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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_gjdgxs)

[Purpose 2](#_30j0zll)

[Scope 2](#_1fob9te)

[Module Three Milestone 2](#_3znysh7)

[Ten Core Security Principles 2](#_2et92p0)

[C/C++ Ten Coding Standards 3](#_3dy6vkm)

[Coding Standard 1 4](#_1t3h5sf)

[Coding Standard 2 5](#_4d34og8)

[Coding Standard 3 6](#_2s8eyo1)

[Coding Standard 4 7](#_17dp8vu)

[Coding Standard 5 8](#_3rdcrjn)

[Coding Standard 6 9](#_26in1rg)

[Coding Standard 7 10](#_lnxbz9)

[Coding Standard 8 11](#_35nkun2)

[Coding Standard 9 13](#_1ksv4uv)

[Coding Standard 10 14](#_44sinio)

[Defense-in-Depth Illustration 15](#_2jxsxqh)

[Project One 15](#_z337ya)

[1.](#_3j2qqm3) Revise the C/C++ Standards 15

[2.](#_1y810tw) Risk Assessment 15

[3.](#_4i7ojhp) Automated Detection 15

[4.](#_2xcytpi) Automation 15

[5.](#_1ci93xb) Summary of Risk Assessments 16

[6.](#_3whwml4) Create Policies for Encryption and Triple A 16

[7.](#_2bn6wsx) Map the Principles 17

[Audit Controls and Management 18](#_qsh70q)

[Enforcement 18](#_3as4poj)

[Exceptions Process 18](#_1pxezwc)

[Distribution 19](#_49x2ik5)

[Policy Change Control 19](#_2p2csry)

[Policy Version History 19](#_147n2zr)

[Appendix A Lookups 19](#_3o7alnk)

[Approved C/C++ Language Acronyms 19](#_23ckvvd)

## 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.

## 

### Ten Core Security Principles

| **Principle** | **Description** |
| --- | --- |
| 1. ValidateInput Data | Input validation refers to the practice of only allowing data which is properly formatted to enter the system (OWASP, 2025). Allowing unvalidated input to enter a system creates software vulnerabilities (Scheila, 2024) which could allow malicious actors to take advantage of a weak system. For example, numbers which are too large for their data type in order to overflow information into other areas of memory is referred to as a buffer overflow attack. Catching instances of numbers which are too large for their data type being entered as user input rather than passing them into the system would be one way to secure the system against this sort of attack.  Having user input directly target parts of the database is considered “a symptom of poor design” (OWASP, 2018) because of how easy it would be to exploit such a system using certain basic attack methods. All user input must be comprehensively evaluated in order for it to be safely used by the system. |
| 1. Heed Compiler Warnings | “[Compiler] warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error” (Free Software Foundation Inc, 2025). In other words, while compiler warnings don’t prevent a program from running, they indicate potentially risky vulnerabilities which should be addressed rather than ignored. Carnegie Mellon University’s Software Engineering Institute recommends “compil[ing] code using the highest warning level available for your compiler and eliminat[ing] warnings by actually modifying the code which caused them (Scheila & Seacord, 2018). Using analysis tools (both status and dynamic) to identify and address any additional security flaws is also a good way to address potential security flaws (Scheila & Seacord, 2018). |
| 1. Architect and Design for Security Policies | Make sure that software and software architectures are created in adherence to the security policy (Scheila & Seacord, 2018). Security policies protect companies from data breaches and leaks by acting as a formalized plan “to protect its physical and information technology (IT) assets” (Lutkevich, 2021) . For example, a policy which involves dividing a system into distinct subsystems, each with an appropriate privilege set, might be optimal for a system which requires different privileges in different situations (Scheila & Seacord, 2018). A policy which focuses on memory management might be optimal for a system with mostly C++ program components. Security policies help organizations address their specific security needs as effectively as possible, and architecting and designing systems components which implement and enforce security policies keeps an organization’s systems safe and secure. |
| 1. Keep It Simple | The design of systems and system components must be kept as simple as possible (Scheila & Seacord, 2018 , as cited in Saltzer, 1974) in order to decrease the likelihood of errors in implementation, use, and configuration (Scheila & Seacord, 2018). More complex security mechanisms lead to an increase in the effort required to achieve confidence or assurance that vulnerabilities have been successfully mitigated (Scheila & Seacord). |
| 1. Default Deny | Access should be denied by default while specific conditions which would grant access are identified in the protection scheme (Scheila & Seacord, 2018 , as cited in Saltzer, 1975). Access decisions should be made based on permission rather than exclusion (Scheila & Seacord, 2018 , as cited in Saltzer, 1975). Denying access by default helps to address potential security threats and reduces a system’s attack surface (Reason Labs, 2023), ultimately making it harder for successful attacks to be perpetrated against the system. |
| 1. Adhere to the Principle of Least Privilege | The idea that systems should allow only the minimum level privilege necessary for users to achieve their goals effectively is referred to as the principle of least privilege (National Institute of Standards and Technology, n. d.). Adhering to the principle of least privilege gives attackers less opportunities to gain entry to the system through unintended means (Saltzer, 1975, as cited in Scheila & Seacord, 2018). Users should only be given the privileges necessary to perform their duties, and should only be given them for the least amount of time necessary to perform said duties (Scheila & Seacord, 2018). |
| 1. Sanitize Data Sent to Other Systems | Data which will be passed to complex subsystems such as databases and command shells should be sanitized in order to prevent attackers from invoking functionalities which are unused (Scheila & Seacord, 2018) or unintended. Data input methods include blacklist sanitizing which involves blocking certain characters or types of input from being entered, or whitelist sanitizing which involves the defining of approved characters and input types which will be allowed to enter the system as input (Buenning, 2024).  Data sanitization is somewhat similar to input validation, but in situations where complex subsystems are not able to understand the context of a call (Scheila & Seacord, 2018). The calling process itself is then responsible for sanitizing the data since it is where the context of the call is defined (Scheila & Seacord, 2018). For example, user input which is maliciously written to change an SQL statement used to access information from a database is referred to as a SQL injection. A data sanitization method for not allowing SQL statements might involve not allowing input to include the ‘-’ character , the ‘=’ or ‘+’ operator, or the words ‘or’ or ‘and’. Catching instances of suspiciously formed user input rather than passing them into the system would secure the system against SQL injections and other types of similar attacks involving maliciously formed data. |
| 1. Practice Defense in Depth | The phrase ‘defense in depth’ in the context of system security refers to a combination of multiple defense strategies of different types and natures to address and mitigate a wide variety of potential attacks (National Institute of Standards and Technology, n. d.). The use of multiple redundant defense strategies means that if one layer of defense proves insufficient, another layer of defense can account for some of the same vulnerabilities (Scheila & Seacord, 2018). Using secure programming techniques while also using secure runtime environments, for example, is likely to reduce the probability that unnoticed vulnerabilities will be present to be exploited upon release (Seacord, 2013, as cited in Scheila & Seacord, 2018). |
| 1. Use Effective Quality Assurance Techniques | Effective quality insurance can be a powerful tool for enhancing the security of a system (Scheila & Seacord, 2018). Quality assurance measures for security include fuzz testing, penetration testing, and source code audits should all be incorporated into quality assurance programs for security (Scheila & Seacord, 2018). These techniques can be quite effective for identifying and eliminating vulnerabilities (Scheila & Seacord, 2018). The term ‘fuzz testing’ (also known as ‘fuzzing’) refers to a technique involving the use of software which observes the behavior of the system when invalid or random input is entered into the system (GitHub, 2025). The term ‘penetration testing’ refers to the practice of having mock cyberattacks performed in order to identify vulnerabilities (IBM, 2023). Having Independent security audits conducted by third parties can bring new perspectives to an organization’s security measures or to the policy as a whole (Seacord, 2013, as cited in Scheila & Seacord, 2018). |
| 1. Adopt a Secure Coding Standard | The development of a secure coding standard optimized for a project’s specifications, such as target development language(s) and platform(s) (Scheila & Seacord, 2018). Since much of the code which goes into our projects is written in C and C++, our coding standard will focus on the development of secure C and C++ programs and components. One of the main security concerns with C and C++ is the issue of memory security (Gaynor, 2019, as cited in Internet Security Research Group, 2025) and the potential for insecure code to allow malicious actors to introduce input to the system which can lead to data being compromised or stolen (Gaynor, 2019). For these reasons, our secure coding standard will focus heavily on addressing memory safety. |

### C/C++ Ten Coding Standards

#### Coding Standard 1

| **Coding Standard** | **Label** | **Use Appropriate Data Types** |
| --- | --- | --- |
| **Data Type** | STD-001-  CPP | In order to prevent buffer overflows, it is important to create an environment which guarantees that data is not copied into a container which is not large enough to contain it (Ballman, 2016). Data which is sent into a container must be validated in order to assure that the data type of that container is able to handle it without the system experiencing a buffer overflow (Ballman, 2016). Where possible, the data type of the destination container should guarantee the ability to accommodate data which will be copied into it by the system (Ballman, 2016). |

| **Noncompliant Code** |
| --- |
| The noncompliant code below is susceptible to buffer overflow attacks. By entering a number which is too large to be handled by the char array data type, the program displays the account number. Buffer overflows can allow malicious actors to extract potentially sensitive information from systems which do not effectively protect against them. |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string account\_number = "CharlieBrown42";  // The char array data type is used to store user input  // (NONCOMPLIANT)  char user\_input[20];  std::cout << "Enter a value: ";  std::cin >> user\_input;  // If the user input contains more than 20 characters,  if (user\_input.length() > 20)  {  // then tell the user that they entered too many digits and ask them  // to try again using the appropriate number of digits.  std::cout << std::endl << "Your number had too many digits. Please enter a number with 20 digits or less." << std::endl;  }  // Put this part of the code in an "else" statement so that the  // number (a string) will only be shown if the input has the  // appropriate number of digits (characters).  else  {  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  }  } |

| **Compliant Code** |
| --- |
| By giving the ‘user\_input’ variable a string data type rather than a char vector data type, the variable can handle much larger values. |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string account\_number = "CharlieBrown42";  // Set the data type of the user\_input variable to a string  // (COMPLIANT)  std::string user\_input = “string”;  std::cout << "Enter a value: ";  std::cin >> user\_input;  // If the user input contains more than 20 characters,  if (user\_input.length() > 20)  {  // then tell the user that they entered too many digits and ask them  // to try again using the appropriate number of digits.  std::cout << std::endl << "Your number had too many digits. Please enter a number with 20 digits or less." << std::endl;  }  // Put this part of the code in an "else" statement so that the number  // will only be shown if the input has the appropriate number of digits.  else  {  std::cout << "You entered: " << user\_input << std::endl;  std::cout << "Account Number = " << account\_number << std::endl;  }  } |

| **Principles:**  **Validate Input Data -** This principle directly relates to our coding standard STD-001-CPP (Use Appropriate Data Types). The use of appropriate data types along with the validation of input data using methods specific to those data types prevents potentially dangerous input from entering the system (Scheila, 2024). Failure to validate input data and to use appropriate data types could result in SQL injection, buffer overflow, and other types of attacks which can be perpetrated through the entry of malicious user input.  **Architect and Design for Security Policies -** This principle also relates to our coding standard STD-001-CPP (Use Appropriate Data Types) since it is important to assure the use of correct data types as early as possible in the production process. By including the appropriate data types in technical design documents such as UML class diagrams, developers are provided with a guide which will help them to adhere to best practices related to data type selection in specific scenarios.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure the use of appropriate data types is an important part of our Defense-in-Depth approach to security.  **Keep it Simple -** Simplicity in the choice of data types, such as the declaration of certain variables as ‘string’ type rather than ‘char array’ type in situations such as what is depicted in the code examples above, will help to prevent buffer overflow attacks.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy.  **Use Effective Quality Assurance Techniques -**  This principle is directly related to our coding standard STD-001-CPP (Use Appropriate Data Types) because the use of appropriate data types should be confirmed during the quality assurance processes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | UNLIKELY | MEDIUM | HIGH | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Axivion Bauhaus Suite** | **7.2.0** | **CertC++-DCL52**  [**C++-DCL52-CPP**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile)  “Never Qualify a Reference Type with const or volatile” (Svoboda, 2023). | The Avaxion Bauhaus Suite is a static testing tool which serves as “a Holistic Solution to Assure the Quality of…Software” (The QT Group, 2025). The suite’s ‘CertC++-DCL52’ checker helps to ensure that reference types are not qualified with type ‘volatile’ or ‘const,’ which (Svoboda, 2023). |
| **Coverity** | **2017.07** | **STRING\_OVERFLOW, BUFFER\_SIZE, OVERRUN, and STRING\_SIZE**  [**STR31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048)  “Guarantee that storage for strings has sufficient space for character data and the null terminator” (Seacord, 2024). | Coverity “provides comprehensive code scanning that empowers developers and security teams to deliver high-quality software that complies with security, functional safety, and industry standards” (Black Duck Software, 2025). Coverity’s “STRING\_OVERFLOW,” “BUFFER\_SIZE,” “OVERRUN,” and “STRING\_SIZE” checkers help to ensure that string storage “has sufficient space for character data and the null terminator” (Seacord, 2024) to mitigate against the possibility of successful buffer overflow attacks. |
| **CppCheck** | **2.15** | [**memsetValueOutOfRange**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152211)  [**INT31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152211)  “Ensure that integer conversions do not result in lost or misinterpreted data” (Seacord, 2024). | CPPCheck is a static testing tool for C and C++ code (CPPCheck, n. d.). CppCheck’s ‘memsetValueOutOfRange‘ checker helps to assure that “both implicit and explicit (using a cast)...[are]...guaranteed not to result in lost or misinterpreted data” (Seacord, 2024). |
| **Parasoft** | **2024.2** | **CERT\_CPP-INT50-a**  [**CPP-INT50**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)  “[Do not cast to an out-of-range enumeration value](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)” (Weise, 2023). | Parasoft C/C++ Check is a “Comprehensive Test Automation Solutions for C/C++ Software Development” (Parasoft, 2025). Parasoft’s ‘**CERT\_CPP-INT50-a**’ checker helps prevent the casting of out-of-range enumeration values (Weise, 2023). |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Always Initialize Variables** |
| --- | --- | --- |
| **Data Value** | STD-002-  CPP | Make sure to initialize variables, especially dynamic variables, before they are able to be called or used (GeeksforGeeks, 2023). Reading uninitialized variables can result in undefined behavior and can lead to a wide range of bugs and vulnerabilities (Gennari & Shrivastava, 2024) and less-than-optimal memory management (Reddy, 2019). |

| **Noncompliant Code** |
| --- |
| The noncompliant code below declares an integer (int) variable called ‘number’ without initializing it with a value. Since the value which will be found inside of the memory space allocated for the value of this variable is indeterminate (Gennari & Shrivastava, 2024), printing the value of this variable may disclose sensitive information. |
| #include <iomanip>  #include <iostream>  int main()  {  // Declare an integer variable called ‘number’  // (NONCOMPLIANT)  int number;  // Put the value of the ‘number’ variable to output.  std::cout << number << endl;  } |

| **Compliant Code** |
| --- |
| By initializing the ‘number’ integer (int) variable with a value of ‘0’ before it is put to output, printing or otherwise using the variable will not read uninitialized memory and therefore is less likely to result in undefined or unexpected behavior (Gennari & Shrivastava, 2024). |
| #include <iomanip>  #include <iostream>  int main()  {  // Declare an integer variable called ‘number’  // and initialize with ‘0’ as the value of the  // variable  // (COMPLIANT)  int number = 0;  // Put the value of the ‘number’ variable to output.  std::cout << number << endl;  } |

| **Principles:**  **Heed Compiler Warnings -** Variables which are not initialized properly within scope are likely to set off compiler warnings, which gives engineers and developers an additional way to assure that this standard is adhered to.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure that variables are initialized before use is an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy.  **Use Effective Quality Assurance Techniques -**  This principle is directly related to our coding standard STD-002-CPP (Always Initialize Variables). The need for the initialization of variables when coding in C++ is relatively common knowledge among professionals who use C++ or work with systems involving C++ code, and quality assurance measures should be able to identify these and other simple warnings and errors relatively easily. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | UNLIKELY | MEDIUM | MEDIUM | SL 2 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **24.04** | [**int-division-by-zero**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152254)  [**INT33-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152254)  and  [**int-modulo-by-zero**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152254)  [**INT33-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152254)  “[Ensure that division and remainder operations do not result in divide-by-zero errors](https://wiki.sei.cmu.edu/confluence/display/c/INT33-C.+Ensure+that+division+and+remainder+operations+do+not+result+in+divide-by-zero+errors)” (Shrivastava, 2024).  **integer-overflow**  [**INT30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152236)  “Ensure that unsigned integer operations do not wrap” (Seacord, 2024).  [**INT32-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152210)  “[Ensure that operations on signed integers do not result in overflow](https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow)” (Seacord, 2024).  [**INT08-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152450)  “Verify that all integer values are in range” (Seacord, 2023). | [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) is a static testing tool which “reports program defects caused by unspecified and undefined behaviours according to the C and C++ language standards, program defects caused by invalid concurrent behaviour, and computes program properties relevant for functional safety” (QA Systems, 2025). Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘int-division-by-zero’ and ‘modulo-by-zero’ checkers “e[nsure that division and remainder operations do not result in divide-by-zero errors](https://wiki.sei.cmu.edu/confluence/display/c/INT33-C.+Ensure+that+division+and+remainder+operations+do+not+result+in+divide-by-zero+errors)” (Shrivastava, 2024). Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘integer-overflow’ checker “ensure[s] that unsigned integer operations do not wrap…[that operations on signed integers do not result in overflow](https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow)…[and]...that all integer values are in range” (Shrivastava, 2024) respectively. |
| **Axivion Bauhaus Suite** | **7.2.0** | [**EXP33-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152129)  “[Do not read uninitialized memory](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)” (Gennari & Shrivastava, 2024). | The Avaxion Bauhaus Suite’s ‘EXP33-C’ checker helps to ensure that no attempt is made to read uninitialized memory (Svoboda, 2023). |
| **Coverity** | **2017.07** | **UNINIT**  [**EXP33-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152129)  “[Do not read uninitialized memory](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)” (Gennari & Shrivastava, 2024).  “” (Seacord, 2024). | Coverity’s ‘UNINIT’ checker helps to ensure that no attempt is made to read uninitialized memory (Svoboda, 2023). |
| **CppCheck** | **2.15** | **uninitdata, uninitMemberVar, uninitstring, uninitStructMember, uninitvar**  [**EXP33-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152129)  “[Do not read uninitialized memory](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)” (Gennari & Shrivastava, 2024). | CppCheck’s ‘uninitdata,’ ‘uninitMemberVar,’ ‘uninitstring,’ ‘uninitStructMember,’ and ‘uninitvar’ checkers help to ensure that no attempt is made to read uninitialized memory (Svoboda, 2023). |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Always Validate Strings to Prevent Buffer Overflows** |
| --- | --- | --- |
| **String Correctness** | STD-003-  CPP | Make sure that string storage has enough space for all string characters and the null terminator (Ballman, 2016). As is the case with any data type, copying string data to a memory location which does not have enough space to hold it results in a buffer overflow (Ballman, 2016, as cited in Seacord, 2013). Different validation methods include ensuring that the destination is large enough to hold what is being copied in, and limiting the number of characters in an input string (Ballman, 2016). |

| **Noncompliant Code** |
| --- |
| The noncompliant code below does not use string input validation to limit the size of the string being saved into the ‘user\_input’ variable. |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string secret\_phrase = "This is a secret phrase.";  string user\_input = “string”;  std::cout << "Would you like to hear a secret phrase?";  std::cin >> user\_input;  // Print the secret phrase without validating user input  // (NONCOMPLIANT)  std::cout << "You said: " << user\_input << std::endl;  std::cout << "Secret phrase = " << secret\_phrase << std::endl;  } |

| **Compliant Code** |
| --- |
| By adding the ‘if’ statement to only allow correctly formatted input, string data is prevented from being added to a memory location which does not have enough space to hold it. This mitigates against buffer overflow vulnerabilities (Ballman, 2016, as cited in Seacord, 2013) |
| #include <iomanip>  #include <iostream>  int main()  {  const std::string secret\_phrase = "This is a secret phrase.";  std::string user\_input = “string”;  std::cout << "Would you like to hear a secret phrase?";  std::cin >> user\_input;  // Use input validation to only allow correctly formatted input  // (COMPLIANT)  // If the user input contains more than 50 characters,  if (user\_input.length() > 50)  {  // then tell the user that they entered too many digits and ask them  // to try again using the appropriate number of digits.  std::cout << std::endl << "Your input had too many digits. Please enter 50 digits or less." << std::endl;  }  // Otherwise, if the user contains 50 characters or less,  // then show the account number.  else  {  std::cout << "You said: " << user\_input << std::endl;  std::cout << "Secret phrase = " << secret\_phrase << std::endl;  }  } |

| **Principles:**  **Validate Input Data -** This principle is directly related to our coding standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows) because the validation of input data is an effective mitigation strategy to prevent successful buffer overflow attack attempts.  **Default Deny -** Upon receiving input data with potentially malicious quantities, digits, operators, or other relevant characters included within, the user’s attempt to have their input used by the system will be denied. This will still take place in instances of accidental or unintentional entry of potentially malicious input since it is difficult to account for every possible attack attempt involving user input.  **Sanitize Data Sent to Other Systems -** This principle is directly related to our coding standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows) since string data a type of data which could be sent to other systems, and such data must be sanitized in order for an organization to avoid being used as an injection method for malicious user input to infect other systems.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure that input strings are validated before being used by the system is an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy.  **Use Effective Quality Assurance Techniques -**  This principle is directly related to our coding standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows) because the validation of strings to prevent buffer overflows should be confirmed during the quality assurance processes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | UNLIKELY | MEDIUM | HIGH | SL 2 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | **7.2.0** | **CertC-STR31**  [**STR31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048)  “[Guarantee that storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2024). | Axivion Bauhaus Suite’s ‘CertC-STR31’ checker helps to ensure that “[storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2023). |
| **Coverity** | **2017.07** | **STRING\_OVERFLOW, BUFFER\_SIZE, OVERRUN, and STRING\_SIZE**  [**STR31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048)  “[Guarantee that storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2024). | Coverity’s ‘STRING\_OVERFLOW,’ ‘BUFFER\_SIZE,’  ‘OVERRUN,’ and ‘STRING\_SIZE’ checkers help to ensure that “[storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2023). |
| **Parasoft** | **2024.2** | **CERT.STR00.CO**  [**STR00-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487614)  “[Don't form strings containing partial characters from variable-width encodings](https://wiki.sei.cmu.edu/confluence/display/java/STR00-J.+Don%27t+form+strings+containing+partial+characters+from+variable-width+encodings)” (Seacord, 2021). | Parasoft’s ‘CERT.STR00.CO’ checker helps to ensure that strings are not formed which contain “partial characters from variable-width encodings” (Seacord, 2021). |
| **TrustInSoft Analyzer** | **1.38** | **mem\_access**  [**STR31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048)  “[Guarantee that storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2024). | TrustInSoft Analyzer’s ‘mem\_access’ checker helps to ensure that “[storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2023). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Always Validate Input to Prevent SQL Injections** |
| --- | --- | --- |
| **SQL Injection** | STD-004-  CPP | SQL Injections occur as a result of a program’s SQL queries being manipulated by cleverly formatted input. For example, the addition of two dashes at the end of an input string could cause parts of the SQL query to be interpreted as comments, or the addition of the ‘=’ and/or ‘+’ operators could modify the query if strings are simply concatenated to create a query without having been validated beforehand. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code segment below, SQL statements are formed from strings which are formed from user input strings (in a function which is not included for the sake of brevity and readability). The ‘run\_query’ function should validate input strings before they are used to form a SQL query. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // Code for validating string input should go here  // (NONCOMPLIANT)  // Print the SQL query resulting from the input.  std::cout << std::endl << "The SQL query resulting from your input was:" << std::endl << sql << std::endl;  // Print a message that a SQL injection attack was detected.  std::cout << "An SQL Injection attempt has been detected." << std::endl << std::endl;  return false;  }  }  void run\_queries(sqlite3\* db)  {  char\* error\_message = NULL;  std::vector< user\_record > records;  // query all  std::string sql = "SELECT \* from USERS";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // Form the actual query  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // run query 1 with injection 5 times  for (auto i = 0; i < 5; ++i)  {  if (!run\_query\_injection(db, sql, records)) continue;  dump\_results(sql, records);  }  } |

| **Compliant Code** |
| --- |
| Having used regex to check for suspicious combinations of characters being included within strings which are to be used to form SQL queries, the chance of a successful SQL attack is mitigated significantly. The system prints the query which resulted from the input and, if an SQL injection is detected, a message saying that the attempt was detected is put to output. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  // Check to see if any combination of characters is being  // used to form a potentially dangerous SQL statement  // (COMPLIANT)  std::regex re("('[a-zA-Z0-9\_/.]\*'='[\\w/.]\*')|([0-9]=[0-9])|([a-z]=[a-z])|([A-Z]=[A-Z])");  std::smatch match;  // If an attack is detected, print a message communicating  // that the system was able to detect it  // (COMPLIANT)  if (std::regex\_search(sql.begin(), sql.end(), match, re) && match.size() > 1) {  // Print the SQL query resulting from the input.  std::cout << std::endl << "The SQL query resulting from your input was:" << std::endl << sql << std::endl;  // Print a message that a SQL injection attack was detected.  std::cout << "An SQL Injection attempt has been detected." << std::endl << std::endl;  return false;  }  }  void run\_queries(sqlite3\* db)  {  char\* error\_message = NULL;  std::vector< user\_record > records;  // query all  std::string sql = "SELECT \* from USERS";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // Form the actual query  sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred'";  if (!run\_query(db, sql, records)) return;  dump\_results(sql, records);  // run query 1 with injection 5 times  for (auto i = 0; i < 5; ++i)  {  if (!run\_query\_injection(db, sql, records)) continue;  dump\_results(sql, records);  }  } |
|  |

| **Principles:**  **Validate Input Data -** This principle is of course directly related to our coding standard STD-004-CPP (Always Validate Input to Prevent SQL Injections), as the prevention of successful SQL injection attempts is an important reason to validate input data.  **Sanitize Data Sent to Other Systems -** This principle is directly related to our coding standard STD-004-CPP (Always Validate Input to Prevent SQL Injections) since user input is a type of data which could be sent to other systems, and the sanitization of data sent to other systems helps mitigate the risk of being used as an injection method for SQL injection attacks against those systems.  **Adhere to the Principle of Least Privilege -** Not all users should have the ability to execute SQL commands to create, read, update or delete data from our databases. Keeping input fields from providing malicious actors with a way to execute SQL commands is incredibly important in limiting user privileges to only what is needed.  **Default Deny -** Upon receiving input data with potentially malicious characters or strings included within, the user’s attempt to have their input used by the system will be denied. This will still take place in instances of accidental or unintentional entry of potentially malicious input since it is difficult to account for every possible attack attempt involving user input.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure that all types of input are validated to thwart SQL injection attacks is an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy.  **Use Effective Quality Assurance Techniques -**  This principle is directly related to our coding standard STD-004-CPP (Always Validate Input to Prevent SQL Injections) since the validation of input to prevent SQL injections should be confirmed during the quality assurance processes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | MEDIUM | MEDIUM | HIGH | SL 3 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **24.04** | [**API00-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152074)  “Functions should validate their parameters” (Broderick & Herder, 2024). | Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘API00-C’ support helps to ensure that functions validate their parameters (Broderick & Herder, 2024). That way any functions which receive user input as parameters will validate the input locally. |
| **Coverity** | **2017.07** | **INTEGER\_OVERFLOW**  [**INT30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152236)  “[Ensure that unsigned integer operations do not wrap](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)” (Seacord & Shrivastava, 2024).  **STRING\_OVERFLOW, BUFFER\_SIZE, ‘OVERRUN and STRING\_SIZE**  [**STR31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048)  “[Guarantee that storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2024). | Coverity’s ‘STRING\_OVERFLOW,’ ‘BUFFER\_SIZE,’ ‘OVERRUN,’ and ‘STRING\_SIZE’ checkers help to “e[nsure that unsigned integer operations do not wrap](https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap)” (Seacord & Shrivastava, 2024), and to “g[uarantee that storage for strings has sufficient space for character data and the null terminator](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)” (Milne, 2024). |
| **CppCheck** | **2.15** | **memsetValueOutOfRange**  [**INT31-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152211)  “[Ensure that integer conversions do not result in lost or misinterpreted data](https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data)” (Seacord & Shrivastava, 2024). | CppCheck’s ‘memsetValueOutOfRange’ checker helps to “e[nsure that integer conversions do not result in lost or misinterpreted data](https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data)” (Seacord & Shrivastava, 2024). |
| **Parasoft** | **2024.2** | **CERT.IDS03.TDLOG**  [**IDS03-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487909)  “Do not log unsanitized user input” (Gale & Mohindra, 2025).  **CERT.IDS06.VAFS**  [**IDS06-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487836)  “Exclude unsanitized user input from format strings” (Britton & Mohindra, 2024). | Parasoft’s ‘CERT.IDS03.TDLOG’ and ‘CERT.IDS06.VAFS’ checkers help prevent unsanitized and potentially dangerous user input from compromising system security (Gale & Mohindra, 2025). ‘IDS06-J’ specifically deals with the use of user input for format strings (Britton & Mohindra, 2024),which could allow malicious code disguised as input to compromise the system. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Refrain From Accessing Dangling Pointers** |
| --- | --- | --- |
| **Memory Protection** | STD-005-  CPP | Do not attempt to access memory which has already been freed up (Britton & Pincar, 2023). Refrain from evaluating a pointer into memory if it has been deallocated by a C++ memory management function (Britton & Pincar, 2023). Evaluating a pointer would include dereferencing, typecasting, or using it as an operand (Britton & Pincar, 2023). Performing these operations on a pointer which has been deallocated by a memory management function could result in undefined behavior (Britton & Pincar, 2023). Since freeing memory renders all pointers into that memory invalid, the data present at a given location could appear valid but change unexpectedly (Britton & Pincar, 2023). For this reason, memory which has been freed up should not be read from or written to (Britton & Pincar, 2023). |

| **Noncompliant Code** |
| --- |
| In the noncompliant code segment below, the variable ‘s’ is dereferenced after first being deallocated (Britton & Pincar, 2023). If data is able to be written to freed memory, the permissions of the vulnerable function can be used to run arbitrary code. This issue is hard to diagnose (Britton & Pincar, 2023), so it is important to identify any such coding errors as early as possible. |
| #include <new>  struct S {  voidf();  };  void g() noexcept(false) {  S \* s = new S;  // The variable ‘s’ is furst deallocated,  delete s;  // then ‘s’ is dereferenced after already  // having been deallocated  // (NONCOMPLIANT)  s->f();  } |

| **Compliant Code** |
| --- |
| In the compliant code segment below, the dynamically allocated memory is first dereferenced before being deallocated (Britton & Pincar, 2023). This prevents dangling pointers from creating vulnerabilities in the code. |
| #include <new>  struct S {  voidf();  };  void g() noexcept(false) {  S \* s = new S;  // The variable ‘s’ is first dereferenced  // (COMPLIANT)  s->f();  // and the variable ‘s’ is deallocated  // after already having been dereferenced  // (COMPLIANT)  delete s;    } |
|  |

| **Principles:**  **Use Effective Quality Assurance Techniques -** This principle is directly related to our coding standard STD-005-CPP (refrain from accessing dangling pointers) since quality assistance teams or automated quality assurance methods should be taken (such as the writing of tests) to check for situations where dangling pointers could be accessed, which could result in undefined behavior (Britton & Pincar, 2023).  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure that dangling pointers are not accessed is an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | UNLIKELY | HIGH | MEDIUM | SL 1 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **24.04** | **dangling\_pointer\_use**  [**MEM30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)  “[Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory)” (Gennari & Shrivastava, 2024). | Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘dangling\_pointer\_use’ checker ensures that freed memory is not accessed through the use of dangling pointers (Gennari & Shrivastava, 2024). |
| **Axivion Bauhaus Suite** | **7.2.0** | **CertC-MEM30**  [**MEM30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)  “[Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory)” (Gennari & Shrivastava, 2024). | Axivion’s ‘CertC-MEM30’ checker ensures that freed memory is not accessed through the use of dangling pointers (Gennari & Shrivastava, 2024). |
| **Coverity** | **2017.07** | **USE\_AFTER\_FREE**  [**MEM30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)  “[Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory)” (Gennari & Shrivastava, 2024). | Coverity’s ‘USE\_AFTER\_FREE’ checker helps to ensure that freed memory is not accessed through the use of dangling pointers (Gennari & Shrivastava, 2024). |
| **CppCheck** | **2.15** | **danglingLifetime, returnDanglingLifetime, autoVariables, and invalidLifetieme**  [**DCL30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)  “[Declare objects with appropriate storage durations](https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations)” (Seacord & Shrivastava, 2024).  **doubleFree, deallocret and deallocuse**  [**MEM30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)  “[Do not access freed memory](https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory)” (Gennari & Shrivastava, 2024). | CppCheck’s **‘danglingLifetime,’ ‘returnDanglingLifetime,’ ‘autoVariables,’ ‘invalidLifetieme’** **‘doubleFree,’ ‘deallocret,’ and ‘deallocuse’** checkers help to ensure that objects are declare with appropriate durations for storage, and that freed memory is not accessed through the use of dangling pointers (Gennari & Shrivastava, 2024). |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use Assertions to Diagnose Errors** |
| --- | --- | --- |
| **Assertions** | STD-006-  CPP | Using assertions helps developers and engineers to identify potential errors and bugs early on in the development process (GeeksforGeeks, 2022). Assertions can be used to determine the expected state of code before it starts running, or to determine the state of code after it has finished running (GeeksforGeeks, 2022). This makes them a useful tool for regular testing during the development process so that issues and bugs can be addressed and corrected as early as possible in the development process. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code segment below, the kernel space sends data to the user space (Bhadrinath & Shrivastava, 2024). While this method safely copies bites to the user space, the use of padding bites may lead to leakage of potentially sensitive information upon copying data to the user space (Bhadrinath & Shrivastava, 2024). |
| #include <stddef.h>  struct test {  int a;  int b;  int c;  };  // Copy bites to the user space  extern int copy\_to\_user(void \*dest, void \*src, size\_t size);  // This function should be steralizing data  // before copying it to an untrusted context  // (NONCOMPLIANT)  void do\_something(void \*usr buf) {  struct test arg = {.a = 1, .b = 2, .c = 3};  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  }  (Bhadrinath & Shrivastava, 2024). |

| **Compliant Code** |
| --- |
| By sterilizing the structure data before it is copied to an untrusted context (Bhadrinath & Shrivastava, 2024), vulnerabilities such as the possibility of information leakage is mitigated. |
| // Include the ‘assert’ header.  #include <assert.h>  #include <stddef.h>    struct test {  int a;  char b;  char padding\_1, padding\_2, padding\_3;  int c;  };    // Copy bytes to user space.  extern int copy\_to\_user(void \*dest, void \*src, size\_t size);    void do\_something(void \*usr\_buf) {  // Ensure that c is the next byte after the last padding byte  // with ‘static\_assert’  // (COMPLIANT)  static\_assert(offsetof(struct test, c) == offsetof(struct test, padding\_3) + 1, "Structure contains intermediate padding");    // Ensure there is no trailing padding  // (COMPLIANT)  static\_assert(sizeof(struct test) == offsetof(struct test, c) + sizeof(int), "Structure contains trailing padding");  struct test arg = {.a = 1, .b = 2, .c = 3};  arg.padding\_1 = 0;  arg.padding\_2 = 0;  arg.padding\_3 = 0;  copy\_to\_user(usr\_buf, &arg, sizeof(arg));  }  (Bhadrinath & Shrivastava, 2024). |

| **Principles:**  **Use Effective Quality Assurance Techniques -** This principle is directly related to our coding standard STD-006-CPP (use assertions to diagnose errors) since quality assurance will involve the use of assertions for error discovery and handling.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of code review and other quality assurance methods to assure the use of assertions to diagnose errors will be an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **24.04** | **bad-function** and **bad-macro-use**  [**ERR06-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152296)  “[Understand the termination behavior of assert() and abort()](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152296)” (Herter & Volkovitsky, 2024). | Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘bad-function’ and ‘bad-macro-use’ checkers esure the correct use of the ‘assert()’ and ‘abort()’ functions (Herter & Volkovitsky, 2024). |
| **Axivion Bauhaus Suite** | **7.2.0** | **CertC-DCL03**  [**DCL03-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152090)  “[Use a static assertion to test the value of a constant expression](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression)” (Britton & Seacord, 2018). | The Avaxion Bauhaus Suite’s ‘CertC-DCL03’ checker helps to ensure the correct use of [static assertion to test the value of a constant expression](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression)” (Britton & Seacord, 2018). |
| **Coverity** | **2017.07** | **ASSERT\_SIDE\_EFFECT**  [**MSC11-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152346)  “[Incorporate diagnostic tests using assertions](https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions)” (Rosenau & Seacord, 2018). | Coverity’s ‘ASSERT\_SIDE\_EFFECT’ checker helps to ensure that assertions are used to perform diagnostic testing (Rosenau & Seacord, 2018). |
| **Parasoft** | **2024.2** | **CERT.MSC60.ASSERT**  [**MSC60-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487376)  “[Do not use assertions to verify the absence of runtime errors](https://wiki.sei.cmu.edu/confluence/display/java/MSC60-J.+Do+not+use+assertions+to+verify+the+absence+of+runtime+errors)” (Long & Rosenau, 2023). | Parasoft’s ‘**CERT.MSC60.ASSERT**’ checker helps to assure that assertions are not used “[to verify the absence of runtime errors](https://wiki.sei.cmu.edu/confluence/display/java/MSC60-J.+Do+not+use+assertions+to+verify+the+absence+of+runtime+errors)” (Long & Rosenau, 2023). |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Use ‘try,’ ‘catch,’ and ‘throw’ statements to catch exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-  CPP | Exceptions are the result of abnormal conditions which take place during program execution (GeeksforGeeks, 2025). Exception handling can be used for a variety of purposes including the transfer of control to other functions, the throwing of descriptive error messages upon gracefully exiting a program, and the handling of exceptions without exiting the program (GeeksforGeeks, 2025). The ‘std::abort(),’ ‘std::quick\_exit(),’ and ‘std::\_Exit()’ functions terminate a program immediately without calling exit handlers registered, “and without executing destructors for objects with automatic, thread, or static storage duration” (Ballman, 2016). ‘Try,’ ‘catch,’ and ‘throw’ statements perform other error handling tasks such as performing specific actions when specific exceptions are caught or throwing exceptions inside of ‘try’ blocks (GeeksforGeeks, 2025).  In using the types of statements mentioned above, developers should avoid the hiding of errors and exceptions by having programs continue on when errors or exceptions are caught. Exception handling should be conducted in ways which make the occurrence of an error exception clear to other developers and users, while also allowing the program to continue running when helpful or necessary. Situation-specific custom error messages should also be used where appropriate in order to make the cause of errors and exceptions abundantly clear to observers. |
|  |  |  |

| **Noncompliant Code** |
| --- |
| In the noncompliant code segment below, the ‘f()’ function is made an exit handler by the inclusion of ‘std::at\_exit().’ The ‘f()’ function might result in a call to ‘sed::terminate()’ since calling the ‘throwing\_func’ function might cause an exception to be thrown (Ballman, 2016). |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  // Rather than using a ‘try’ statement, the function exits the  // program if a condition with in an ‘if’ statement is met  // (NONCOMPLIANT)  int main() {  if (0 != std::atexit(f)) {  }  }  (Ballman, 2016) |

| **Compliant Code** |
| --- |
| In the compliant code segment below, the ‘f()’ function catches any exceptions which might be thrown by the ‘throwing\_func()’ function (Ballman, 2016). |
| #include <cstdlib>  void throwing\_func() noexcept(false);  void f()  {  // The use of a ‘try’ statement here allows the  // program to attempt to use the ‘throwing\_func’  // function,  // (COMPLIANT)  try {  throwing\_func();  }  // and any errors which are thrown by the  // ‘throwing\_func’ function will be caught  // within the ‘f()’ function  // (COMPLIANT)  catch (...)  {  }  }  int main() {  if (0 != std::atexit(f)) {  }  }  (Ballman, 2016). |

| **Principles:**  **Use Effective Quality Assurance Techniques -** This principle is directly related to our coding standard STD-007-CPP (Use ‘try,’ ‘catch,’ and ‘throw’ statements to catch exceptions) since quality assurance should definitely involve writing tests to verify the correct use of ‘try,’ ‘catch,’ and ‘throw’ statements.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of code review and other quality assurance methods to assure the use of ‘try,’ ‘catch,’ and ‘throw’ statements will be an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Axivion Bauhaus Suite** | **7.2.0** | **CertC-SIG35**  [**SIG35-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152239)  “[Do not return from a computational exception signal handler](https://wiki.sei.cmu.edu/confluence/display/c/SIG35-C.+Do+not+return+from+a+computational+exception+signal+handler)” (Shrivastava, 2024). | The Avaxion Bauhaus Suite’s ‘**CertC-SIG35**’ checker helps to ensure that computation exception signal handlers are not returned from (Shrivastava, 2024). |
| **Coverity** | **7.5** | **MISSING\_THROW**  [**ERR00-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487876)  “[Do not suppress or ignore checked exceptions](https://wiki.sei.cmu.edu/confluence/display/java/ERR00-J.+Do+not+suppress+or+ignore+checked+exceptions)” (Mohindra & O’Donnel, 2021). | Coverity’s ‘MISSING\_THROW’ checker ensures that exceptions are not ignored or suppressed (Mohindra & O’Donnel, 2021). |
| **CodeSonar** | **8.3p0** | **LANG.STRUCT.RFCESH** [SIG35-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152239) “[Do not return from a computational exception signal handler](https://wiki.sei.cmu.edu/confluence/display/c/SIG35-C.+Do+not+return+from+a+computational+exception+signal+handler)” (Shrivastava, 2024). | ConeSonar’s ‘LANG.STRUCT.RFCESH’ checker helps to ensure that computation exception signal handlers are not returned from (Shrivastava, 2024). |
| **Parasoft** | **2024.2** | **CERT.ERR00.LGE**  **and CERT.ERR00.UCATCH**  [**ERR00-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487876)  “[Do not suppress or ignore checked exceptions](https://wiki.sei.cmu.edu/confluence/display/java/ERR00-J.+Do+not+suppress+or+ignore+checked+exceptions)” (Mohindra & O’Donnel, 2021).  **CERT.ERR01.ACPST, CERT.ERR01.CETS, and CERT.ERR01.ACW**  [**ERR01-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487679)  “[Do not allow exceptions to expose sensitive information](https://wiki.sei.cmu.edu/confluence/display/java/ERR01-J.+Do+not+allow+exceptions+to+expose+sensitive+information)” (Mohindra & Rozenau, 2021).  **CERT.ERR05.ARCF**  **and**  **CERT.ERR05.ATSF**  [**ERR05-J**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487445)  “[Do not let checked exceptions escape from a finally block](https://wiki.sei.cmu.edu/confluence/display/java/ERR05-J.+Do+not+let+checked+exceptions+escape+from+a+finally+block)” (Mohindra & Rosenau, 2021). | Parasoft’s ‘CERT.ERR00.LGE,’ ‘CERT.ERR00.UCATCH,’ ‘CERT.ERR01.ACPST,’ ‘CERT.ERR01.CETS,’ ‘CERT.ERR01.ACW,’ ‘CERT.ERR05.ARCF,’ and  ‘CERT.ERR05.ATSF’ checkers help prevent the suppression or the ignoring of exceptions (Mohindra & O’Donnel, 2021). Parasoft’s ‘ERR01-J’ and ‘ERR05-J’ checkers help prevent exceptions from exposing sensitive information, and help prevent the escape of checked exceptions from a ‘finally’ block respectively (Mohindra & Rosenau, 2021). |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do Not Invoke Virtual Functions From Constructors or Destructors** |
| --- | --- | --- |
| **Object Oriented Programming** | STD-008-  CPP | Use inheritance and overriding (Ballman, 2016) to support code reusability, scalability, modularity and code organization (Unstop, 2025). Directly invoking virtual functions from the constructors and destructors used in object oriented programming is dangerous because construction begins with the base class and then moves to derived classes, and destruction begins with more derived classes and then moves toward the base class (Ballman, 2016). |

| **Noncompliant Code** |
| --- |
| In the noncompliant code segment below, the base class tries to use object resources by calling virtual functions from the constructor and the destructor (Ballman, 2016). The ‘B::B()’ constructor then calls ‘B::seize()’ rather than ‘D::seize()’ and the ‘B::~B()’ destructor calls ‘B::release()’ rather than ‘D::release().’ |
| struct B {  B() { seize(); }  virtual ~B() { release(); }  protected:  virtual void seize();  virtual void release();  };  struct D : B {  virtual ~D() = default;  protected:  void seize() override {  // Calls a virtual function  // (NONCOMPLIANT)  B::seize();  }  void release() override {  // Calls a virtual function  // (NONCOMPLIANT)  B::release();  }  };  (Ballman, 2016). |

| **Compliant Code** |
| --- |
| In the compliant code segment below, constructors and destructors call a private and non-virtual member function rather than calling a virtual function (Ballman, 2016). Classes are then responsible for seizing and releasing their own resources (Ballman, 2016). |
| class B {  void seize\_mine();  void release\_mine();  public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }  protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };  class D : public B {  void seize\_mine();  void release\_mine();  public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }  protected:  void seize() override {  // Calls a private and non-virtual  // member function  // (COMPLIANT)  B::seize();  seize\_mine();  }  void release() override {  release\_mine();  // Calls a private and non-virtual  // member function  // (COMPLIANT)  B::release();  }  };  (Ballman, 2016). |

| **Principles:**  **Heed Compiler Warnings -** The calling of virtual functions from constructors or destructors is likely to set off a compiler warning, which gives engineers and developers an additional way to assure that this standard is adhered to.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing and other quality assurance methods to assure that virtual functions are not invoked from constructors and destructors will be an important part of our Defense-in-Depth approach to security.  **Use Effective Quality Assurance Techniques -** This principle is directly related to our coding standard STD-008-CPP (Do Not Invoke Virtual Functions From Constructors or Destructors) since quality assurance should absolutely write tests to check that virtual functions are not invoked from constructors and destructors.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | MEDIUM | LOW | MEDIUM | SL 1 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **22.10** | **virtual-call-in-constructor**  **and**  **invalid\_function\_pointer**  [**OOP50-CPP**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570)  “[Do not invoke virtual functions from constructors or destructors](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors)” (Britton & Dewhurst, 2023) | Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘virtual\_call\_in\_constructor’ and ‘invalid\_function\_parameter’ checkers help to ensure that virtual functions are not invoked from constructors or destructors (Britton & Dewhurst, 2023). |
| **Axivion Bauhaus Suite** | **7.9** | **CertC++-OOP50**  [**OOP50-CPP**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570)  “[Do not invoke virtual functions from constructors or destructors](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors)” (Britton & Dewhurst, 2023) | The Avaxion Bauhaus Suite’s ‘CertC++-OOP50’ checker helps to ensure that virtual functions are not invoked from constructors or destructors (Britton & Dewhurst, 2023). |
| **CodeSonar** | **8.3p0** | **LANG.STRUCT.VCALL\_IN\_CTOR**  **and LANG.STRUCT.VCALL\_IN\_DTOR**  [**OOP50-CPP**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570)  “[Do not invoke virtual functions from constructors or destructors](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors)” (Britton & Dewhurst, 2023) | CodeSonar’s ‘LANG.STRUCT.VCALL\_IN\_CTOR’  and ‘LANG.STRUCT.VCALL\_IN\_DTOR’ checkers help to ensure that virtual functions are not invoked from constructors or destructors (Britton & Dewhurst, 2023). |
| **Parasoft** | **2024.2** | **CERT\_CPP-OOP50-a, CERT\_CPP-OOP50-b, CERT\_CPP-OOP50-c,**  **and**  **CERT\_CPP-OOP50-d**  [**OOP50-CPP**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570)  “[Do not invoke virtual functions from constructors or destructors](https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors)” (Britton & Dewhurst, 2023) | Parasoft’s ‘**CERT\_CPP-OOP50-a,’ ‘CERT\_CPP-OOP50-b,’ ‘CERT\_CPP-OOP50-c,’**  **and ‘CERT\_CPP-OOP50-d**’ checkers help to ensure that virtual functions are not invoked from constructors or destructors (Britton & Dewhurst, 2023). |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Prevent User Input Format String Vulnerabilities** |
| --- | --- | --- |
| **Input Output** | STD-009-  CPP | Exclude user input strings used for formatting (Ballman, 2016) and configuration so that users are not able to adjust them with tainted input values (Ballman, 2016). Giving attackers the ability to control format and configuration strings can allow them to execute malicious code by using the permissions of the functions where the vulnerabilities exist (Ballman, 2016). Formatted output functions can even be used to write integer values to specific addresses with the ‘%n’ conversion specifier (Ballman, 2016). |

| **Noncompliant Code** |
| --- |
| In the noncompliant code section below, the ‘incorrect\_password()’ function is called during identification and authentication so that an error message is displayed if the username is not found or if the password does not correspond to the username (Ballman, 2016). The function accepts the string entered by the user as a username and constructs an error message (Ballman, 2016). The “addition’ operators are never checked for integer overflow, and since the ‘%s’ characters are replaced by the string entered by the user, the string contents fit narrowly into memory (Ballman, 2016). The code causes a vulnerability since ‘msg’ consists of untrusted user input (Ballman, 2016). |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  // User names are restricted to 256 or fewer characters  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  // Handle error  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  // Handle error  }  else if (ret >= len) {  // Handle truncated output using ‘fprintf’  // (NONCOMPLIANT)  }  fprintf(stderr, msg);  free(msg);  }  (Ballman, 2016). |

| **Compliant Code** |
| --- |
| In the complaint segment of code, the issue is resolved by using ‘fputs()’ rather than ‘fprintf(),’ which means that msg will be output directly to ‘stderr’ without evaluating the contents (Ballman, 2016). |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  // User names are restricted to 256 or fewer characters  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  // Handle error  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  // Handle error  } else if (ret >= len) {  // Handle truncated output using ‘fputs’  // (COMPLIANT)  }  fputs(msg, stderr);  free(msg);  }  (Ballman, 2016). |

| **Principles:**  **Validate Input Data -** This principle is directly related to our coding standard STD-009-CPP (Prevent User Input Format String Vulnerabilities) since validating input data is an important mitigation strategy for avoiding vulnerabilities and cyberattacks involving user input.  **Default Deny -** Upon receiving string input data with potentially malicious characters or strings included within, the user’s attempt to have their input used by the system will be denied. This will still take place in instances of accidental or unintentional entry of potentially malicious input since it is difficult to account for every possible attack attempt involving user input.  **Sanitize Data Sent to Other Systems -** This principle is directly related to our coding standard STD-009-CPP (Prevent User Input Format String Vulnerabilities) since user input is a type of data which could be sent to other systems, and such data must be sanitized in order for an organization to avoid being used as an injection method for malicious user input to infect other systems.  **Adhere to the Principle of Least Privilege -** Not all users should have the ability to execute SQL commands to create, read, update or delete data from our databases. Keeping input fields from providing malicious actors with a way to execute SQL commands is incredibly important in limiting user privileges to only what is needed.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of static testing, code review, and other quality assurance methods to assure user input from being used to set the values of format strings will be an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy.  **Use Effective Quality Assurance Techniques -**  This principle is directly related to our coding standard STD-009-CPP (Prevent User Input Format String Vulnerabilities) since the presence of vulnerabilities related to user input (such as a lack of sufficiently comprehensive input validation) should be identified and flagged during the quality assurance processes. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | MEDIUM | HIGH | HIGH | SL 3 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| **Axivion Bauhaus Suite** | **7.2.0** | **CertC-FIO30** [FIO30-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152197) “[Exclude user input from format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings)” (Burch & Shrivastava).  **CertC-FIO47**  [**FIO47-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152167)  “[Use valid format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO47-C.+Use+valid+format+strings)” (Pincar & Shrivastava, 2024). | The Avaxion Bauhaus Suite’s ‘CertC-FIO47’ checker assures the use of valid format strings, and its ‘CertC-FIO30’ checker helps to ensure that user input is excluded from format strings (Burch & Shrivastava). |
| **Coverity** | **2017.07** | **TAINTED\_STRING** [FIO30-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152197) “[Exclude user input from format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings)” (Burch & Shrivastava).  **PW**  [**FIO47-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152167)  “[Use valid format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO47-C.+Use+valid+format+strings)” (Pincar & Shrivastava, 2024). | Coverity’s ‘TAINTED\_STRING’ checker assures the use of valid format strings, and its ‘PW’ checker helps to ensure that user input is excluded from format strings (Burch & Shrivastava). |
| **CppCheck** | **2.15** | **invalidscanf,**  **wrongPrintfScanfArgNum, invalidLengthModifierError,**  **invalidScanfFormatWidth,**  **and wrongPrintfScanfParameterPositionError**  [**FIO47-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152167)  “[Use valid format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO47-C.+Use+valid+format+strings)” (Pincar & Shrivastava, 2024). | CppCheck’s ‘invalidscanf,’  ‘wrongPrintfScanfArgNum,’ ‘invalidLengthModifierError,’  ‘invalidScanfFormatWidth,’  and ‘wrongPrintfScanfParameterPositionError’ checkers help to ensure the use of valid format strings (Pincar & Shrivastava, 2024). |
| **Parasoft** | **2024.2** | **CERT\_C-FIO30-a, CERT\_C-FIO30-b,**  **and**  **CERT\_C-FIO30-c** [FIO30-C](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152197) “[Exclude user input from format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+strings)” (Burch & Shrivastava).  **CERT\_C-FIO47-a, CERT\_C-FIO47-b, CERT\_C-FIO47-c, CERT\_C-FIO47-d, CERT\_C-FIO47-e,**  **and**  **CERT\_C-FIO47-f**  [**FIO47-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152167)  “[Use valid format strings](https://wiki.sei.cmu.edu/confluence/display/c/FIO47-C.+Use+valid+format+strings)” (Pincar & Shrivastava, 2024). | Parasoft’s ‘CERT\_C-FIO47-a,’ ‘CERT\_C-FIO47-b,’ ‘CERT\_C-FIO47-c,’ ‘CERT\_C-FIO47-d,’ ‘CERT\_C-FIO47-e,’ and ‘CERT\_C-FIO47-f’ checkers help to ensure the use of valid format strings.  Parasoft’s ‘CERT\_C-FIO30-a,’  ‘CERT\_C-FIO30-b,’  and  ‘CERT\_C-FIO30-c’ checkers help to ensure that user input is excluded from format strings (Burch & Shrivastava). |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do Not Access an Object Outside of Its Lifetime** |
| --- | --- | --- |
| **Expressions** | STD-0010-  CPP | An object’s lifetime begins when sufficient and aligned storage has been obtained for it completing its initialization (Ballman, 2016). The object’s lifetime ends when a nontrivial destructor is called for the object and the object’s storage has been released or reused to store other data (Ballman, 2016). The use of objects or object pointers outside of their lifetime i likely to result in undefined behavior (Ballman, 2016). |

| **Noncompliant Code** |
| --- |
| In the noncompliant code section below, a pointer object calls a non-static member function of the same object before the pointer’s lifetime begins (Ballman, 2016). This increases the possibility that undefined behavior wil occur (Ballman, 2016). |
| struct S {  void mem\_fn();  };    void f() {    // The pointer object calls the mm\_fn() function  // before the pointer’s lifetime has begun  // (NONCOMPLIANT)  S \*s;  s->mem\_fn();  }  (Ballman, 2016). |

| **Compliant Code** |
| --- |
| In the compliant code section below, the pointer’s storage is obtained before the ‘S::mem\_fn()’ function is called (Ballman, 2016). |
| struct S {  void mem\_fn();  };    void f() {    // The pointer’s storage is obtained  // (the pointer’s lifetime has begun)  // before the mm\_fn() function is called  // (COMPLIANT)  S \*s = new S;  s->mem\_fn();  delete s;  }  (Ballman, 2016). |

| **Principles:**  **Use Effective Quality Assurance Techniques -** This principle is directly related to our coding standard STD-010-CPP (Do Not Access an Object Outside of its Lifetime) since quality assurance will write a series of tests designed to identify where and when code could result in the accessing of an object outside of its lifetime.  **Practice Defense in Depth -** Defense-in-Depth is a “security strategy integrating people, technology, and operations capabilities to establish variable barriers across multiple layers and missions of the organization” (National Institute of Standards and Technology, n. d.). The use of code review and other quality assurance methods to ensure that objects are not accessed outside of their lifetime will be an important part of our Defense-in-Depth approach to security.  **Adopt a Secure Coding Standard -** Adherence to this coding standard will form part of one’s adherence to the overall security policy. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | LOW | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428) | **24.04** | **temporary-object-modification**  [**EXP35-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152058)  “[Do not modify objects with temporary lifetime](https://wiki.sei.cmu.edu/confluence/display/c/EXP35-C.+Do+not+modify+objects+with+temporary+lifetime)” (Seacord & Shrivastava, 2024). | Astr[é](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152428)e’s ‘temporary-object-modification’ checker helps to ensure that objects with temporary lifetimes are not modified (Seacord & Shrivastava, 2024). |
| **Axivion Bauhaus Suite** | **7.9** | **CertC-EXP35**  [**EXP35-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152058)  “[Do not modify objects with temporary lifetime](https://wiki.sei.cmu.edu/confluence/display/c/EXP35-C.+Do+not+modify+objects+with+temporary+lifetime)” (Seacord & Shrivastava, 2024). | The Avaxion Bauhaus Suite’s ‘CertC-EXP35’ checker helps to ensure that objects with temporary lifetimes are not modified (Seacord & Shrivastava, 2024). |
| **Coverity** | **2024.12.0** | **RETURN\_LOCAL**  [**DCL30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)  “[Declare objects with appropriate storage durations](https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations)” (Seacord & Shrivastava, 2024). | Coverity’s ‘RETURN\_LOCAL’ checker helps to ensure that objects are declared with appropriate storage durations (Seacord & Shrivastava, 2024). |
| **CppCheck** | **2.15** | **danglingLifetime, returnDanglingLifetime, autoVariables, and invalidLifetime**  [**DCL30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)  “[Declare objects with appropriate storage durations](https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations)” (Seacord & Shrivastava, 2024). | CppCheck’s ‘**danglingLifetime,’ ‘returnDanglingLifetime,’ ‘autoVariables,’ and ‘invalidLifetime’**  checkers help to ensure that objects are declared with appropriate storage durations (Seacord & Shrivastava, 2024). |
| **Parasoft** | **2024.2** | **CERT\_C-DCL30-a**  **and**  **CERT\_C-DCL30-b**  [**DCL30-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)  “[Declare objects with appropriate storage durations](https://wiki.sei.cmu.edu/confluence/display/c/DCL30-C.+Declare+objects+with+appropriate+storage+durations)” (Seacord & Shrivastava, 2024).  **CERT\_C-EXP35-a**  [**EXP35-C**](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152058)  “[Do not modify objects with temporary lifetime](https://wiki.sei.cmu.edu/confluence/display/c/EXP35-C.+Do+not+modify+objects+with+temporary+lifetime)” (Seacord & Shrivastava, 2024). | Parasoft’s ‘CERT\_C-DCL30-a’ and ‘CERT\_C-DCL30-b’ checkers help to ensure that objects are declared with appropriate storage durations (Seacord & Shrivastava, 2024).  Parasoft’s ‘CERT\_C-EXP35-a’ checker helps to ensure that objects with temporary lifetimes are not modified (Seacord & Shrivastava, 2024). |

### Defense-in-Depth Illustration



### Automation



In order to automate the enforcement of compliance to the standards and principles described herein, the tools and techniques described throughout this document must be integrated into the organization’s existing DevOps processes and infrastructure. After conducting research into the tools which are commonly used in the industry to meet the standards described in this document, the tools which will be used for automation throughout the DevSecOps process are Astrée, Axivion Bauhaus Suite, CodeSonar, Coverity, CppCheck, and Parasoft. Static code testing will be a major aspect of our DevSecOps infrastructure, and the use of each of these static testing tools (specifically of the versions specified under each of the standards described in this document) will ensure the enforcement of each of the standards and principles described above among our organization’s teams. Quality Assurance professionals will be responsible for periodically using these tools to review and test code as it is written. The frequency of the use of these tools on developed code will depend on many situation-specific variables.

The static testing tools described herein are used during the ‘Build’ and the ‘Verify and Test’ stages depicted above. While the second of these phases is focused exclusively on verifying and testing code, static testing tools should be used throughout the development (Build) process to ensure exhaustive and redundant testing measures in adherence to Defense-in-Depth principles (National Institute of Standards and Technology, n. d.). Developers should use CppCheck throughout the coding process, and should use other tools . Comprehensive and redundant testing will help to ensure that the code which makes up our systems is secure and bug-free, and that sufficient measures have been taken to enforce adherence to this security policy. Since Axivion Bauhaus Suite, Coverity, and CppCheck cover the majority of the standards described in this document with significant redundancy (these three tools check for a wide variety of common issues related to each standard with significant overlap), developers and engineers should be trained to use these tools so that they can perform static testing on their own code during the ‘Build’ phase of development process. These same three tools will also be used by testers and quality assurance professionals during the ‘Verify and Test’ phase, along with CodeSonar, Parasoft and Astrée for further redundancy and for the detection of specific cases less comprehensively covered by the first three tools.

Principle 3 (Architect and Design for Security Policies) highlights the importance of taking secure coding practices into consideration during early stages of the development process (or of an iteration of the process). Although code is written during the ‘Build’ and the ‘Verify and Test’ phases of development, decisions which are made during the ‘Assess and Plan’ and the ‘Design’ stages of iterative development must take secure coding practices into consideration. For example, design documents such as UML class diagrams should adhere to best practices related to data type selection in adherence to standard STD-001-CPP (Use Appropriate Data Types).

In the interest of making our systems able to share data with different types of devices with different operating systems (not only Windows) securely, our system will use both the LDAP protocol and Microsoft's Active Directory (AD) (TutorialsPoint, 2025). Since LDAP is platform independent and AD is Microsoft-specific, Microsoft’s Active Directory will be used to communicate with Windows machines while the LDAP protocol will be used to share data across non-windows networks.

Having automated the process of enforcing adherence to our security policy using the tools described above, Our DevSecOps processes and infrastructure will improve our current development practices so that we can create secure systems written in C++.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | HIGH | UNLIKELY | MEDIUM | HIGH | SL 2 (Fortinet, 2025). |
| STD-002-CPP | MEDIUM | UNLIKELY | MEDIUM | MEDIUM | SL 2 (Fortinet, 2025). |
| STD-003-CPP | HIGH | UNLIKELY | MEDIUM | HIGH | SL 2 (Fortinet, 2025). |
| STD-004-CPP | HIGH | MEDIUM | MEDIUM | HIGH | SL 3 (Fortinet, 2025). |
| STD-005-CPP | MEDIUM | UNLIKELY | HIGH | MEDIUM | SL 1 (Fortinet, 2025). |
| STD-006-CPP | MEDIUM | MEDIUM | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |
| STD-007-CPP | MEDIUM | MEDIUM | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |
| STD-008-CPP | MEDIUM | MEDIUM | LOW | MEDIUM | SL 1 (Fortinet, 2025). |
| STD-009-CPP | HIGH | MEDIUM | HIGH | HIGH | SL 3 (Fortinet, 2025). |
| STD-010-CPP | MEDIUM | LOW | MEDIUM | MEDIUM | SL 1 (Fortinet, 2025). |

### Policies for Encryption and Triple A

| **Encryption** | **“Encryption is a form of** [**data security**](https://www.fortinet.com/resources/cyberglossary/data-security) **in which information is converted to ciphertext. Only authorized people who have the key can decipher the code and access the original plaintext information” (Fortinet, 2025). The encryption of data is an important part of data security, and systems where data is not encrypted should not be trusted with potentially sensitive data. In order to ensure data security throughout our systems, data should be encrypted while it is at rest, while it is in flight, and while it is in use.** |
| --- | --- |
| Encryption at rest | ‘Encryption at rest’ refers to the encryption of data while it is being stored (IBM, 2025) in a data store or database. Encrypting data while it is at rest within a system adds a layer of security by making stored data uninterpretable without a decryption key (IBM, 2025). Our policy for encryption at rest will involve implementing encryption when it is generated or input (Raza, 2023), and storing the encrypted data so that it will only be decrypted when it needs to be accessed. This will make it much harder for our data to be compromised even if an attacker is able to gain some level of access to the system. |
| Encryption in flight | ‘Encryption in flight’ refers to the encryption of data when it is sent from one place to another (IBM, 2025). Encrypting data while it is in transit helps to ensure secure communication between system components or with external systems and devices. Our policy for encryption in flight will involve "encrypting data that is transferred between two nodes of the network” (Raza, 2023) or that is sent to or received from external systems or devices. The policy will involve the use of SSL(TLS) certificates, and the use of HTTPS (secure hypertext transfer protocol) for communication across the internet (Google Cloud, n. d.). |
| Encryption in use | ‘Encryption in use’ refers to the encryption of data which is in memory at runtime (Google Cloud, n. d.). The encryption of data while it is in memory during runtime helps to further reduce the possibility of data being compromised. Our encryption in use policy will consist of hardware-based encryption and partial or full homomorphic encryption, “which allows data to be processed while it’s still encrypted” (SentinelOne, 2024). Since full homomorphic encryption is more computationally intensive (Velimirovic, 2023), partial homomorphic encryption may be used for less sensitive data in use while full homomorphic encryption is be used for more sensitive data while it is in use by a system component. |

| **Triple-A Framework** | **Triple a is “a security framework that controls access to computer resources, enforces policies, and audits usage…its combined processes play a major role in…cybersecurity by screening users and keeping in track of their activity while they are connected (Fortinet, 2025). The three As in Triple-A stand for authentication, authorization, and accounting.** |
| --- | --- |
| Authentication | Authentication involves prompting system users for information which proves that they are who they say that they are (Fortinet, 2025). User login credentials, for example, tell the system that a specific user is logging in. The system trusts that the person logging in is authorized to use that account to access the system due to the fact that the user has entered the correct password which, ideally, only said user has. Our authentication policy will involve user authentication via the usual method of giving every user a username and password (the user is able to change their password later if they choose), and using multi-factor authentication (MFA) as an additional layer of security. MFA will also be used for access to any third-party services used by our organization such as cloud providers for example. Additionally, user authentication consisting of password re-entry should take place when a user attempts to access or make changes to our databases. The addition of new users should also involve authentication via one of the new user’s personal devices such as a smartphone in order to prevent the addition of users through unauthorized system manipulation. The strength of our authentication policy for user logins will help to prevent unauthorized access to our systems and will allow us to implement access control in accordance with Principle 6 described above (‘Adhere to the Principle of Least Privilege’). |
| Authorization | “Authorization follows authentication…a user can be granted privileges to access certain areas of a network or system…authorization dictates what a user is allowed to do” (Fortinet, 2025). After user authentication takes place, the implementation of our authorization policy will prevent users from accessing data and methods which are irrelevant to their function within the system. Only administrative users (admin) will be granted access to the entire system. Other types of users, whether institutional or external, will only be able to access parts of the system which they need to access in order to achieve their goals within the system. Each user should also only be able to access their own data, and there should be no way for users to access data related to other users (unless said data is required for them to achieve their goals within the system). The strength of our Authorization policy will prevent unauthorized access to data and functions which could result in compromised data or even damage to system components. |
| Accounting | “Accounting keeps track of user activity…such as how long they were logged in, the data they sent or received, their…(IP) address, the…(URI) they used, and the different services they accessed” (Fortinet, 2025). Effective accounting enhances the ability for administrators to prevent unauthorized access, and can be used to block users suspected of repetitively attacking the system purposefully. Input which contains data indicating a potential attack should be documented, and the system should be automated to respond accordingly. For example, a system or system component must check for and document certain combinations of the ‘-,’ ‘(,’ ‘>,’ or ‘=’ characters within an input string, since certain arrangements or uses of these characters could indicate that the user is likely attempting a SQL injection attack. The repeated entry of numerical data types (int, float, etc) which are far too large for their use-case must also be documented. User attempts to repeatedly access files with sensitive data will also be accounted for, and the action taken in this scenario will depend on the type of user, on the type of data being accessed, and on the specific situation in general. Effective accounting to track user interaction with the system mitigates against certain types of attacks by blocking suspected attackers from continuing to interact with the system. |

### Map the Principles

Each of the ten principles defined in the ‘Ten Core Security Principles’ section applies directly to one or several of the ten standards defined in this document. Principle 1 (Validate Input Data) relates to Standard STD-001-CPP (Use Appropriate Data Types) in that the use of appropriate data types prevents potentially dangerous input from entering the system (Scheila, 2024). The same principle applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows) because the validation of input data is an effective mitigation strategy to prevent successful buffer overflow attack attempts, and to Standard STD-004-CPP since the prevention of successful SQL injection attempts is an important reason to validate input data. Principle 1 also applies directly to Standard STD-009-CPP since validating input data is an important mitigation strategy for avoiding vulnerabilities and cyberattacks involving user input.

Principle 2 (Heed Compiler Warnings) applies directly to Standard STD-002-CPP (Always Initialize Variables) since variables which are not initialized properly within scope are likely to set off compiler warnings, which gives engineers and developers an additional way to assure that this standard is adhered to. The same principle also directly applies to Standard STD-008-CPP (Do Not Invoke Virtual Functions From Constructors or Destructors) since the calling of virtual functions from constructors or destructors is also likely to set off a compiler warning, which gives engineers and developers an additional way to assure that this standard is adhered to.

Principle 3 (Architect and Design for Security Policies) applies directly to Standard STD-001-CPP (Use Appropriate Data Types) since it is important to assure the use of correct data types as early as possible in the production process so that early-phase technical design documents help engineers and developers to adhere to best practices (such as the use of appropriate data types) in later stages. This is the only principle which explicitly describes what should take place at the earliest stages of each iteration of the software development lifecycle (when software is architected and designed). The principle is directly related only to Standard STD-001-CPP of this policy because the drafting of technical design documents (such as UML class diagrams) often defines the data types which will be used for each class.

Principle 4 (Keep it Simple) applies directly to Standard STD-001-CPP (Use Appropriate Data Types) since the use of more intuitive and less convoluted data types is often a better strategy for avoiding vulnerabilities, as is the case in the code examples included within the section of this policy which describes Standard STD-001-CPP.

Principle 5 (Default Deny) applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows), STD-004-CPP (Always Validate Input to Prevent SQL Injections), and STD-009-CPP (Prevent User Input Format String Vulnerabilities) because upon receiving input data with potentially malicious quantities, digits, operators, digits, strings, or other relevant characters or character combinations, the user’s attempt to have their input used by the system will be denied by default.

Principle 6 (Adhere to the Principle of Least Privilege) applies directly to Standard STD-004 (Always Validate Input to Prevent SQL Injections) and Principle STD-009-CPP (Prevent User Input Format String Vulnerabilities) since not all users will have the ability to execute SQL commands to create, read, update or delete data from our databases.

Principle 7 (Sanitize Data Sent to Other Systems) applies directly to Standard STD-003-CPP (Always Validate Strings to Prevent Buffer Overflows), Standard STD-004-CPP (Sanitize data sent to other systems), and Standard STD-009-CPP (Prevent User Input Format String Vulnerabilities) because user input is a type of data which could certainly be sent to other systems, and because the sanitization of user data could refer to measures meant to ensure adherence to any of these three standards.

Principle 8 (Practice Defense in Depth) applies directly to all ten of the standards defined in this policy because adherence to all of these standards will adhere to the concept of Defense-in-Depth and contribute to our comprehensive and redundant DevSecOps pipeline. Principle 9 (Use Effective Quality Assurance Techniques) applies directly to all ten of the standards defined in this policy since the use of effective quality assurance measures will help to enforce this policy by identifying areas where the standards and principles described herein are not adhered to. Principle 10 (Adopt a Secure Coding Standard) applies directly to all ten of the standards defined in this policy since each coding standard described within this policy works together to assure the security of our systems.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 01/26/2025 | Added principles, coding standards, and examples of compliant and noncompliant code. | Andrew Emilio DiStefano |  |
| 1.2 | 02/14/2025 | Finished the coding standards by adding relevant principles from the ‘Principles’ section, threat-level information.  Finished automation specifications including specific tools and the SEI CERT C++ Coding Standards which each tool addresses.  Explained how automation will be used for the enforcement of and compliance to the standards defined in this policy.  Defined policies for Encryption and Triple-A.  Mapped the principles to each of the standards, and provided a justification for the connection between the two. | Andrew Emilio DiStefano |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

**Sources:**

Ballman, A. (2016). *SEI Cert C++ Coding Standard*. Carnegie Mellon University.   
 [https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-  
 standard-2016-v01.pdf](https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-cpp-coding-standard-2016-v01.pdf)

Bhadrinath, J., & Shrivastava, S. (2024, Dec 11). [*Avoid information leakage when*](https://wiki.sei.cmu.edu/confluence/display/c/DCL39-C.+Avoid+information+leakage+when+passing+a+structure+across+a+trust+boundary)

[*passing a structure across a trust boundary*](https://wiki.sei.cmu.edu/confluence/display/c/DCL39-C.+Avoid+information+leakage+when+passing+a+structure+across+a+trust+boundary)*.* Carnegie Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/c/DCL39-C.+Avoid+information+leakage+](https://wiki.sei.cmu.edu/confluence/display/c/DCL39-C.+Avoid+information+leakage+when+passing+a+structure+across+a+trust+boundary)

[when+passing+a+structure+across+a+trust+boundary](https://wiki.sei.cmu.edu/confluence/display/c/DCL39-C.+Avoid+information+leakage+when+passing+a+structure+across+a+trust+boundary)

Britton, J., & Dewhurst, S. C. (2023, April 20). *OOP50-CPP.* Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046570>

Britton, J., & Mohindra, D. (2024, June 11). *IDS06-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487836>

Britton, J., & Pincar, J. (2023, April 20). *Do Not Access Freed Memory.* Carnegie

Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

[freed+memory](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

Britton, J., & Seacord, R. C. (2018,December 11). *DCL03-C*. Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152090)

[?pageId=87152090](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152090)

Buenning, M. (2024, October 25). *What is Input Sanitization?* NinjaOne. <https://www.ninjaone.com/it-hub/endpoint-security/what-is-input-sanitization/>

Burch, H., & Shrivastava, S. (2024, December 12). *FIO30-C*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152197>

Cppcheck. (n. d.). *CppCheck - A Tool for Static C/C++ Code Analysis.*

<https://cppcheck.sourceforge.io/>

Fortinet. (2025). *IEC-62443 Standard.*

<https://www.fortinet.com/resources/cyberglossary/iec-62443>

Fortinet. (2025). *What Is Authentication, Authorization, And Accounting*

*(AAA) Security?*

<https://www.fortinet.com/resources/cyberglossary/aaa-security>

Free Software Foundation Inc. (2025). *Options to Request or Suppress Warnings.*   
 <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

Gale, A., & Mohindra, D. (2025, January 31). *IDS03-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487909>

Gaynor, A. (2019, August 12). *Introduction to Memory Unsafety for VPs*

*of Engineering.* Alexgaynor.net.

[https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-  
 engineering/](https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/)

GeeksforGeeks. (2022, February 28). Assertions in C/C++.

<https://www.geeksforgeeks.org/assertions-cc/>

GeeksforGeeks. (2023, March 11). *Different Ways to Initialize a Variable in C++.*

<https://www.geeksforgeeks.org/different-ways-to-initialize-a-variable-in-cpp/>

GeeksforGeeks. (2025, January 11). *Exception Handling in C++.*

<https://www.geeksforgeeks.org/exception-handling-c/>

Gennari, J., & Shrivastava, S. (2024, December 11). *Do Not Read Uninitialized Memory.*

Carnegie Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)

[uninitialized+memory](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)

Gennari, J., & Shrivastava, S. (2024, December 12)*. MEM30-C.* Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)

[pageId=87152153](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152153)

GitHub. (2025). *What is Fuzzing and Fuzz Testing?*   
 <https://github.com/resources/articles/security/what-is-fuzz-testingn>

Google Cloud. (n. d.). *Confidential Computing*.

<https://cloud.google.com/security/products/confidential-computing?hl=en>

Google Cloud. (n. d.). *Encryption in Transit.*

<https://cloud.google.com/docs/security/encryption-in-transit>

Herter, J., & Volkovitsky, A. (2024, October 18). *ERR06-C.* Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152296)

[?pageId=87152296](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152296)

IBM. (2023, January 24). *What Is Penetration Testing?* <https://www.ibm.com/think/topics/penetration-testing>

IBM. (2025, January 6). *Content Encryption: In Flight and At Rest.*

[https://www.ibm.com/docs/en/aspera-on-cloud?topic=encryption-](https://www.ibm.com/docs/en/aspera-on-cloud?topic=encryption-content-in-flight-rest)

[content-in-flight-rest](https://www.ibm.com/docs/en/aspera-on-cloud?topic=encryption-content-in-flight-rest)

Internet Security Research Group. (2025). *What Is Memory Safety and Why*

*Does It Matter?* <https://www.memorysafety.org/docs/memory-safety>

Long, F., & Rosenau, M. (2023, January 10). *MSC60-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487376>

Lutkevich, B. (2021). Security Policy. TechTarget.   
 <https://www.techtarget.com/searchsecurity/definition/security-policy>

Milne, C. (2024, November 7). *STR31-C.* Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152048>

Mohindra, D., & O’Donnell, J. (2021, August 6). *ERR00-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487876>

Mohindra, D., & Rosenau, M. (2021, May 18). *ERR01-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487679>

Mohindra, D., & Rosenau, M. (2021, May 18). *ERR05-J*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487445>

National Institute of Standards and Technology. (n. d.). *Defense-in-Depth.*

<https://csrc.nist.gov/glossary/term/defense_in_depth>

OWASP. (2025). *SQL Injection Prevention Cheat Sheet.*   
 [https://cheatsheetseries.owasp.org/cheatsheets/SQL\_Injection\_Prevention\_  
 Cheat\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)

OWASP. (2025). *Validate All inputs Checklist.*   
 [https://owasp.org/www-project-developer-guide/draft/design/web\_app\_  
 checklist/validate\_inputs/](https://owasp.org/www-project-developer-guide/draft/design/web_app_checklist/validate_inputs/)

Parasoft. (2025). *Comprehensive Test Automation Solutions for C/C++*

*Software Development.*

<https://www.parasoft.com/products/parasoft-c-ctest/>

QA Systems. (2025). *Proven Absence of Runtime Errors with Sound Static Analysis.*

<https://www.qa-systems.com/tools/astree/>

Raza, M. (2023, January 31). *Encryption Explained: At Rest, In Transit &*

*End-To-End Encryption.* Splunk.

<https://www.splunk.com/en_us/blog/learn/end-to-end-encryption.html>

Reason Labs. (2023). *What is a Default-Deny Approach?* <https://cyberpedia.reasonlabs.com/EN/default-deny%20approach.html>

Reddy. (2019, July 30). *What do you mean by a dynamic initialization of variables?*

TutorialsPoint. [https://www.tutorialspoint.com/what-do-you-mean-by-a-](https://www.tutorialspoint.com/what-do-you-mean-by-a-dynamic-initialization-of-variables)

[dynamic-initialization-of-variables](https://www.tutorialspoint.com/what-do-you-mean-by-a-dynamic-initialization-of-variables)

Rosenau, M., & Seacord, R. C. (2018, August 6). *MAC11-C*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152346>

Saltzer, J. H. (1974). Protection and the Control of Information Sharing in Multics.

*Communications of the ACM, 17*(7), 388-402. [https://doi.org/10.1145/](https://doi.org/10.1145/361011.361067)

[361011.361067](https://doi.org/10.1145/361011.361067)

Saltzer, J. H. (1975). The Protection of Information in Computer Systems. *Proceedings of*

*the IEEE, 63*(9), 1278-1308. <https://www.cs.virginia.edu/~evans/cs551/saltzer/>

Scheila, R. (2024, February 14). *SEI Cert C++ Coding Standard.* Carnegie Mellon University. <https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>

Schiela, R., & Seacord, R. C. (2018, May 2). *Top 10 Secure Coding Practices.* Carnegie

Mellon University.  
 [https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+](https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices)

[Practices](https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices)

Seacord, R. C. (2013). *Secure Coding in C and C++.* Pearson Education.

Seacord, R. C. (2021, May 18). [*STR00-J*](https://wiki.sei.cmu.edu/confluence/display/java/STR00-J.+Don%27t+form+strings+containing+partial+characters+from+variable-width+encodings). Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487614>

Seacord, R. C. (2023, June 12). *Verify that all Integer Values Are in Range.*

Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152450>

Seacord, R. C. (2024, December 6). *Coverity.*

<https://wiki.sei.cmu.edu/confluence/display/c/Coverity>

Seacord, R. C., & Shrivastava, S. (2024, December 11)*. DCL30-C.* Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)

[?pageId=87152466](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152466)

Seacord, R. C., & Shrivastava, S. (2024, December 11)*. EXP35-C.* Carnegie Mellon

University. [https://wiki.sei.cmu.edu/confluence/pages/viewpage.action](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152058)

[?pageId=87152058](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152058)

Sentinel One. (2024, August 27). *What is Encryption? Types, Use Cases, and Benefits*.

Shrivastava, S. (2024, December 11). *INT33-C.* Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152254>

Shrivastava, S. (2024, December 11). *SIG35-C*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152239>

Shrivastava, S. (2024, December 12). *SIG35-C*. Carnegie Mellon University.

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152239>

Svoboda, D. (2023, April 20). [*DCL52-CPP. Never qualify a reference type*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile)

[*with const or volatile*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile)*.* Carnegie Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile)

[qualify+a+reference+type+with+const+or+volatile](https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL52-CPP.+Never+qualify+a+reference+type+with+const+or+volatile)

The QT Group. (2025). *Avixion Suite: Static Code Analysis & Architecture Verification.*

<https://www.qt.io/product/quality-assurance/axivion-suite>

TutorialsPoint. (2025). *Difference Between LPAD and Active Directory.*

<https://www.tutorialspoint.com/difference-between-ldap-and-active-directory>

Unstop. (2025). *Advantages of Inheritance in C++.*

<https://unstop.com/blog/inheritance-in-cpp>

Velimirovic, A. (2023 November 16). *Data Encryption In Use Explained.* PhoenixNap.

<https://phoenixnap.com/blog/encryption-in-use>

Weise, A. (2023, December 1).[*INT50-CPP. Do not cast to an out-of-range*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)

[*enumeration value*](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)*.* Carnegie Mellon University.

[https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)

[Do+not+cast+to+an+out-of-range+enumeration+value](https://wiki.sei.cmu.edu/confluence/display/cplusplus/INT50-CPP.+Do+not+cast+to+an+out-of-range+enumeration+value)