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



# Green Pace Secure Development Policy

## Contents

[Green Pace Secure Development Policy 0](#_Toc190598510)

[Contents 1](#_Toc190598511)

[Overview 3](#_Toc190598512)

[Purpose 3](#_Toc190598513)

[Scope 3](#_Toc190598514)

[Module Three Milestone 3](#_Toc190598515)

[Ten Core Security Principles 3](#_Toc190598516)

[C/C++ Ten Coding Standards 4](#_Toc190598517)

[Coding Standard 1 5](#_Toc190598518)

[Coding Standard 2 8](#_Toc190598519)

[Coding Standard 3 11](#_Toc190598520)

[Coding Standard 4 14](#_Toc190598521)

[Coding Standard 5 17](#_Toc190598522)

[Coding Standard 6 20](#_Toc190598523)

[Coding Standard 7 23](#_Toc190598524)

[Coding Standard 8 27](#_Toc190598525)

[Coding Standard 9 31](#_Toc190598526)

[Coding Standard 10 34](#_Toc190598527)

[Defense-in-Depth Illustration 37](#_Toc190598528)

[Project One 37](#_Toc190598529)

[Revise the C/C++ Standards 37](#_Toc190598530)

[Risk Assessment 37](#_Toc190598531)

[Automated Detection 37](#_Toc190598532)

[Automation 37](#_Toc190598533)

[Summary of Risk Assessments 43](#_Toc190598534)

[Create Policies for Encryption and Triple A 43](#_Toc190598535)

[Map the Principles 50](#_Toc190598536)

[Audit Controls and Management 51](#_Toc190598537)

[Enforcement 51](#_Toc190598538)

[Exceptions Process 51](#_Toc190598539)

[Distribution 52](#_Toc190598540)

[Policy Change Control 52](#_Toc190598541)

[Policy Version History 52](#_Toc190598542)

[Appendix A Lookups 52](#_Toc190598543)

[Approved C/C++ Language Acronyms 52](#_Toc190598544)

[Appendix B Works Cited 52](#_Toc190598545)

[Works Cited 52](#_Toc190598546)

## 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 data must be validated as a preventative measure against common vulnerabilities such as SQL injection and cross site scripting. This includes checking data types, format, length, and range to ensure data input can be handled safely and securely. This also contributes to a more seamless user experience for non-malicious end users. |
| 1. Heed Compiler Warnings | Security vulnerabilities can arise when complier warnings are present. Developers are able to identify and resolve problems in programs early in development by addressing these warnings promptly when they occur. This will reduce the risk of security issues at the time of deployment. |
| 1. Architect and Design for Security Policies | By building the system architecture and design based on security policies, the final product will be inherently safer. This is because security measures like encryption and access control will be a foundational aspect of the architecture and more robustly and thoughtfully implemented. |
| 1. Keep It Simple | Complexity in security can lead to a system that is hard to maintain. By designing the system to be as simple as possible, it will be easier for developers to identify and troubleshoot vulnerabilities throughout the product lifecycle. It will also reduce maintenance costs to upkeep the security of the system over time when kept simple. |
| 1. Default Deny | To deny by default reduces the risk of unauthorized access by assuming all requests are threats unless they have the necessary permissions. This can be implemented in several ways such as through a firewall or whitelisting protocol. This principle is well aligned with a strong access control policy and the principle of least privilege that limits access of data to only users are resources that need it. |
| 1. Adhere to the Principle of Least Privilege | This principle reduces the risk of sharing sensitive data, resources, and programs with unauthorized parties by granting users, resources, and systems the minimum permissions required to perform their required duties. This way, if an account is compromised in any way, the extent of the breach is limited to only the data necessary for the compromised account. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization, or the process of permanently deleting sensitive data, is a process that supports the protection of sensitive data and prevention of data breaches. It minimizes the amount of data stored, which completely removes the risk of deleted data from being exposed in a breach. This also increases the amount of data storage available, and in some cases is required for regulatory compliance (Brook, 2024). |
| 1. Practice Defense in Depth | Implementing redundant measures of security provides protection of systems against different kinds of threats. Defense in depth can include preventative and responsive strategies to both keep attackers out of a system and also minimize the consequences in the case of a successful hack. |
| 1. Use Effective Quality Assurance Techniques | Quality checks of systems are a measure that can catch vulnerabilities before they are put into production. These techniques can include static and dynamic testing, documentation audits, compliance reviews, and cybersecurity testing. While quality standards may vary industry to industry and company to company, it is important to have them well defined and implemented for the development of secure systems. |
| 1. Adopt a Secure Coding Standard | Defining a standard for secure coding will ensure developers can easily follow best practice for reducing the risks of security vulnerabilities in a consistent and repeatable way. When standards are clear, all programs should have the same consistent security considerations regardless of who is contributing to the code. This will create a dependable reputation for Green Pace. |

### 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-CLG] | Signed char and unsigned char should be used instead of char when assigning numeric data to a character variable. This is because the char type can act as either signed or unsigned characters, leading to unpredictable behavior when compiled. Explicitly declaring signed char or unsigned char based on the needs for the variable will provide developer more consistent and predictable results (Seacord & Britton, INT07-C. Use only explicitly signed or unsigned char type for numeric values, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard any time a char data type is used to store numeric data in C and C++ programs. |
| **Where to Apply** | Apply this standard at the variable declaration, as well as in instances of type casting. |
| **How to Apply** | Apply this standard by using signed char or unsigned char for char-type numeric variable declarations. The plain char declaration should be used for only non-numeric declarations. |

| **Noncompliant Code** |
| --- |
| In the following code, it is ambiguous whether value, declared as a char variable, will be treated as a signed or unsigned character. This will impact the results because if it is treated as a signed character, the result will be -212, while if it is unsigned, the result will be 300. |
| #include <stdio.h>  void calculate() {  char value = 150;  int factor = 2;  printf("Result: %d\n", value \* factor);  }  int main(){  calculate();  return 0;  } |

|  |
| --- |
| In the following code, the process\_payments function relies on negative values for payment validation, and assumes that the payment value will be signed. If the program compiles the payment vector as unsigned characters, negative values will be processed as positive values. |
| #include <iostream>  #include <vector>    void process\_payment(const std::vector<char>& payments) {  for (auto payment : payments) {  // This condition assumes 'char' is signed  if (payment < 0) {  std::cout << "Invalid payment: " << (int)payment << std::endl;  }  }  }  int main() {  std::vector<char> payments = {100, -100};  process\_payment(payments);  return 0;  } |

| **Compliant Code** |
| --- |
| In the following code, value is assigned as an unsigned char. This will make any numeric operations on that variable predictable and consistent across platforms. This is because it eliminates ambiguity in its representation and the risk of unintended behavior dependent on how a platform compiles value. |
| #include <stdio.h>  void calculate() {  unsigned char value = 150;  int factor = 2;  assert(value <= 255); // verify char is within valid range  printf("Result: %d\n", value \* factor);  }  int main(){  calculate();  return 0;  } |

|  |
| --- |
| In the following code, the process\_payments function relies on negative values for payment validation, and requires the payment value to be signed. This will allow for both negative and positive payment values to be properly validated. |
| #include <iostream>  #include <vector>    void process\_payment(const std::vector<char>& payments) {  for (signed char payment : payments) {  if (payment < 0) {  std::cout << "Invalid payment: " << payment << std::endl;  }  }  }  int main() {  std::vector<signed char> payments = {100, -100};  process\_payment(payments);  return 0;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **1. Validate Input Data** | When using numeric data as a char data type, declaring it explicitly as a signed or unsigned char type ensures the data is correctly interpreted regardless of the platform used. This will help prevent undefined behavior which as negative values being interpreted as large positive values, as well as large positive vales being interpreted as negative. System stability or security could be compromised by errors related to lack of type validation. Therefore, it is crucial to properly validate data to ensure predictable behavior as a measure against exploitable vulnerabilities. |
| **10. Adopt a Secure Coding Standard** | By defining expectations for use of unsigned and signed char types, the policy enforces uniformity for how data is handled. This will reduce ambiguity for developers as they have clear rules consistent among all contributing to any given code base. Consistency in use of data types is a large part of implementing a secure and systematic approach to programming. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Medium | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://www.mathworks.com/help/bugfinder/index.html) | R2024a | [CERT C: Rec. INT07-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.int07c.html) | Checks numeric values for char type |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | **CERT\_C-INT07-a** **CERT\_C-INT07-b** | Checks that char type used only for character data, and signed/unsigned char type only used for numeric data |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Initialize variables at time of declaration to avoid unexpected behavior. Even if the variable is overwritten before being used elsewhere in the code, assigning an initial value will allow the developer to more accurately predict behavior in the case of the variable being used prior to re-assignment (Pincar & Britton, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever variables are declared. |
| **Where to Apply** | Apply this standard in all code, functions, constructors, and data structures where uninitialized variables run the risk of being used. |
| **How to Apply** | Apply this standard by assigning default values to variables upon declaration, even if this value is promptly updated after declaration. If the initial value is unknown, or there is a risk associated with assigning an arbitrary initial value, assign it to a value of zero or nullptr, or the safest state for the variable purpose (for example, consider for Boolean declarations if true or false is safer for an initial state given the variable purpose). |

| **Noncompliant Code** |
| --- |
| In this example, variables a and b are declared as integers in the add function but not initialized with a value assignment. They are then returned from the function. This can cause an uninitialized memory error or other unexpected output. |
| #include <iostream>  int add() {  int a, b; // uninitialized variables  return a + b; // Risks producing garbage values/undefined behavior  }  int main()  {  int sum = add();  std::cout << sum << std::endl;  return 0;  } |

|  |
| --- |
| In the following code, the sum variable used in the calculate\_average function is declared but not initialized, so it runs the risk of containing an arbitrary value or garbage data. This will result in undefined behavior and incorrect return values. |
| #include <iostream>  #include <vector>  double calculate\_average(const std::vector<int>& numbers) {  // Uninitialized variable  double sum;  // Undefined behavior: sum variable is not initialized  for (int num : numbers) {  sum += num;  }  return sum / numbers.size();  }  int main() {  std::vector<int> data = {10, 20, 30, 40, 50};  double average = calculate\_average(data);  std::cout << "Average: " << average << std::endl;  return 0;  } |

| **Compliant Code** |
| --- |
| In the code below, integer variables a and b are initialized with values before being returned. Now the behavior of this function is well defined and has predictable consistent behavior. |
| #include <iostream>  int add() {  int a = 2, b = 3;  assert(a == 2 && b == 3); // verify proper initialization  return a + b;  }  int main()  {  int sum = add();  std::cout << sum << std::endl;  } |

|  |
| --- |
| In the following code, the sum variable used in the calculate\_average function is declared and initialized. This will ensure that operations involving sum are well defined and produce predictable values. |
| #include <iostream>  #include <vector>  double calculate\_average(const std::vector<int>& numbers) {  // Initialized variable  double sum = 0.0;  for (int num : numbers) {  sum += num;  }  return sum / numbers.size();  }  int main() {  std::vector<int> data = {10, 20, 30, 40, 50};  double average = calculate\_average(data);  std::cout << "Average: " << average << std::endl;  return 0;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **1. Validate Input Data** | Uninitialized variables can contain garbage data, leading to unexpected, undefined behavior, and therefore security vulnerabilities. These vulnerabilities can include issues like leaking sensitive memory content. In combination with validating all input data, initialization of variables when they are declared will ensure they are always in a valid state and reduce the risk of using invalid or garbage data in ways that could cause holes in security. |
| **9. Use Effective Quality Assurance Techniques** | Initialization of variables with a value at the time of declaration allows developers to accurately predict code behavior during testing and production. Quality assurance methods like the Clang static analyzer can help developers identify uninitialized variables (Umannab & Porkolábac, 2022). Use of effective quality assurance and quality check techniques provides a systematic approach to identifying logical issues early to avoid them from propagating to later development stages. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: EXP53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp53cpp.html) | Checks for uninitialized variables and pointers |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-EXP53-a** | Checks for use of variable before it is initialized |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | C++ code must use and manipulate strings and character arrays in a way that prevents buffer overflow and accounts for null-terminated strings. All buffers should be appropriately sized for input data (including null terminators), or use other data types such as strings to avoid buffer overflow. Buffer overflow occurs when a buffer receives data exceeding the allocated memory. This can lead to undefined behavior and security vulnerabilities (Pincar & Britton, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever working with user input, file input and output, and any other data that deals with strings or character arrays. |
| **Where to Apply** | Apply this standard wherever character or string data is declared or resized. |
| **How to Apply** | Apply this standard by using safe alternative to character arrays such as std::string whenever possible. In the case where character arrays must be used, implement null termination, buffer size checking, and other forms of input validation to avoid buffer overflows. |

| **Noncompliant Code** |
| --- |
| In this example, the use of a character array for user input will allow the program to write beyond the allocated buffer. The std::cin >>buffer operation does not require input to be less than the allocated buffer. This will lead to buffer overflow if the entered data exceeds the 15-character buffer size. |
| #include <iostream>  void noncompliant\_process\_input() {  char buffer[15];  std::cin >> buffer;  std::cout << "Input data: " << buffer << std::endl;  } |

|  |
| --- |
| In this example, the strcopy() function does not check if the user input exceeds the buffer size. This will lead to buffer overflow and security vulnerabilities |
| #include <iostream>  #include <cstring>  void string\_copy(const char\* user\_input) {  char buffer[10];  // Unsafe: No bounds checking  strcpy(buffer, user\_input);  std::cout << "User input: " << buffer << std::endl;  }  int main() {  const char\* input = "This long input will overflow the buffer!";  // Causes buffer overflow  string\_copy(input);  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant code uses a string to receive user input, which can safely manage memory without risk of buffer overflow. The std::string class dynamically manages memory, so there is no risk of buffer overflow when using a string instead of a character array for input. |
| #include <iostream>  void compliant\_process\_input() {  std::string input;  std::cin >> input;  std::cout << "Input data: " << input << std::endl;  } |

|  |
| --- |
| In this example, the strcopy() function ensures that the maximum input size is one less than the buffer size to leave space for the null terminator. The buffer is then null terminated to prevent undefined behavior down the road. |
| #include <iostream>  #include <cstring>    void string\_copy(const char\* user\_input) {  char buffer[10];  // Safely copies with bounds  strncpy(buffer, user\_input, sizeof(buffer) - 1);  // Ensures null termination  buffer[sizeof(buffer) - 1] = '\0';  std::cout << "User input: " << buffer << std::endl;  }    int main() {  const char\* input = "Safe input";  // Proper handling of user input  string\_copy(input);  return 0;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **1. Validate Input Data** | Validation of input data will prevent user-provided data from exceeding the allocated memory buffer size. This preventative measure will stop the attacker’s ability to overwrite adjacent memory regions, which is a common mechanism of buffer overflow attacks. These kinds of attacks could lead to arbitrary code execution, so robust methods of input validation should be developed and tested in depth to stop attack attempts early (Parker, n.d.). |
| **8. Practice Defense in Depth** | Combining input validation with other layers of security, such as use of non-executable stacks, address space layout randomization, regular patch distribution, and other safe memory management practices can help mitigate against the impact of attacks even if another layer of defense fails. For example, if a weakness in input validation fails to prevent a case of input validation, subsequent layers of protection like address space layout randomization can stop the attack in other ways (Lightner, 2024). |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.3p0 | **MISC.MEM.NTERM** **LANG.MEM.BO** **LANG.MEM.TO** | Checks for null terminator, buffer overrun, and type overrun |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CLG] | All SQL queries should use clean data checked for suspicious characters to prevent injection that could compromise data integrity. The data check should ensure that no dangerous characters, sequences of characters, suspicious patterns, or input of extreme lengths are used within SQL queries. An attacker could exploit unchecked user input for injection attacks and gain unauthorize access to data (Seacord & Britton, 2022). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard when executing database queries using data from user input |
| **Where to Apply** | Apply this standard in the code base while translating user input to query formation for SQL databases, or other similar queries (such as API calls) |
| **How to Apply** | Apply this standard by validating and sanitizing all user inputs destined for query formation or other forms of communication between systems. The validation process should check for unexpected or unsafe characters and patterns, as well as unsafe input lengths. Once an instance of injection is suspected, stop processing and prevent communicating the query to another resources like a database or external system. Provide adequate feedback or error messages to the user to support the resubmission of a safe input for query formation. |

| **Noncompliant Code** |
| --- |
| The following code is noncompliant because it uses concatenated data to form a query without performing and checks for dangerous characters or other high-risk patterns. It simply takes an input and concatenates it with a query before submitting it to a database. This code has high risk for SQL injection. |
| #include <stdio.h>  #include <string.h>  void query\_database(char \*user\_input) {  char query[256] = "";  strcat(query, "SELECT \* FROM users WHERE username = '");  strcat(query, user\_input);  strcat(query, "';");  printf("Executing query: %s\n", query);  // handle database query here  } |

| **Compliant Code** |
| --- |
| In this compliant example, a regular expression is used to ensure the user input contains only alphanumeric characters, underscores, and hyphens. If the input matches the regex pattern and its lengths is shorter than the specified clean input length (128 characters including the null terminator), it is considered valid and used in the query. This approach ensures robust input validation and sanitization, preventing SQL injection attacks. |
| #include <stdio.h>  #include <string.h>  #include <regex.h>    // method for checking for SQL injection based on character and size  int is\_injection(const char \*input, char \*clean, size\_t size) {  // use regular expressions  regex\_t regex;  regcomp(&regex, "^[a-zA-Z0-9\_-]+$", REG\_EXTENDED);  // check for regular expression matching and input size  int valid = (regexec(&regex, input, 0, NULL, 0) == 0) &&  (strlen(input) < size - 1);  // if valid, copy string including null terminator  if (valid) {  strncpy(clean, input, size - 1);  clean[size - 1] = '\0';  }  regfree(&regex);  return valid;  }    // query to database with user input  void query\_database(char \*user\_input) {  char clean\_input[128];    // check for SQL injection concerns, only passing to database if  // clean  if (is\_injection(user\_input, clean\_input, sizeof(clean\_input))) {  char query[256];  snprintf(query, sizeof(query), "SELECT \* FROM users WHERE  username = '%s';", clean\_input);  printf("Executing clean query: %s\n", query);  // Simulate execution of the query  } else {  printf("Possible SQL injection. Query not sent.\n");  }  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **7. Sanitize Data Set to Other Systems** | By sanitizing data before it is sent to other systems, potentially dangerous characters and sequences, as well as sensitive information, is removed from data before it is transmitted. Certain characters or sequences can change the intended SQL query structure, leading to injection attacks. This principle is applied quite explicitly through this policy by ensuring all user input has been evaluated for and cleaned of malicious patterns prior to posting SQL queries. |
| **1. Input Validation** | Input validation can prevent SQL injection by only allowing data of a certain type or format into the system in the first place. This principle acts as the first line of defense in programs by directly evaluating data as it enters the system. Input validation can check for specific malicious patterns similar to sanitization, or even prevent uncharacteristically large inputs from being allowed as input. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 8.3p0 | **IO.INJ.COMMAND** **IO.INJ.FMT** **IO.INJ.LDAP** **IO.INJ.LIB** **IO.INJ.SQL** **IO.UT.LIB** **IO.UT.PROC** | These checkers check for command injection, format string injection, LDAP injection, library injection, SQL injection, untrusted library load, and untrusted process creation |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented checker for tainted strings |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2024.2 | **CERT\_C-STR02-a** **CERT\_C-STR02-b** **CERT\_C-STR02-c** | Protects against command, file name, and SQL injection |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. STR02-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.str02c.html) | Checks for external command execution, and commands and library loading from externally controlled path |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Memory allocation errors should be detected and handled appropriately. Exceptions must be caught and handled appropriately or new variables can be declared using std::nothrow if it is preferable to avoid exceptions altogether and instead detect a nullptr (Pincar & Britton, MEM52-CPP. Detect and handle memory allocation errors, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever memory is being manually managed |
| **Where to Apply** | Apply this standard in functions that manually manage memory for arrays, objects, buffers, or any other data type that runs the risk of causing memory allocation errors. |
| **How to Apply** | Apply this standard by implementing thorough exception handling, or by using nothrow declarations with *new* declarations to handle memory allocation errors. Properly deallocate memory whenever it is no longer in use. Whenever reasonable to do so, use automatic memory management to avoid memory allocation risks intrinsically. Prior to using a pointer after declaration, check for successful memory allocation. |

| **Noncompliant Code** |
| --- |
| A variable created without anticipating a memory allocation error can result in abnormal termination of the program if there is not enough memory available to allocate for this declaration. |
| int \*data = new int[100]; |

|  |
| --- |
| In the following noncompliant code example memory is dynamically allocated, but never freed. Memory allocation errors are not handled either. This can cause a memory leak later in processing. |
| void process\_data() {  // Dynamically allocated memory  int\* data = new int[1000];    for (int i = 0; i < 1000; ++i) {  data[i] = i;  }    // No delete call, causing a memory leak  } |

| **Compliant Code** |
| --- |
| Two variations of handling a memory allocation error are shown. The first uses nothrow to return a null pointer when there is not enough memory to allocate to the declaration. The second uses the bad\_alloc exception to catch the error and print an error message to the console. In both cases, the lack of memory to allocate should be handled after detecting the null pointer or exception. |
| // using nothrow to return a null pointer for memory allocation errors  int \*data = new (std::nothrow) int[100];  if (!data) {  std::clog << "Memory allocation error" << std::endl;  // handle error  }    // using exception handling to identify memory allocation errors  try {  int \*data = new int[100];  } catch (std::bad\_alloc &e) {  std::cerr << "Memory allocation error: " << e.what() << std::endl;  // handle error  } |

|  |
| --- |
| In the following compliant code example, a smart pointer is used to automatically allocate memory for processing. This will help prevent memory leaks. |
| #include <iostream>  #include <memory>    void process\_data() {  std::unique\_ptr<int[]> data = std::make\_unique<int[]>(1000);    for (int i = 0; i < 1000; ++i) {  data[i] = i;  }  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **5. Default Deny** | In the case where memory allocation fails the default action from the system should be to deny any additional operations relying on memory. This principle will protect the system by preventing continued operation without enough memory resources, which could otherwise result in crashes without proper termination handling. |
| **8. Practice Defense in Depth** | Addition of memory allocation error handling adds a layer of defense to the system. Even in the case of resources exhaustion, such as in DOS attacks, the system can handle failures of memory allocation safely and stably, without causing security risks. This standard should be one of many layers for defending the system in depth. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem52cpp.html) | Checks for unprotected dynamic memory allocation (rule partially covered) |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.4 | **C++3225, C++3226, C++3227, C++3228, C++3229, C++4632** |  |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/cplusplus/PVS-Studio) | 7.35 | [V522](https://pvs-studio.com/en/docs/warnings/v522/)**,**[V668](https://pvs-studio.com/en/docs/warnings/v668/) |  |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Klocwork) | 2024.4 | **NPD.CHECK.CALL.MIGHT** **NPD.CHECK.CALL.MUST** **NPD.CHECK.MIGHT** **NPD.CHECK.MUST** **NPD.CONST.CALL** **NPD.CONST.DEREF** **NPD.FUNC.CALL.MIGHT** **NPD.FUNC.CALL.MUST** **NPD.FUNC.MIGHT** **NPD.FUNC.MUST** **NPD.GEN.CALL.MIGHT** **NPD.GEN.CALL.MUST** **NPD.GEN.MIGHT** **NPD.GEN.MUST** **RNPD.CALL** **RNPD.DEREF** |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CLG] | Assertion statements must be free of side effects to produce predictable, consistent behavior regardless of assertion status. To achieve this, assertions must be used for testing assumptions only, not for altering program states. This will prevent expressions from being skipped when NDEBUG is defined (Seacord & Shrivastava, 2024). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever assert() statements are used |
| **Where to Apply** | Apply this standard throughout any debugging or testing code |
| **How to Apply** | Apply this standard by formulating assertion statements that do not change any program states. To properly assert that a program state has changed, first assert initial conditions, then complete the execution of any state-changing code, and then assert the outcome of the state change after the execution. Assertions should only validate conditions, not make modifications to any variables. |

| **Noncompliant Code** |
| --- |
| In the C code below, the counter variable is incremented within the assertion statement. If assertions are disabled, then the counter increment expression will be skipped and the counter variable will not be changed. This could result in incorrect or unexpected program behavior. |
| #include <assert.h>  static int counter = 0;  void increment\_counter() {  assert(counter++ > 0);  } |

|  |
| --- |
| In the noncompliant C++ code below, a function call is used in an assertion. If the assertions are disabled, this call will never be executed, to the program will have different behavior, which can lead it errors and unexpected results. |
| #include <cassert>  static int value = 10;  int get\_value(){  return ++value;  }    int main(){  // Function call alters program state  assert(get\_value() > 10);  return 0;  } |

| **Compliant Code** |
| --- |
| In this example, the counter variable is incremented in an expression before the assertion. Then, the output of that expression is tested in the assertion. This way, the assertion is purely diagnostic, and if assertions are turned off the counter will still be incremented. |
| #include <assert.h>  int counter = 0;  void increment\_counter() {  int counter\_init = counter;  assert (counter\_init == counter);  counter++;  assert(counter > counter\_init);  } |

|  |
| --- |
| In the compliant C++ code below, the function call is completed prior to the assertion to ensure there are no side effects. This way the program will produce the same results regardless of if assertions are turned off. |
| #include <cassert>  static int value = 10;  int get\_value(){  return ++value;  }    int main(){  int increment\_value = get\_value();  // assertion validates value without changing program state  assert(increment\_value > 10);  return 0;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **4. Keep It Simple** | Assertions should be used for diagnostic purposes only, and therefore should not change any program states. To avoid unexpected side effects, the program should include assertions that are simple, without any execution statements within assertions. This standard supports this principle by simplifying assertion code to maintain predictable program behavior. |
| **9. Use Effective Quality Assurance Techniques** | Assertions are useful tools for program testing ad debugging. By avoiding any program state changes within assertions, they can properly be used for testing code quality by accurately identifying logical errors without adding inconsistencies in the way code runs during development and testing versus in productions. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Low | Low | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-PRE31** |  |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | **premium-cert-pre31-c** |  |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.EXP31** **CC2.PRE31** |  |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/c/Helix+QAC) | 2024.4 | **C3462, C3463, C3464,C3465,C3466,C3467**  **C++3225, C++3226, C++3227, C++3228, C++3229** |  |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions should be handled to prevent abnormal termination of a program. Abnormal termination can cause resources like memory allocation to bypass proper destruction and open up the program to denial-of-service vulnerabilities. Catch-all statements, while useful for ensuring programs do not crash, completely hide errors if it does not log or rethrow the exceptions it catches. In the case where a catch all statement is used, log error details before rethrowing to avoid silently ignoring errors. By properly handling all exceptions, the developer can create a clean, safe, and predictable system even when internal errors occur (Ballman & Britton, ERR51-CPP, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever there is code that throws or catches exceptions |
| **Where to Apply** | Apply this standard in error-handline functions, API calls, when calling functions that can throw exceptions, and in all system operations that carry risk with abnormal program termination. |
| **How to Apply** | Apply this standard by ensuring all possible thrown exceptions are handled. Use specific catch blocks with clear messages to support the troubleshooting of exceptions thrown. When writing sequential catch blocks, order them from most-derived to least derived to ensure messages are returned with the most specificity. In the case where a catch all block is used, log error details and clean up resources (such as file closing or deallocating memory) before rethrowing. |

| **Noncompliant Code** |
| --- |
| In the following code, the function throwing\_function() throws a runtime error exception that is passed on to the handler\_function, which uses only a catch all block to handle the exception without proper logging or rethrowning. This hides the error and will make it very difficult to take corrective action and identify the location and cause of the error. Later, in main, the throwing\_function() is called directly without any form of handling. This can cause unexpected termination of the program, introducing a vulnerability to the system. |
| #include <iostream>  // function for throwing exceptions  void throwing\_function() {  throw std::runtime\_error("Runtime error thrown.");  }    void handler\_function() {  try {  throwing\_function();  // Catches all exceptions but hides details  } catch (...) {  // No logging, no rethrowing, and no specific handling  }  }    int main() {  // No useful error output  handler\_function();    // calling throwing function without any form of handling  throwing\_function();  return 0;  } |

|  |
| --- |
| In the following non-compliant code, there is an error handler defined for each exception type thrown in the throwing\_function, but the handler has catch blocks handling exceptions in the wrong order. Both invalid\_argument and runtime\_error exceptions are derived from the standard exception class, so these handlers will be unreachable. |
| #include <iostream>  #include <stdexcept>    void throwing\_function(int value) {  if (value < 0) {  throw std::runtime\_error("Negative value detected");  } else if (value == 0) {  throw std::invalid\_argument("Zero is an invalid input");  } else {  throw std::exception();  }  }    int main() {  try {  throwing\_function(0);    // Least derived type first  } catch (const std::exception& e) {  std::cerr << "General exception: " << e.what() << std::endl;  // More derived, but unreachable  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << std::endl;  // More derived, but unreachable  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  }    return 0;  } |

| **Compliant Code** |
| --- |
| The throwing\_function below throws a runtime error. This method is passed into handler\_function where the error is handled by a series of try-catch statements. These statements are organized by decreasing specificity, where the specific runtime error is individually handled before being passed to a catch statement that handles all standard exceptions. Finally, a catch all statement is used to catch any exceptions that are not caught by prior statements so they can be handled. To prevent error hiding, all exceptions are rethrown to be handled by the higher level handling in main(). Exceptions are also logged in an external file and printed to the console to support better debugging and tracking. This is particularly important for the catch all handler to avoid silently ignoring errors when non-standard exceptions are thrown. |
| #include <fstream>  #include <iostream>  #include <stdexcept>    // logging function  void log\_error(const std::string &message) {  // Log to console  std::cerr << "Error: " << message << std::endl;  // Append to log file  std::ofstream log\_file("error\_log.txt", std::ios::app);  if (log\_file) {  log\_file << "Error: " << message << std::endl;  }  }    // Function that throws an exception  void throwing\_function() {  throw std::runtime\_error("Runtime error thrown.");  }    void handler\_function() {  try {  throwing\_function();  // Specific exception handling  } catch (const std::runtime\_error &e) {  log\_error(std::string("Runtime error: ") + e.what());  // Rethrow to avoid error hiding  throw;  // General standard exception handling  } catch (const std::exception &e) {  log\_error(std::string("Standard exception: ") + e.what());  // Rethrow to avoid error hiding  throw;  // Catch-all, with proper logging  } catch (...) {  log\_error("Unknown error occurred. Logging and rethrowing.");  // Rethrow to avoid silent failures  throw;  }  }    int main() {    // higher level handler to handle rethrown exceptions  try {  handler\_function();  } catch (const std::exception &e) {  log\_error(std::string("Exception caught in main(): ") + e.what());  }    return 0;  } |

|  |
| --- |
| In the following compliant code, there is an error handler defined for each exception type thrown in the throwing\_function, and the handler has catch blocks handling exceptions in order of most derived to least derived. Both invalid\_argument and runtime\_error exceptions are derived from different branches of the standard exception class, so the order in which these handlers are called does not matter. |
| #include <iostream>  #include <stdexcept>    void throwing\_function(int value) {  if (value < 0) {  throw std::runtime\_error("Negative value detected");  } else if (value == 0) {  throw std::invalid\_argument("Zero is an invalid input");  } else {  throw std::exception();  }  }    int main() {  try {  throwing\_function(0);    // most derived exceptions caught first  } catch (const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << std::endl;  } catch (const std::runtime\_error& e) {  std::cerr << "Runtime error: " << e.what() << std::endl;  }  // least derived exceptions caught last  catch (const std::exception& e) {  std::cerr << "General exception: " << e.what() << std::endl;  }    return 0;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **6. Adhere to the Principle of Least Privilege** | Handling exceptions properly can prevent unauthorized access to resources when a program enters an error state. As an example, if an error is left unhandled, failure to close files or clean up other resources, could cause them to be left exposed to unauthorized parties. The handling of exceptions aligns with the principle of least privilege by limiting the exposure of resources to users, resources, or other integrations that should not have access to them during error states. |
| **9. Use Effective Quality Assurance Techniques** | Systems will be predictable and testable when all exceptions are fully handled. Robust exception handling will reduce the probability of unhandled exceptions causing crashes or other undefined behavior. A large part of quality assurance is reliability: ensuring that functions behave in an expected pattern even in fringe cases or abnormal states. Handling all exceptions will support quality assurance techniques by reliably handling abnormal system states and conditions. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | High | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.3p0 | **LANG.STRUCT.UCTCH** | Checks for unreachable catches |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-ERR51-a** **CERT\_CPP-ERR51-b** | Checks that exceptions are always caught and each exception that is thrown in the code has a compatible handler in each call path it could be thrown in |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for all unhandled exceptions |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **main-function-catch-all** **early-catch-all** |  |

#### 

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-008-CPP] | When a function is value-returning, make sure that all paths within the function return a value that matches the return type. For value-returning functions, creating paths that do not return a value can result in runtime errors, unpredictable behavior, and security vulnerabilities. The only exception to this standard is the main() function, which automatically returns 0 (Ballman & O'Donnell, 2024); |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever there is a non-void-returning function declaration |
| **Where to Apply** | Apply this standard wherever functions have multiple exit paths, such as in the case of conditional branches. |
| **How to Apply** | Apply this standard by ensuring all possible thrown exceptions are handled. Use specific catch blocks with clear messages to support the troubleshooting of exceptions thrown. When writing sequential catch blocks, order them from most-derived to least derived to ensure messages are returned with the most specificity. |

| **Noncompliant Code** |
| --- |
| In the following non-compliant code example, if the function is passed if a multiple less than zero, the function will not return a value although it is defined to return an integer. |
| int double\_func (int multiple) {  if (multiple > 0) {  return multiple \* 2;  }  } |

|  |
| --- |
| In the following code example, the function also fails to return a value in the catch block. This path may lead to undefined behavior. |
| int divide\_func (int dividend, int divisor) {  try {  if (divisor == 0) {  throw std::invalid\_argument("Division by zero error");  }  return a / b;  } catch (const std::invalid\_argument &e) {  // Exception handler, no value returned  }  } |

| **Compliant Code** |
| --- |
| In the following code example, the compliant\_double\_func function has a return statement for all valid paths, ensuring compliance with the standards by defining return statements for all paths and preventing undefined behavior. |
| int double\_func (int multiple) {  if (multiple > 0) {  return multiple \* 2;  }  // addition of return for pathway multiple < 0  return 0;  } |

|  |
| --- |
| In the following code example, the compliant\_divide\_func, even if an exception is thrown, the function will still return a value as defined by its int declaration. |
| int divide\_func (int dividend, int divisor) {  // return an integer in all pathways even if unreachable after  // catch  try {  if (divisor == 0) {  throw std::invalid\_argument("Division by zero error");  }  return a / b;  } catch (const std::invalid\_argument &e) {  // Exception handler  return 0;  } catch(…){  // handle all other exceptions  return 0;  }  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **4. Keep it Simple** | One way to help ensure all paths within a function return a value when expected is by keeping functions as simple as possible. In some cases, this can mean breaking down large, highly complex functions into smaller individual functions to be run in series for a more modular approach. Having smaller, simpler functions can help reduce path ambiguity or complexity and make it more straightforward to test properly to ensure all paths return a value when expected. Additionally, by simplifying the paths for modular functions, the program as a whole can be simplified and easier to maintain and troubleshoot. |
| **10. Adopt a Secure Coding Standard** | Requiring all function paths to return values reduces risks by reducing the risk of undefined behavior caused by missing return values (or return values of the wrong type). Enforcing this standard supports the adoption of a secure coding standard by providing a clear rule for developers to consistently align with best practices during their software development. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **return-implicit** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **2 D, 36 S** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-MSC52-a** | Checks that all paths within non-void return functions have an explicit return statement |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MSC52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmsc52cpp.html) | Checks for missing return statements |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **return-implicit** |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Management** | [STD-009-CPP] | Pointers should not be accessed after deallocating memory. Attempting to reference a deallocated pointer can lead to memory leaks and exploitable programs. When possible, use automatic storage duration to reduce the risk of accessing deallocated memory (Pincar & Britton, MEM50-CPP. Do not access freed memory, 2023). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever dynamically allocated is freed. |
| **Where to Apply** | Apply this standard wherever functions use dynamically allocated memory and pointers. |
| **How to Apply** | Apply this standard by using automatically managed memory whenever possible, such as through smart pointers or by using scope-limited declarations that do not need manual memory management. Whenever memory is dynamically managed, set pointers to nullptr after memory is freed. |

| **Noncompliant Code** |
| --- |
| In the following code example, the function allocates memory for an integer. It then deletes the pointer, freeing the allocated memory. After deleting the pointer, it attempts to access the deallocated memory by calling the deleted pointer. Accessing the dangling pointer could lead to undefined behavior and/or exploitable vulnerabilities. |
| #include <iostream>  void unsafe\_memory\_deallocation\_func() {  int \*ptr = new int(1);  delete ptr;  std::cout << \*ptr;  } |

| **Compliant Code** |
| --- |
| In the following code example, safe\_memory\_deallocation\_func dynamically allocates memory for an integer value. It accesses this memory, deallocates the memory, and then assigns it to the null pointer to avoid accidental attempts to access deallocated memory. |
| #include <iostream>  // using dynamic memory allocation  void safe\_memory\_deallocation\_func() {  int \*ptr = new int(1);  std::cout << \*ptr;  delete ptr;  ptr = nullptr;  } |

|  |
| --- |
| In the following code example, automatic storage duration is used. Since the variable, value, has a scope only within the function, memory does not need to manually deallocated. |
| #include <iostream>  // using automatic storage duration  void automatic\_storage\_duration\_func() {  int value = 1;  std::cout << value << std::endl;  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **8. Practice Defense in Depth** | Freed memory access is a vulnerability that can be exploited in attacks. By preventing access to freed memory through enforcement of this standard, there is a layer of protection against attacks such as use-after-free attacks, which can lead to remote code execution, information leaks, privilege escalation, and other impacts (Use after free, n.d.). |
| **3. Architect and Design for Security Policies** | When possible, select a language during the design phase that has automatic memory management, such as Java or Python (MITRE, 2024). These languages have automatic garbage collection that manage memory automatically, creating an inherently safer approach to avoiding use-after-free attacks (D., 2024). |
| **10. Adopt a Secure Coding Standard** | When it is not possible to use a language that has automatic memory management by default, as is the case with C and C++, adoption of a coding standard with clear guidelines for how to safely manage memory will enable developers to still prevent accidentally accessing freed memory in their programs. For example, the coding standard should clearly define how to safely manually manage memory using allocation and deallocation operators, when to use smart pointers for automatic memory management in C++, and how to use automatic storage duration to limit the lifetime of allocated memory to the scope its defined within (Obregon, 2024) (Pincar & Britton, MEM50-CPP. Do not access freed memory, 2023). |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Medium | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem50cpp.html) | Checks for pointers accessed out of bounds, deallocation of previously deallocated pointers, and use of previously freed pointers |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | **CERT\_CPP-MEM50-a** | Checks that resources are not used after freed |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 8.3p0 | **ALLOC.UAF** | Checks that resources are not used after freed |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **dangling\_pointer\_use** |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-010-CLG] | Do not declare variables within a switch state. This will limit the scope of the variable to the switch block and leading to uninitialized memory (Rungta & Shrivastava, 2024). |

| **Use Guidelines** | **Explanation** |
| --- | --- |
| **When to Apply** | Apply this standard whenever using switch statements |
| **Where to Apply** | Apply this standard wherever switch blocks are declared |
| **How to Apply** | Apply this standard by declaring variables prior to the beginning of switch statements to ensure they are properly initialized |

| **Noncompliant Code** |
| --- |
| In this non-compliant case, the variable result is declared within the switch block. If for any reason the switch is skipped, the result variable will not be initialized and lead to undefined behavior. |
| #include <stdio.h>  extern void func(int value);  void evaluate\_switch(int condition) {  switch (condition) {  // declaration of integer within switch  int result = 0;  func(result);  case 1:  result = 5;  // Falls through into case 2  case 2:  printf("Result: %d\n", result);  break;  default:  printf("Default case executed.\n");  break;  }  } |

| **Compliant Code** |
| --- |
| In this compliant code, the result variable is declared and initialized before the switch block. This ensures it has been defined regardless how the switch is evaluated. By initializing result before the switch, its behavior will be defined and predictable throughout the scope of the evaluate function. |
| #include <stdio.h>  extern void func(int value);  void evaluate\_switch(int condition) {  // declaration of integer before switch start  int result = 0;  func(result);  switch (condition) {  case 1:  result = 5;  // Falls through into case 2  case 2:  printf("Result: %d\n", result);  break;  default:  printf("Default case executed.\n");  break;  }  } |

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

| **Principles(s):** | **Connection to Standard** |
| --- | --- |
| **4. Keep It Simple** | When working with switches, keeping variable scope as simple as possible will help reduce potential bugs by making the code simpler to understand and maintain. |
| **10. Adopt a Secure Coding Standard** | Use of a clear coding standard that provides specific details and examples of how to manage variable initialization and scope within switches will help developers create consistent, predictable switch behavior that complies with this standard. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Low | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | 24.04 | Switch-skipped-code |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL41** |  |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **MISRA C 2004 Rule 15.0**  **MISRA C 2012 Rule 16.1** |  |
| [Cppcheck Premium](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck+Premium) | 24.11.0 | **premium-cert-dcl41-c** |  |

### 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 will be integrated throughout Green Pace’s DevSecOps process to provide enforcement and compliance with the secure coding standards and principles defined in this policy. Throughout the development process, automation tools will complement the existing DevOps infrastructure by adding continuous assessment, detection, and mitigation measures for security risks. Secure practices will be integrated with every stage of preproduction and production to support an inherently safer system. By implementation automation targeted at security optimization throughout the DevSecOps pipeline, Green Pace will reduce human error, create consistent and repeatable security practices, and dramatically reduce the response time when addressing security issues. This systematic approach will seamlessly integrate security considerations into the development process.

In preproduction, during the assess and planning stage, automation tools will be implemented to evaluate the threat landscape and regulatory compliance in relation to the product being developed. This could include automated regulatory compliance tools that consider laws relevant to the product, such as HIPAA, FERPA, or GDPR, and flag potential conflicts between laws and planned features. This can reduce the risk of investing time and resources into any features that are inherently out of legal compliance, but compliance automation should also be used throughout the DevSecOps process to ensure the product stays in compliance continuously (Fortinet, n.d.). Ealy integration of automated compliance checking will save significant development costs for Green Pace by preventing non-compliant features from being built in the first place.

During design, automation will be used to help developers to create security templates based on standards from OWASP or CIS Benchmarks. For example, OWASP Software Assurance Maturity Model (SAMM) can help Green Pace evaluate how they currently are handling security, determine the ideal security approach, create a detailed roadmap for the journey to the ideal state, and provide specific, actionable guidelines for executing specific security improvement initiatives (OWASP, n.d.). Once a template is created for security design considerations, it can be used across multiple projects, providing a systematic approach to design security. Green Pace’s automation of security templates will create consistency across projects while reducing the time spent on security architecture. This standardization will not only improve the security of each system that comes out of the DevSecOps pipeline, but also allow developers to focus on innovation more than repetitive security decisions.

In the build phase of pre-production, automation will not only check dependencies for vulnerabilities, but actively enforce securities policies and rules related to open-source components. Snyk is one tool that can handle this automation by scanning and monitoring open-source dependencies for vulnerabilities, providing automatic improvements, implementing of custom policies and rules, analyzing infrastructure code, and identifying of misconfigured security issues (What is Snyk?, n.d.). Tools like Snyk can help developers catch vulnerabilities in real time and address them before the propagate to later stages of development. Automated dependency scanning and policy enforcement creates a robust security foundation that scales with the project. This will prevent vulnerabilities within the code base in the first place and therefor reduce risks and costs of addressing vulnerabilities later in the DevSecOps process.

The verify and test phase will use automation in several ways. Fuzz testing is an automatic way of testing software that throws random inputs at the software in attempt to cause failures that could lead to security threats. This is an effective way to identify SQL injection or XSS vulnerabilities (GitLab, n.d.). Runtime Application Self-Protection (RASP) tools take a less random approach than Fuzz testing. It is typically built into the runtime environment and protects the app by monitoring, detecting, and blocking malicious activity in real time (Pathak, 2025). AI tools like ThreatModeler can also be used to simulate how a hacker would attack a system by modeling attack paths. It can then help identify and recommend controls to mitigate against vulnerabilities found in a method akin to a pseudo-white hat hacker strategy (ThreatModeler Software, n.d.). The combination of several different automated testing methods will help Green Pace release a product to production that they, regulatory bodies, and end users trust to securely guard and manage data. The combination of automated testing approaches provides comprehensive coverage that would be astronomically expensive and time consuming to achieve manually.

Moving to the production part of the DevSecOps process, during transition and health check phase, automation tools like Terraform’s tools that can be used to automate the process of setting up an IT infrastructure including security settings and firewalls. Automation of this process will be key for faster turnaround of infrastructure provisioning consistent across all environments (Automate infrastructure on any cloud with Terraform, n.d.). Once the IT infrastructure is configured, real-world attacks should be simulated on the environment to identify risk areas through penetration testing. This testing should also be automated for diversity and breadth of simulated situations. Based on the results of penetration tests, the security protocols such as additional layers of protection, bug fixes, or access control policy can be updated to address vulnerabilities. Automated infrastructure provisioning and security testing will provide standardization that eliminates configuration inconsistencies between environments and accelerates the delivery of new products and features.

Monitoring and detection will also be automated using traditional logging and analytics. Security information and event management (SIEM) solutions which handle logging and analysis of organizational data in real time, allowing security incidents to be handled promptly (Microsoft, n.d.). Given the technological advancements today, this can also be taken a step further into predictive or behavioral analytics tools that incorporate machine learning to speed up anomaly detection and minimize reaction time to threats. Cisco’s Splunk offers an AI solution for automates and speeds up detection, investigation, and response to threats. This type of solution is high-value because it goes beyond just logging of events, but it also analyzes trends and uses anomaly detection from machine learning algorithms to quickly identify and respond to abnormal system or user behavior (Splunk, n.d.). Automated monitoring, logging, and analysis provides perpetual surveillance at a scale that is not possible (or highly difficult) for human analysts to achieve manually. This will allow Green Pace to identify and address threats before significant damage is caused to the system.

Response to threats in a production environment should also be automated to efficiently and immediately mitigate against any abnormal or threatening behavior. Security orchestration, automation, and response (SOAR) systems can automate responses to attacks by defining normal system behavior and also response playbooks. This type of system is highly impactful for protecting systems because it can manage the bulk of day-to-day threats and allow security operations teams focus on only more major incidents that are more difficult to automate responses to (Microsoft, n.d.). Automated response systems provide immediate mitigation around the clock, significantly reducing the impacts of security incidents. This will allow security teams to focus their expertise on more complex challenges and innovation rather that common threats.

Maintain and stabilize phases typically include ensuring continuous compliance through regular audits. Before, during, and after attacks, it is important that a system maintains compliance with whatever regulatory standards govern the data collected and used by the system. To ensure Green Pace’s systems remain aligned with defined security baselines, automation such as Splunk’s compliance scanning tools should be used to ensure that production environments maintain clean and secure states that are within the guidelines for whichever standard is required by law (Splunk, n.d.). Automated processes to maintain and stabilize the system to return it to baseline security will provide consistent compliance with regulatory bodies. This will help maintain a secure system and also make audits simple and stress-free for Green Pace.

Implementing proactive and thorough automated security strategies throughout the DevSecOps pipeline will allow Green Pace to exceed standard security practices and achieve strong resilience, compliance, and operational efficiency. This automation-heavy approach ensures that vulnerabilities are proactively mitigated against, and that compliance is dynamically maintained. This will allow the system to respond effectively to even new and novel threats. This approach not only promotes inherent security, but also reduces operational costs by allowing development teams to deliver secure features faster.

### 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-CLG | Medium | Medium | Medium | High | 3 |
| STD-002-CPP | High | Medium | Low | High | 2 |
| STD-003-CPP | High | High | Medium | High | 3 |
| STD-004-CLG | High | High | Medium | High | 5 |
| STD-005-CPP | High | High | Medium | High | 5 |
| STD-006-CLG | Low | Low | Low | Low | 1 |
| STD-007-CPP | Medium | High | Medium | Medium | 3 |
| STD-008-CPP | Medium | High | Medium | Medium | 3 |
| STD-009-CPP | High | Medium | Medium | High | 5 |
| STD-010-CLG | Medium | Low | Low | Low | 2 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest is the encryption of data in storage, such as in databases or file systems. This prevents unauthorized access in case of a data breach or physical theft of storage devices.  For data at rest, AES-256 is recommended for the encryption standard. AES is an efficient encryption mechanism as it is both fast and strong. AES-256 was specifically chosen because it is a recommended standard for data security policies like HIPAA, PCI-DSS, and FIPS 140-3 (Alder, 2025) (Baykara, 2021) (NIST, 2024). In cases where another encryption standard is recommended by a governing body, that encryption mechanism should be used instead.  This policy applies to all sensitive data including customer information, health records of any form, financial information, and credentials. Other types of data not listed here should be encrypted to comply with any regulations such as HIPAA, GDPR, or FERPA. This includes encryption of databases, files, and drives for all primary and backup data locations for protected data.  Furthermore, encryption keys must be managed using a secure key management system restricting access based on the principle of least privilege. Access logs must be implemented to track decryption activities. |
| Encryption in flight | Encryption in flight is the encryption of data as it is transmitted over networks to prevent reading or tampering of data by unauthorized parties. This prevents interception and man in the middle attacks to maintain data confidentiality from starting point to ending point of the transmission.  For data in flight, TLS 1.3 protocol should be used to encrypt data in flight. This protocol enables the detection of data tampering, interception, and forgery using a handshake system that uses a public key to prove identity and MAC address verification to verify no tampering occurred during transmission (Cloudflare, n.d.).  This standard applies to all data transmissions including but not limited to: web application traffic (HTTPS), file transfers, email transmissions, database connections, API communications, VPN connections, and cloud service interactions. |
| Encryption in use | Encryption in use refers to the protection of data while it is being processed by memory. This will reduce the risk of exposure during execution and insider threats. Encryption of this data is a unique challenge because it is actively in use and therefore must be exposed for manipulation to some extent.  To protect data in use, secure enclaves should be used to isolate the memory as it is processing data. This will prevent even insiders and users with the highest level of privilege from accessing data while it is in use (Anjuna Security Inc, 2020). If there is ever a case where data must be processed outside of a secure enclave and the data must be protected due to its sensitive nature or to meet compliance requirements, techniques that allow for manipulation during concealment such as homomorphic encryption or secure multiparty computation should be used.  This policy applies to processing of any sensitive data including payment processing, healthcare analyses, cryptography operations, password validation, or any other manipulation of data protected by regulations. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of verifying a user or resources identity before granting access to systems or data. At Green Pace, all user authentication meets the following requirements:   * + 1. All users must have strong passwords, minimum 12 characters, including at least one of each of the following character types: uppercase letter, lowercase letter, number, non-alphanumeric symbol (OWASP, 2021)     2. Multifactor authentication (MFA) must be used (OTP to SMS/email, or biometric)     3. OAuth, OpenID, and SAML are acceptable forms of authentication     4. System generated one-time-passwords must be secure and randomly generated, with a minimum character length of 6. They should also expire after 15 minutes of issuing (OWASP, 2021)     5. In the event where an authentication factor is changed, the user should be notified     6. After five failed login attempts, the account should be temporarily locked, and the user and a system admin should be notified.     7. In the case where a password must be reset, this should be done through a secure recovery process, such as a time-restricted one-time password.   This policy helps to ensure that only legitimate users can access data and systems protected from public access. Having stringent policies about password complexity, maximum failed login attempts, and MFA provide layers of protection against brute force attacks. |
| Authorization | Authorization defines the types of access authenticated users have based on their identity or assigned user group. Strong authorization prevents users or systems from accessing data or resources beyond what is required for their purpose. This helps to prevent privilege escalation attacks. The following rules must be followed for defining authorization within systems at Green Pace:  The principle of least privilege should be implemented to ensure users and systems only have access to what is necessary for their account type and purpose.  Permissions should be reviewed regularly and changes should be made based on role changes.  Role-based access control should be implemented to simplify the process of defining authorization levels and to make authorization policies easier to maintain.  When it comes to structural changes to databases, only database administrators should be able to make modifications.  For data changes to databases, permissions should be managed by the role-based access control such that users can only modify personal data (not system data such as unique IDs) related to their own account.  In the case of new user creation, new users should be assigned roles based on principle of least privilege, and accounts must be verified before they are activated, such as through MFA guidelines defined in the authorization section.  Elevated access requests must be logged and manually managed by administrators. In the case where an elevated access request is granted, the elevated access must be revoked after a predefined amount of time based on the requirements of the request. All activity must be logged during the period of elevated access.  For file access default setting on documents should be read-only unless explicit permission is granted by the document owner or other authorized party.  No account can update authorization setting on their own account, changes must be made by another account.  This policy prevents unauthorized changes to critical systems and data that could lead to security breaches. It also ensures new user accounts are created securely and all users are limited to minimum amount of data and resources for their account purpose. Implementation of role-based access control promotes an authorization protocol that is simple to manage (in contrast with permissions managed on a per-user basis) and systematic. It also creates pathways to privilege elevation while implementing safeguards against privilege escalation. |
| Accounting | Accounting is the process of logging and tracking user activity in a system. This provides accountability and security monitoring to enable identification of abnormal activity, forensic investigations, troubleshooting of abnormal system or account behavior, and traceability for security compliance. At Green Pace, the following rules are set for accounting:  All system interactions must be logged and retained for at least one year or the minimum time requirement to meet compliance regulation. This includes logins for all security levels, login attempts, system configuration changes, data modifications, permission changes, elevated access requests, and any other data required for compliance reporting based on the quality standards required for the system.  Logs should be stored in a secure, encrypted, tamper-proof system for the amount of time required for compliance with quality standards.  Alert should be automatically sent to system administrators when abnormal activity is identified.  All file access and modifications should be logged and version history should be saved for review by document owners  Settings related to logging can only be modified by designated security administrators, and any changes to accounting settings will themselves be logged with timestamp and account information related to who, when, and where the setting was changed. This setting itself cannot be changed by any authentication level.  In the case where logging is disabled or changed, alerts should be sent to all security administrators.  Logging files cannot be manually deleted or overwritten unless their retention period has completed.  Thorough accounting of system activity creates a clear audit trail for security investigations related to security incidents, preventative maintenance, and routine or random audits. It ensures accountability even at elevated privilege levels and can provide insight for both insider and external system threats. It creates safeguards so that at even elevated privilege levels, such as for security administrators, there are logging practices in place to track activity and provide accountability. |

**\***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 | 02/16/2025 | Security Policy Initial Revision | Danielle Eeg |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

## Appendix B Works Cited

# Works Cited

Alder, S. (2025, January 9). *HIPAA Encryption Requirements*. Retrieved from The HIPAA Journal: https://www.hipaajournal.com/hipaa-encryption-requirements/

Anjuna Security Inc. (2020). *Secure Enclaves: The Powerful Way to Make Data Secure by Default*. Retrieved from Anjuna: https://cdn.prod.website-files.com/63f08d2744acc65d1d4d6d85/6405c9d5d2eb32d4786d1891\_case-study-what-is-a-secure-enclave.pdf

*Automate infrastructure on any cloud with Terraform*. (n.d.). Retrieved from Terraform Community: https://www.terraform.io/

Ballman, A., & Britton, J. (2023, April 20). *ERR51-CPP*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions

Ballman, A., & O'Donnell, J. (2024, April 01). *MSC52-CPP. Value-returning functions must return a value from all exit paths*. Retrieved from Carnegie Mellon Instituate: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths

Baykara, S. (2021, July 2021). *What Are the PCI DSS Encryption Requirements*. Retrieved from PCI DSS Guide: https://pcidssguide.com/what-are-the-pci-dss-encryption-requirements/

Brook, C. (2024, April 18). *What is Data Sanitization? How it Protects Your Organization*. Retrieved from Fortra: https://www.digitalguardian.com/blog/what-data-sanitization-how-it-protects-your-organization

Cloudflare. (n.d.). *What is TLS (Transport Layer Security)?* Retrieved from Cloudflare: https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

D., P. (2024, September 8). *Memory Management Strategies In Different Programming Languages*. Retrieved from Peerdh.com: https://peerdh.com/blogs/programming-insights/memory-management-strategies-in-different-programming-languages

Fortinet. (n.d.). *What Is Compliance Automation?* Retrieved from Fortinet: https://www.fortinet.com/resources/cyberglossary/compliance-automation

GitLab. (n.d.). *What is fuzz testing?* Retrieved from GitLab: https://about.gitlab.com/topics/devsecops/what-is-fuzz-testing/

Lightner, N. (2024, March 31). *How to detect, prevent, and mitigate buffer overflow attacks*. Retrieved from Blackduck: https://www.blackduck.com/blog/detect-prevent-and-mitigate-buffer-overflow-attacks.html

Microsoft. (n.d.). *What is Siem*. Retrieved from Microsoft Security: https://www.microsoft.com/en-us/security/business/security-101/what-is-siem

Microsoft. (n.d.). *What is SOAR*. Retrieved from Microsoft Security: https://www.microsoft.com/en-us/security/business/security-101/what-is-soar

Milne, C. (2024, November 7). *STR31-C Guarantee that storage for strings has sufficient space for character data and the null terminator*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

MITRE. (2024, November 19). *CWE-416: Use After Free*. Retrieved from CWE: https://cwe.mitre.org/data/definitions/416.html

Monga, A. (2024, May 6). *Parameterized Queries C++ Guide: How to Prevent SQL Injection with Parameterized Queries*. Retrieved from Medium: https://medium.com/@ajay.monga73/parameterized-queries-c-guide-how-to-prevent-sql-injection-with-parameterized-queries-94b8105cacbd

NIST. (2024, December 20). *Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program.* Retrieved from NIST: https://csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf

Obregon, A. (2024, May 9). *How Memory Management in C++ Works*. Retrieved from Medium: https://medium.com/@AlexanderObregon/how-memory-management-in-c-works-ed3b33b0fb9c

OWASP. (2021). *V2 Authentication*. Retrieved from GitHub: https://github.com/OWASP/ASVS/blob/master/4.0/en/0x11-V2-Authentication.md#v21-password-security-requirements

OWASP. (n.d.). *Quick start guide for version 2.0*. Retrieved from SAMM: https://owaspsamm.org/guidance/quick-start-guide/

Parker, J. W. (n.d.). *How does exploiting buffer overflow lead to privilege escalation?* Retrieved from Cyberly: https://www.cyberly.org/en/how-does-exploiting-buffer-overflow-lead-to-privilege-escalation/index.html

Pathak, A. (2025, January 20). *6 Runtime Application Self-Protection (RASP) Tools for Modern Applications*. Retrieved from Geekflare: https://geekflare.com/cybersecurity/rasp-tools/

Pincar, J., & Britton, J. (2023, April 20). *EXP53-CPP. Do not read uninitialized memory*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory

Pincar, J., & Britton, J. (2023, April 20). *MEM50-CPP. Do not access freed memory*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory

Pincar, J., & Britton, J. (2023, April 20). *MEM52-CPP. Detect and handle memory allocation errors*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM52-CPP.+Detect+and+handle+memory+allocation+errors

Pincar, J., & Britton, J. (2023, January 19). *STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Rungta, R., & Shrivastava, S. (2024, December 11). *DCL41-C Do not declare variables inside a switch statement before the first case label*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/c/DCL41-C.+Do+not+declare+variables+inside+a+switch+statement+before+the+first+case+label

Seacord, R., & Britton, J. (2022, April 27). *STR02-C. Sanitize data passed to complex subsystems*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems

Seacord, R., & Britton, J. (2023, April 20). *INT07-C. Use only explicitly signed or unsigned char type for numeric values*. Retrieved from Carnegie Mellon University : https://wiki.sei.cmu.edu/confluence/display/c/INT07-C.+Use+only+explicitly+signed+or+unsigned+char+type+for+numeric+values

Seacord, R., & Shrivastava, S. (2024, December 10). *PRE31-C. Avoid side effects in arguments to unsafe macros*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/c/PRE31-C.+Avoid+side+effects+in+arguments+to+unsafe+macros

Sen, T., & Rozenau, M. (2025, January 19). *EXP06-J Expressions used in assertions must not produce side effects*. Retrieved from Carnegie Mellon University: https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Splunk. (n.d.). *Compliance at Splunk*. Retrieved from Splink: https://www.splunk.com/en\_us/about-splunk/splunk-data-security-and-privacy/compliance-at-splunk.html#global

Splunk. (n.d.). *Splunk AI*. Retrieved from Splunk: https://www.splunk.com/en\_us/solutions/splunk-artificial-intelligence.html

*ThreatModeler Software*. (n.d.). Retrieved from ThreatModeler: https://www.threatmodeler.com/threatmodeler

Umannab, K., & Porkolábac, Z. (2022). Detecting Uninitialized Variables in C++ with the Clang Static Analyzer. *Acta Cybernetica, 25*(4), 923-940. doi:https://doi.org/10.14232/actacyb.282900

*Use after free*. (n.d.). Retrieved from Snyk: https://learn.snyk.io/lesson/use-after-free/

*What is Snyk?* (n.d.). Retrieved from Snyk: https://snyk.io/product/