Avoid calling functions printf/wprintf with only one argument other than string constant Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable Never use unfiltered data from an untrusted user as the format parameter |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **592** | Partially supported: reports non-literal format strings |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule FIO30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulefio30c.html) | Checks for tainted string format (rule partially covered) |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.25 | [**V618**](https://pvs-studio.com/en/docs/warnings/v618/) |  |
| [Splint](https://wiki.sei.cmu.edu/confluence/display/c/Splint) | 3.1.1 |  |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Allocate and free memory in the same module, at the same level of abstraction** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-cpp] | Dynamic memory management is a common source of programming flaws that can lead to security vulnerabilities. Poor memory management can lead to security issues, such as heap-buffer overflows, dangling pointers, and double-free issues [Seacord 2013]. From the programmer's perspective, memory management involves allocating memory, reading and writing to memory, and deallocating memory.  Allocating and freeing memory in different modules and levels of abstraction may make it difficult to determine when and if a block of memory has been freed, leading to programming defects, such as memory leaks, double-free vulnerabilities, accessing freed memory, or writing to freed or unallocated memory.  To avoid these situations, memory should be allocated and freed at the same level of abstraction and, ideally, in the same code module. This includes the use of the following memory allocation and deallocation functions described in subclause 7.23.3 of the C Standard [ISO/IEC 9899:2011]:  **void** \*malloc(size\_t size);    **void** \*calloc(size\_t nmemb, size\_t size);    **void** \*realloc(**void** \*ptr, size\_t size);    **void** \*aligned\_alloc(size\_t alignment, size\_t size);    **void** free(**void** \*ptr); |

| **Noncompliant Code** |
| --- |
| [This noncompliant code example shows a double-free vulnerability resulting from memory being allocated and freed at differing levels of abstraction. In this example, memory for the list array is allocated in the process\_list() function. The array is then passed to the verify\_size() function that performs error checking on the size of the list. If the size of the list is below a minimum size, the memory allocated to the list is freed, and the function returns to the caller. The calling function then frees this same memory again, resulting in a double-free and potentially exploitable vulnerability.  The call to free memory in the verify\_size() function takes place in a subroutine of the process\_list() function, at a different level of abstraction from the allocation, resulting in a violation of this recommendation. The memory deallocation also occurs in error-handling code, which is frequently not as well tested as "green paths" through the code. |
| **enum** { MIN\_SIZE\_ALLOWED = 32 };    **int** verify\_size(**char** \*list, **size\_t** size) {  **if** (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/  **free**(list);  **return** -1;    }  **return** 0;  }    **void** process\_list(**size\_t** number) {  **char** \*list = (**char** \*)**malloc**(number);  **if** (list == NULL) {      /\* Handle allocation error \*/    }    **if** (verify\_size(list, number) == -1) {  **free**(list);  **return**;    }      /\* Continue processing list \*/    **free**(list);  } |

| **Compliant Code** |
| --- |
| To correct this problem, the error-handling code in verify\_size() is modified so that it no longer frees list. This change ensures that list is freed only once, at the same level of abstraction, in the process\_list() function. |
| **enum** { MIN\_SIZE\_ALLOWED = 32 };    **int** verify\_size(**const** **char** \*list, **size\_t** size) {  **if** (size < MIN\_SIZE\_ALLOWED) {      /\* Handle error condition \*/  **return** -1;    }  **return** 0;  }    **void** process\_list(**size\_t** number) {  **char** \*list = (**char** \*)**malloc**(number);    **if** (list == NULL) {      /\* Handle allocation error \*/    }    **if** (verify\_size(list, number) == -1) {  **free**(list);  **return**;    }      /\* Continue processing list \*/    **free**(list);  } |

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

| **Principles(s):**  9) Use Effective Quality Assurance Techniques: Allocating and freeing memory is a good habit to have and is key to quality and secure code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **ALLOC.DF ALLOC.LEAK** | Double free Leak |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Could detect possible violations by reporting any function that has malloc() or free() but not both. This would catch some false positives, as there would be no way to tell if malloc() and free() are at the same level of abstraction if they are in different functions |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **RESOURCE\_LEAK** | Fully implemented |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **FREE.INCONSISTENT** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** **MLK.MIGHT** **MLK.MUST** **MLK.RET.MIGHT** **MLK.RET.MUST** **FNH.MIGHT** **FNH.MUST** **FUM.GEN.MIGHT** **FUM.GEN.MUST** **RH.LEAK** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **50 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-MEM00-a** **CERT\_C-MEM00-b** **CERT\_C-MEM00-c** **CERT\_C-MEM00-d** **CERT\_C-MEM00-e** | Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not use resources that have been freed Ensure resources are freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) |  |  | Runtime analysis |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **449, 2434** | Partially supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rec. MEM00-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.mem00c.html) | Checks for:   * Invalid free of pointer * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rec. partially covered. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | [STD-006-cpp] | The C Standard, subclause 7.2.1.1 [ISO/IEC 9899:2011], defines assert() to have the following behavior:  The assert macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if expression (which shall have a scalar type) is false (that is, compares equal to 0), the assert macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function—the latter are respectively the values of the pre-processing macros \_\_FILE\_\_ and \_\_LINE\_\_ and of the identifier \_\_func\_\_) on the standard error stream in an implementation-defined format. It then calls the abort function.  Because assert() calls abort(), cleanup functions registered with atexit() are not called. If the intention of the programmer is to properly clean up in the case of a failed assertion, then runtime assertions should be replaced with static assertions where possible. (See DCL03-C. Use a static assertion to test the value of a constant expression.) When the assertion is based on runtime data, the assert should be replaced with a runtime check that implements the adopted error strategy (see ERR00-C. Adopt and implement a consistent and comprehensive error-handling policy).  See ERR04-C. Choose an appropriate termination strategy for more information on program termination strategies and MSC11-C. Incorporate diagnostic tests using assertions for more information on using the assert() macro. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example defines a function that is called before the program exits to clean up: |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **assert**(/\* Something bad didn't happen \*/);      /\* ... \*/  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the call to assert() is replaced with an if statement that calls exit() to ensure that the proper termination routines are run: |
| **void** cleanup(**void**) {    /\* Delete temporary files, restore consistent state, etc. \*/  }    **int** main(**void**) {  **if** (**atexit**(cleanup) != 0) {      /\* Handle error \*/    }      /\* ... \*/    **if** (/\* Something bad happened \*/) {  **exit**(EXIT\_FAILURE);    }      /\* ... \*/  } |

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

| **Principles(s):**  9) Use Effective Quality Assurance Techniques: Understanding the use of abort() and assert() are great in ensuring quality. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Can detect some violations of this rule. However, it can only detect violations involving abort() because assert() is implemented as a macro |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Enhanced enforcement |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | CERT\_C-ERR06-a | Do not use assertions |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **586** | Fully supported |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-cpp] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. The C++ Standard, [except.handle], paragraph 9 [ISO/IEC 14882-2014], states the following:  If no matching handler is found, the function std::terminate() is called; whether or not the stack is unwound before this call to std::terminate() is implementation-defined.  The default terminate handler called by std::terminate() calls std::abort(), which abnormally terminates the process. When std::abort() is called, or if the implementation does not unwind the stack prior to calling std::terminate(), destructors for objects may not be called and external resources can be left in an indeterminate state. Abnormal process termination is the typical vector for denial-of-service attacks. For more information on implicitly calling std::terminate(), see ERR50-CPP. Do not abruptly terminate the program.  All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process.  As per ERR50-CPP-EX1, a program that encounters an unrecoverable exception may explicitly catch the exception and terminate, but it may not allow the exception to remain uncaught. One possible solution to comply with this rule, as well as with ERR50-CPP, is for the main() function to catch all exceptions. While this does not generally allow the application to recover from the exception gracefully, it does allow the application to terminate in a controlled fashion. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| **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):**   1. Heed Compiler Warnings: Understanding and being able to handle warnings displayed by the compiler will help debug and prevent errors.   9) Adopt a Secure Coding Standard: Handling exceptions is a good standard to abide by when coding. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++4035, C++4036, C++4037** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **MISRA.CATCH.ALL** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **527 S** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Guarantee that container indices and iterators are within the valid range** |
| --- | --- | --- |
| **Container Indices** | [STD-008-cpp] | Ensuring that array references are within the bounds of the array is almost entirely the responsibility of the programmer. Likewise, when using standard template library vectors, the programmer is responsible for ensuring integer indexes are within the bounds of the vector. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    **void** insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  **if** (pos >= tableSize) {      // Handle error  **return**;    }    table[pos] = value;  } |

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

| **Principles(s):**  10) Adopt a Secure Coding Standard: This is a good coding standard to have to prevent data from overflowing out of containers as well as accessing memory that shouldn’t be accessed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **overflow\_upon\_dereference** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.3p0 | **LANG.MEM.BO LANG.MEM.BU LANG.MEM.TO LANG.MEM.TU LANG.MEM.TBA LANG.STRUCT.PBB LANG.STRUCT.PPE LANG.STRUCT.PARITH** | Buffer overrun Buffer underrun Type overrun Type underrun Tainted buffer access Pointer before beginning of object Pointer past end of object Pointer Arithmetic |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2023.1 | **C++3139, C++3140**  **DF2891** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2023.1 | **ABV.ANY\_SIZE\_ARRAY** **ABV.GENERAL** **ABV.GENERAL.MULTIDIMENSION** **ABV.STACK** **ABV.TAINTED** **SV.TAINTED.ALLOC\_SIZE** **SV.TAINTED.CALL.INDEX\_ACCESS** **SV.TAINTED.CALL.LOOP\_BOUND** **SV.TAINTED.INDEX\_ACCESS** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-CTR50-a** | Guarantee that container indices are within the valid range |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2023a | [CERT C++: CTR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcctr50cpp.html) | Checks for:   * Array access out of bounds * Array access with tainted index * Pointer dereference with tainted offset   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.25 | [**V781**](https://pvs-studio.com/en/docs/warnings/v781/) |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Access** | [STD-009-cpp] | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities.  It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed. |

| **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;  } |

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

| **Principles(s):**   1. Heed Compiler Warnings: This is a common compiler warning and should be followed to prevent accessing the memory.   10) Adopt a Secure Coding Standard: This is a good standard to have as accessing memory can lead to serious bugs. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 | **dangling\_pointer\_use** | Supported  Astrée reports all accesses to freed allocated memory. |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM30** | Detects memory accesses after its deallocation and double memory deallocations |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **ALLOC.UAF** | Use after free |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **USE\_AFTER\_FREE** | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **DF4866, DF4867, DF4868, DF4871, DF4872, DF4873**  **C++3339, C++4303, C++4304** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **51 D, 484 S, 112 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-MEM30-a** | Do not use resources that have been freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) |  |  | Runtime analysis |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **449, 2434** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule MEM30-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem30c.html) | Checks for:   * Accessing previously freed pointer * Freeing previously freed pointer   Rule partially covered. |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.25 | [**V586**](https://pvs-studio.com/en/docs/warnings/v586/), [**V774**](https://pvs-studio.com/en/docs/warnings/v774/) |  |
| [Splint](https://wiki.sei.cmu.edu/confluence/display/c/Splint) | 3.1.1 |  |  |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **dangling\_pointer** | Exhaustively verified (see [one compliant and one non-compliant example](https://taas.trust-in-soft.com/tsnippet/t/0d556bb8)). |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Free dynamically allocated memory when no longer needed** |
| --- | --- | --- |
| **Free Memory** | [STD-010-cpp] | Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to free() with that pointer value. |

| **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):**  10) Adopt a Secure Coding Standard: Freeing memory should be a standard that happens every time you allocate it. If memory is not freed, it can be utilized by attackers. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 23.04 |  | Supported, but no explicit checker |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM31** | Can detect dynamically allocated resources that are not freed |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.3p0 | **ALLOC.LEAK** | Leak |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **RESOURCE\_LEAK**  **ALLOC\_FREE\_MISMATCH** | Finds resource leaks from variables that go out of scope while owning a resource |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **leakReturnValNotUsed** | Doesn't use return value of memory allocation function |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2023.1 | **DF2706, DF2707, DF2708**  **C++3337, C++3338** |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/c/Klocwork) | 2023.1 | **CL.FFM.ASSIGN** **CL.FFM.COPY** **CL.SHALLOW.ASSIGN** **CL.SHALLOW.COPY** **FMM.MIGHT** **FMM.MUST** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **50 D** | Partially implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-MEM31-a** | Ensure resources are freed |
| [Parasoft Insure++](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) |  |  | Runtime analysis |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **429** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2023a | [CERT C: Rule MEM31-C](https://www.mathworks.com/help/bugfinder/ref/certcrulemem31c.html) | Checks for memory leak (rule fully covered) |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.25 | [**V773**](https://pvs-studio.com/en/docs/warnings/v773/) |  |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | [**S3584**](https://www.sonarsource.com/products/codeanalyzers/sonarcfamilyforcpp/rules-c.html#RSPEC-3584) |  |
| [Splint](https://wiki.sei.cmu.edu/confluence/display/c/Splint) | 3.1.1 |  |  |

### 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.

To create a DevSecOps process, we must implement security features throughout the DevOps. During the Access and plan phase we should be developing a security policy of standards that the team should follow. It is there that we should address IDE security and Threat Modeling to identify potential risks. This can transition to the building phase where we should be aware and implement the best coding standards to minimize bugs and risks. Frequent testing should also be done to ensure that the program is still functioning correctly. This will build up a good defense in layers. Lastly during the production phase, we should continue to monitor the program’s performance. We need to upkeep security as well as respond to any bugs that arise.

### 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 | Low | Unlikely | Low | **P3** | **L3** |
| STD-002-CPP | Low | Unlikely | Low | **P3** | **L3** |
| STD-003-CPP | Low | Unlikely | Medium | **P2** | **L3** |
| STD-004-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High | Probable | Medium | **P12** | **L1** |
| STD-006-CPP | Medium | Unlikely | Medium | **P4** | **L3** |
| STD-007-CPP | Low | Probable | Medium | **P4** | **L3** |
| STD-008-CPP | High | Likely | High | **P9** | **L2** |
| STD-009-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-010-CPP | Medium | Probable | Medium | **P8** | **L2** |

### 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 in rest | Encryption in rest is how data is protected when it is not in use. This policy protects by encrypting the disks or storage spaces the data is stored in. Databases are prime points of attack for hackers as they house valuable data. |
| Encryption at flight | Encryption in flight is how data is protected when it is being sent out. Because data is often sent through a network, the network must be secured whether it is through a firewall or some sort of authentication. This is important as hackers can intercept the data if not secured properly. |
| Encryption in use | Encryption in use is how data is protected when it has been sent out and currently in use. To protect this data, there should be several input validations in case there are any modifications to the data. There should also be authentication for if the user decides to send data back into the database. This should be kept to a minimum to limit access to the database. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is how we determine the identity of someone. This can have a user log in through their password or two-way authentication. This is important as it helps protect a user’s account and data from being accessed by another individual. |
| Authorization | Authorization is the process of giving permission or access to users. This is to limit the access someone has to something. By only providing permissions to necessary functions, we also limit the possible entry ways for attackers. This can be only allowing certain users to be able to make changes to the database, or who has access to what data. |
| Accounting | Accounting is the process of how we monitor activities within the system. By keeping an eye on who is in the systems and their activities, we can have an easier time determining if there is something out of place. This can also help with locating where the source of attack may be coming from. |

**\***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 |  |
| 2.0 | 5/21/23 | Coding principles and standards | Jiayu Lin |  |
| 3.0 | 6/11/23 | Standard risk assessment | Jiayu Lin |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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