**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation is one of the easiest to secure an application but is easily overlooked or not fully encompassed. Validating input isn’t a singular solution but instead a methodology for every variable type. Validating data for strings might include verifying the length of the variable is within size bounds to prevent buffer overflows. This validation is not concerned with the meaning of the string but the data itself. Validating a string for its intent can be useful for preventing SQL injections by preventing keywords or syntax within the string. This might include entire refusal of the data or sanitizing the data by removing aspects to alter its intentions. (Seacord, 2013, p. 44) |
| 1. Heed Compiler Warnings | Modern compilers are becoming a layer of security. In the GNU C (GCC) compiler flags can be passed to the compiler to check for various security mechanisms. These mechanisms include buffer overflow prevention by not overwriting surrounding variables but don’t throw any visible errors when a buffer overflow is detected unless specified with a different flag. Additionally, compilers are now tracking variable sources by called them “tainted” and propagating that flag with any variable that might utilize its data. This allows developers to protect vulnerable internal functions from being used with tainted variable types. (Seacord, 2013, p. 125) |
| 1. Architect and Design for Security Policies | “The architecture and design of a system significantly influences the security of the final system” says security author Robert C Seacord. The author continues to say that if architecture or design is flawed, no standard can make your system secure (Seacord, 2013, p. 172). For instance, if a system can be divided so that minimal portions of code have access to privileged security levels we can mitigate our attack surface to a singular section of code. |
| 1. Keep It Simple | When projects begin to grow people tend to complicate them. Programs should be small enough to understand without compromising the original designs intention. When a mechanism is easily implemented and verified everyone can feel confident in the project’s integrity (Seacord, 2013, p. 173). When a projects team is able to maneuver its design easily it will not only hasten the development efforts but will allow haste in understanding the program for maintainers when vulnerabilities are found. Assuming your program will not encounter a vulnerability because ‘we built it right the first time’ is no more than hope as a security tactic. |
| 1. Default Deny | In a program or in terms of a user we should by default deny all privileges. We should explicitly allow a user to have any form of privilege whenever reasonably possible. This plays on the principle of zero trust where we always verify and never assume. This could apply to continuously authenticating a user’s bearer token or requiring user privileges to be regularly renewed (Lindemulder & Kosinski, 2024). |
| 1. Adhere to the Principle of Least Privilege | Like default denying, in the principle of the least privilege we won’t always have the benefit of denying all access potentially because of business rules dictating an acceptable level of risk to cost benefit. In this event we should incorporate the least privilege to a user, program, database, or application whenever possible. Take for example a program responsible for changing a password. Passwords are naturally maintained by higher privileges but if a program were to have an exploit that allowed unplanned queries or events to occur, we could affect all components on an application or database by allowing a blanket privileged policy. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data is an important task for receiving input to ensure that the input isn’t malicious, but project teams commonly forget to sanitize data sent. Trimming the data sent won’t just improve performance between systems but enhance your operational security as well. When every system does its part not just to protect itself but all operational components of an application you start to build a robust security atmosphere that embodies defense in depth ideals. Additionally, data sent to another system may contain privileged information that you’re attempting to protect. In a zero-trust application you must assume that the system you’re interacting will is compromised and should only receive the minimal data possible to perform its task. |
| 1. Practice Defense in Depth | Defense in depth is a practice of layering multiple security mechanisms at various levels. This may extend development time and increase complexity but the trade off is an application with layers of redundancy. Betting on one security mechanism to work is poor practice because hackers are constantly evolving and by layering multiple security types together, we ensure that no singular development in exploitation can jeopardize our systems (Seacord, 2013, p. 182). |
| 1. Use Effective Quality Assurance Techniques | Just as exploitation is consistently evolving as is security specialists. Security research allows us to bolster security mechanisms, but it isn’t always needed to recreate the wheel. Educating security experts on the latest trends is a singular aspect to a robust project team. When we contribute to industry standard security policies, we can retain our development and research time and devote those precise resources to unknown issues your specific application may have. Common policies have been tested from various angles by security research organizations that may have larger resource pools. |
| 1. Adopt a Secure Coding Standard | When we adopt secure coding standards, we are allowing ourselves to take advantage of already known and harden defense mechanisms that haven’t failed other organizations. This allows us to keep implementations conformed across every application or component in your system. Instead of having four different query sanitizing approaches implemented for your application you’re faced with a singular implementation that can be easily verified. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and non-compliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Data types can play a large role in projects’ ability to stay secure. Using appropriate data types that are driven to the exact uses of your expected data not only makes a project enhanced in terms of efficiency but security as well. For example, if you’re expecting a field on an application to only be 10 characters long it may be more beneficial to use the ‘char’ data type over a string data type (NUM00-J. Detect or prevent integer overflow, 2024). |

| **Noncompliant Code** |
| --- |
| Using a char array of size ten for the expected input, but the expected input is larger than the array causing an overflow. |
| **OR** |

| **Compliant Code** |
| --- |
| This code still uses a character array to improve system performance but instead does bounds checking to ensure input data is within range to prevent an overflow. This is accomplished in the method below by using the ‘strcpy’ function found in the ‘<cstring>’ library. By default, the ‘strcpy’ function doesn’t use bounds checking but we can instead use ‘strncpy’ and pass a size parameter as the third parameter to perform bounds checking. |
|  |

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

| **Principles(s):**  1. Validate Input – Using the proper data type for a given task can be seen as a form of data validation. In the example given earlier to use a char array instead of a string type to prevent integer overflows is an example of proactively managing your environment’s security through good design decisions.  4. Keep it Simple – Simple data types help promote readability and complexity. 6. Adhere to the principle of least privilege – By applying the most restrictive data type we’re able to control the level of functionality at time times which can help reduce our attack surface. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://582328.fs1.hubspotusercontent-na1.net/hubfs/582328/GrammaTech-Files/CWE_CPP_7.3.pdf) | 7.3 | CWE:89 | Improper Neutralization of Special Elements used in an SQL  Command (‘SQLInjection’) |
| [SonarQube](https://docs.sonarsource.com/sonarqube-server/9.8/user-guide/rules/security-related-rules/) | 9.8 | CWE-89 | Security-injection rules: there is a vulnerability here when the inputs handled by your application are controlled by a user (potentially an attacker) and not validated or sanitized. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Data values are the life blood of any application and should be carefully handled whenever possible. In STD-001-CPP we discussed the importance of selecting the appropriate data type for your variables, but for this standard we’re discussing the data values. This standard focusses on data initialization. Your variables should always be initialized to a starting value to avoid unintended results. This might include initiating a string to a blank string value, so it doesn’t assume the value of NULL initially or initializing an integer to a value (EXP53-CPP. Do not read uninitialized memory, 2023) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code we can see that the ‘show\_balance’ function only gives the balance variable an assignment if it were to have an email passed to it. For most cases this would work fine, but if the user submitted the field without an email the balance variable wouldn’t get an assignment creating unknowns in our program. Without an assignment the memory space that is balanced still contains residual data from previous uses and will reference this data instead. This could be detrimental depending on what data was stored in memory prior at worst. At best the user would have poor user experience diminishing the service’s reputation. |
|  |

| **Compliant Code** |
| --- |
| By simply initializing the balance variable within the ‘show\_balance’ function we can avoid unknowns in our program. Now that balance is initialized to zero, even if the user submits the email field with nothing it will always return $0.00. While it may be better to stop this issue from happening at the email assignment level, we might not be the developer assigned to that portion of the project, or a vulnerability could be later found that allows no email to be passed successfully despite data validation efforts. Despite the reasoning we should build these considerations redundantly throughout the system to create a defense in-depth atmosphere in our project. Be sure to well document your security considerations as they may need to be adjusted later as new information, regulations, or best practices appear. |
|  |

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

| **Principles(s):**  1. Validate Input – Some data may be provided to you from other sources, and you cannot always guarantee that the data will be initialized. For this reason, validating your input is a high severity issue as uninitialized values will cause unknown effects in your program. 10. Adopt a secure coding standard – Make it standard with your development group that all variables should be initialized. While this isn’t a guarantee that it will work it does give a level of creditability to enforce in your group as far as best practices. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | uninitialized-read | Partially checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | LANG.STRUCT.RPL  LANG.MEM.UVAR | Return pointer to local  Uninitialized variable |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | uninitialized-read | Partially checked |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | String operations can pose interesting effects if not handled correctly. This standard focuses on using correct replacement techniques. (Use valid references, pointers, and iterators to a basic\_string, 2024). |

| **Noncompliant Code** |
| --- |
| This code is noncompliant as it causes a segmentation when referencing inaccessible memory. |
|  |

| **Compliant Code** |
| --- |
| Whenever possible using generic algorithms such as replace we’re able to enact industry standard algorithms that have been proven to resist a hacker’s grasp, is well documented of any vulnerabilities, or will be quickly updated as a collective effort to resist vulnerable code. |
|  |

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

| **Principles(s):** 2. Heed Compiler Warnings – Compilers are great tools themselves and can detect many unsafe operations. Fixing these issues as the compiler warns about them is important as you never known when the scenario in which the warning is built around becomes an issue for your environment. 4. Keep it simple – when working with strings use any built-in function such as the standard string library replace function to handle replacements over iteration and manual manipulation. Using standard libraries can help mitigate known or overlooked logical errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | ALLOC.UAF | Use After Free |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2025.1 | DF4746, DF4747, DF4748, DF4749 |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C++: STR52-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr52cpp.html) | Checks for use of invalid string iterator (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | SQL injections can be at the top of the list for most detrimental attacks because they target the most valuable asset a program has, its data. An application should go through the effort to sanitize queries and remove potentially harmful keywords, key characters, or should use prepared queries (IDS00-J. Prevent SQL injection, 2025). |

| **Noncompliant Code** |
| --- |
| This code is vulnerable to SQL injection attacks because it concatenates data into a query string. The entire query is passed to the sqlite execution function. This is accomplished by passing the username “admin’--” which fundamentally alters the queries intentions. The ‘--’ starts sql comments which means the rest of the query that requires a password to authenticate will effectively be null and no longer enforced. |
|  |

| **Compliant Code** |
| --- |
| This code is compliant because it used prepared statements. Prepared statements take the data and treat it separately from the query doing an effective sanitization of the query. This means that our commenting trick won’t work here because it would search literally for “admin’--". |
|  |

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

| **Principles(s):** 1. Validate Input – When working with databases it’s critical to validate the input. Prepared queries can help resolve this issue. 6. Adhere to the principle of least privilege – Databases are structured in such a way that users are configured and given permission on how they can interact with the database. Setting up a user for your application that covers the minimum requirements such as reading only if no writing is needed or not being able to access other tables that the application doesn’t need. 7. Sanitize data sent to other systems – Since your application is working with another system, a database, it’s advised that you attempt to limit the damage your data could do when passing it to this system. This might include denying queries that contain keywords that could be used for malicious purposes. 8. Practice Defense in Depth – SQL injections are difficult to combat because they can be achieved through various means which implies the solution is through multiple modes of coverage. Using the combination of the above principles together can help mitigate issues over just implementing one of them. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Fortify](https://wiki.sei.cmu.edu/confluence/display/java/Fortify) | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_\_Persistence  SQL\_Injection | Implemented |
| [Klocwork](https://wiki.sei.cmu.edu/confluence/display/java/Klocwork) | 2024.4 | SV.DATA.DB  SV.SQL  SV.SQL.DBSOURCE | Implemented |
| [SpotBugs](https://wiki.sei.cmu.edu/confluence/display/java/SpotBugs) | 4.6.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE  SQL\_PREPARED\_STATEMENT\_GENERATED\_FROM\_NONCONSTANT\_STRING | Implemented |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/java/Coverity) | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | When an error occurs, we can normally gracefully exit the code but when we access memory that has been deleted, we get what’s called a segmentation error. These errors are difficult to troubleshoot and can cause unexpected results in your application. For these reasons and many more we should be protecting our memory from these types of calls (MEM50-CPP. Do not access freed memory, 2023). |

| **Noncompliant Code** |
| --- |
| This code is noncompliant because it attempts to access the myStruct structure after it’s deleted. This means that we’re referencing memory that no longer contains our desired structure causing a segmentation error. |
|  |

| **Compliant Code** |
| --- |
| The solution to protect your code from accessing deleted memory is to simply not call delete until after all accessible calls have been made. Additionally, not calling delete all together might be the optimal choice as modern versions of C++ will automatically collect the myStruct definition once leaving the main() scope as it isn’t referenced anywhere else. |
|  |

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

| **Principles(s):**  2. Heed Compiler Warnings – Most compilers can detect a high likelihood of a variable being accessed after a potential release of memory has occurred potentially due to logic implemented. Listening to these warnings and potentially following the advised course of action is always suggested. 10. Adopt a secure coding standard – Implementing a coding standard for your development group can help mitigate these issue by defining when and where to release memory or checking to ensure memory is valid before using functions or computations. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | dangling\_pointer\_use |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | ALLOC.UAF | Use after free |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a powerful tool found in most object-oriented programming languages. These tools allow you to make ‘rules’ that must be followed in your code such as ensuring a variable size stays below a specific size. This can be useful in many scenarios but as an example we can imagine an environment that is heavily controlled by its resources such as an embedded system which may be operating with a lightweight operating system such as a 16-bit OS. This would become an issue if we planned to write four bytes of data to a 2 byte integer as defined by a 16-bit OS (DCL03-C. Use a static assertion to test the value of a constant expression, 2025). |

| **Noncompliant Code** |
| --- |
| In this example a variable is larger than the allowed ‘business rule’. Our hypothetical goal is to ensure that the int is four bytes because we plan to assign four bytes worth of data with it. |
|  |

| **Compliant Code** |
| --- |
| This code is compliant because we’re enforcing our ‘business rule’ using static assertion. Our goal is to ensure the int is 4 bytes which is successful. |
|  |

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

| **Principles(s):** 2. Heed Compiler Warning – When assertions fail many compilers will provide useful information into why the assertion was not met. Using this information can make your program a more robust environment. 3. Architect and Design for security polies – Consider the project as a whole and what system they’ll be ran on. Even being unsure of what system it operates on would be a design consideration as the first logical step is creating instructions for various versions or ample error documentation to be triggered when an unaccepted environment is detected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 9.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | CC2.DCL03 | Fully implemented |
| [Compass/ROSE](https://wiki.sei.cmu.edu/confluence/display/c/Rose) |  |  | Could detect violations of this rule merely by looking for calls to assert(), and if it can evaluate the assertion (due to all values being known at compile time), then the code should use static-assert instead; this assumes ROSE can recognize macro invocation |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Some objects or functions do not utilize automatic destructors for various reasons. The most common way these are handled is with an exit handler. Exit handlers are called with the atexit() function call. When the program is terminated it will execute code within all exit handlers but only if the program doesn’t come to an abrupt stop such as when exit() is called. If an exit handler itself errors out it will prevent further exit handlers from executing. It’s important that all exit handlers have robust error handling to ensure other exit handlers may execute. |

| **Noncompliant Code** |
| --- |
| This exit handler, b(), is registered in the bad() function call. When the program terminates it will call the b() exit handler registered. This program will have the exit handler throw an error. If it’s not caught all other exit handlers will not run. |
|  |

| **Compliant Code** |
| --- |
| The exit handler g() will be registered in good(). When the program terminates it will call the g() exit handler which will throw an error like the b() exit handler. The difference is that the error is wrapped in a try catch allowing further exit handlers to execute. |
|  |

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

| **Principles(s):**  3. Architecture and Design for security policies – Reliable program termination is key for graceful shutdown of systems. In embedded systems that could be providing a vital service and needs a specific shutdown sequence it’s important that those options are met with built in functions. 4. Keep it Simple – There is no need to complicate the issues as hand. Most modern languages have built in functions to control the termination of a program. Rather than implementing a solution yourself it’s important to consider standard library approaches that have been well tested and documented. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | stdlib-use | Partially checked |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 122 S | Enhanced Enforcement |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: ERR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr50cpp.html) | Checks for implicit call to terminate() function (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | stdlib-use | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output Validation | [STD-008-CPP] | Data is the ‘life blood’ of any application and reading and writing from files are frequent operations in an application. There comes a time when you need to read a file, interpret, and rewrite the file. In the event that you don’t close the file and reopen it under a new fstream() object we can have undefined results when the accessing it the second time without resetting the file position (FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call, 2023). |

| **Noncompliant Code** |
| --- |
| This code will have an undefined effect as it reaches the end of the file but continues to write and to use the fstream object after the data is read. This can create unintended results. |
|  |

| **Compliant Code** |
| --- |
| In this compliant code example, we can observe a seekg() function call. This function call allows us to call our fstream object position back to the beginning of the file. This removes the chances that you’ll have an undefined result as long as its called between every read and write operation. |
|  |

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

| **Principles(s):** 1. Validate Input – Improperly handled file streams can lead to data in a corrupt manner. Before reading or writing verify the state of your input/output stream. 2. Heed Compiler Warning – Again, compilers are great tools that can provide valuable information on the state of your project. Compilers can often detect the state of an input or output stream and provide a warning for you to adjust it. 3. Keep it simple – Many file streams have built in functions such as various forms of seek or clearing a stream. Use these functions over ‘house’ built functions to ensure that there is robust error handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | CertC++-FIO50 |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | IO.IOWOP  IO.OIWOP | Input After Output Without Positioning  Output After Input Without Positioning |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP-FIO50-a | Do not alternately input and output from a stream without an intervening flush or positioning call. |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2025.1 | DF4711, DF4712, DF4713 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Deleting an array | [STD-009-CPP] | Many developers struggle to grasp lower-level languages as you are needed to handle memory carefully. In C++ we’re required to use pointers for many operations. It’s imperative that we do not delete an array through a pointer of the incorrect type (EXP51-CPP. Do not delete an array through a pointer of the incorrect type, 2025). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, we can see that the array is stored in a Base \*. This definition will result in undefined behavior that could be troublesome to resolve. |
|  |

| **Compliant Code** |
| --- |
| In the compliant code example below, we can see that the array is instead stored in a Derived \*. This takes away to possibility of the undefined behavior when indexing or deleting the pointer. |
|  |

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

| **Principles(s):**  4. Keep it simple – Simplify your memory management using consistent types and use smart pointers whenever possible. 8. Practice Defense in Depth – Memory safety can quickly become a security issue. Using multiple types of checks to ensure memory management is adhered to can be time consuming but worth it if implemented properly. Static code analysis tools and manual code reviews are some forms of these checks beyond the automation below. 10. Adopt a secure coding standard – There are a wide variety of coding standards to choose from that are available and tested by larger organizations. Implementing these policies into your development group can ensure that data is treated with care and formal rules such as the types of pointers to use are implemented thoroughly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | -analyzer-checker=cplusplus | Checked with clang -cc1 or (preferably) scan-build |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | ALLOC.TM | Type Mismatch |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2024.2 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024b | [CERT C++: EXP51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp51cpp.html) | Checks for delete operator used to destroy downcast object of different type. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Strings have room for null terminators and characters | [STD-010-CPP] | When working with strings it’s important to work with them carefully as creating a buffer overflow is extremely likely. We need to check that the string has room for all the data we’re assigning it and a final null terminator character (STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator, 2025). |

| **Noncompliant Code** |
| --- |
| In this noncompliant example we can see that our input variable is set to 12 characters, but the input stream doesn’t restrict it to such. This could lead to larger input sizes being assigned to the variable and overflowing into nearby memory causing undefined results. |
|  |

| **Compliant Code** |
| --- |
| The best solution is to use a bounded array such as std::string. This will prevent the input stream overflowing when data reaches the bound of the variable. This could affect performance on resource-controlled environments such as embedded systems, but the recommendation is still the same but with additional checks to append the data into a smaller variable size. |
|  |

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

| **Principles(s):**  1. Validate code input – String manipulation can be tricky and is a common attack surface and thus you need to ensure that your inputs are checked before appending, editing, or copying. This prevents buffer overflows and truncation issues. 4. Keep it simple – Many standard library variables such as std::string have a method for bounds checking. Using these bounds checks can ensure that data is conformed. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | stream-input-char-array | Partially checked + soundly supported |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 9.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024b | [CERT C++: STR50-CPP](https://www.mathworks.com/help/bugfinder/ref/certcstr50cpp.html) | Checks for:  Use of dangerous standard function  Missing null in string array  Buffer overflow from incorrect string format specifier  Destination buffer overflow in string manipulation  Insufficient destination buffer size  Rule partially covered. |

### Defense-in-Depth Illustration

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



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Starting with the Assess and plan section, we can start by addressing technical debt from previous iterations or from the start of the current state of the program. During the planning and Assessing phase we aren’t just limited to technical debt but all forms of debt such as security debt as well. Threat modeling and DevSec metrics are gathered and detailed during this phase. Accomplishing this with automation can be done through tools such as dependency scanning with OWASP. These scanners give early indications of what your threat surface may include.  
 Moving next to the Design phase of the DevOps process we can start with our test-driven design. This type of development includes unit tests being built prior to the development even starting. These unit tests can be implemented in a pipeline with tools such as Jenkins to ensure that Developers are getting real time feedback with tests during the later Build phase. These unit tests aren’t just restricted to security improvements but general best practices as well. Using unit tests to ensure that style guides and company wide development standards are being upheld is a great resource. During the verify and test section of the DevOps process we can start to perform vulnerability scanning on not just our dependencies but our own code base. Tools such as CPPcheck and other static code analysis tools can ensure that errors are caught early and often.

Moving to the production side of the DevOps process we can see Transition and health checks are implemented. During this time Infrastructure as Code security audits can be implemented to and automated with platforms such as Terraform and AWS CloudFormation. Monitoring and Detection is a continuous endeavor that takes hyper vigilance and since it’s usually done through structured logs and tools it makes a great area for automation. This might include security information and event management (SIEM) tools to be implemented which act as an aggregate for the various locations of data and logs. SIEM gives operations staff a ‘pulse’ on the project to ensure it is doing fine.   
 During the response phase it’s easy for companies to assume they won’t need to handle an emergency. Having a game plan of time can get your project back to production with good health nearly instantly. Tools such as auto rollback all enable operations teams to remove faulty code and restore them to the most previously known working version. Isolation is another endeavor commonly performed during the response phase. This might mean isolating an infected project instantly from services like its database when malicious intent is detected or isolating the entire project from the outside network to retain its healthy state. Finally, during the maintain and stabilize section of the DevOps chart we can automate our health checks to compare themselves against previous checks. If the health is declining but isn’t flagged by any previous section of DevOps, we can compensate by adding additional resources by spinning up more virtualization. Common platforms for this include AWS and Google Cloud. This can be very important if your service has been down after dealing with a response and a wave of users are expected to return after not being able to utilize your service.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Likely | Medium | P18 | L1 |
| STD-002-CPP | High | Probable | Medium | P12 | L1 |
| STD-003-CPP | High | Probable | High | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Low | Likely | Medium | P6 | L2 |
| STD-009-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-010-CPP | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | At this time the data is stored in hard drives typically and can be vulnerable to physical attacks such as theft of the hard drive. To mitigate this, it’s recommended to encrypt the drive with services such as BitLocker which use AES 256 encryption. |
| Encryption in flight | During this time the data is being transferred over potentially unsecure methods and needs to be encrypted with TLS 1.2 or higher. This might include implementing HTTPS to sites to ensure that data is transferred and interpreted by only the two parties. This prevents man in the middle attacks and many others. |
| Encryption in use | During this time the data is being worked upon by an application or user. This means that the data is loaded in memory and could be at risk of being stolen from malicious programs with access to that memory. This is especially important when it comes to SPII or other highly sensitive data sources. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is an important step to the Triple – A Framework. During this step we ensure that all users undergo checks to ensure their identity. Adding as many noninvasive checks as possible is important. This could include multi-factor authentication (MFA) and enforcing strong passwords. |
| Authorization | Authorization can be summarized as giving access to users that are minimal and exactly what is needed. Ensuring a user cannot utilize sections of a system restricted to them is a key portion of the Triple A framework. This might mean that users cannot access data from other users for example. This ensures that databases are managed with the least privilege necessary. Creating permission levels such as staff and average user is important distinctions at this stage. |
| Accounting | All actions that a user makes need to be logged in detail to ensure that a recreation of events could occur. This makes the job of a rollback that much easier if you understand the actions that were taken. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 06/22/2025 | Filled in policies | Cade Bray |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

# References

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

Carnegie Mellon University. (2023, January 19). *FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO50-CPP.+Do+not+alternately+input+and+output+from+a+file+stream+without+an+intervening+positioning+call

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

Carnegie Mellon University. (2024, February 14). *NUM00-J. Detect or prevent integer overflow*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/java/NUM00-J.%2BDetect%2Bor%2Bprevent%2Binteger%2Boverflow

Carnegie Mellon University. (2024, January 19). *STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic\_string*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Carnegie Mellon University. (2025, May 19). *DCL03-C. Use a static assertion to test the value of a constant expression*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression

Carnegie Mellon University. (2025, January 30). *EXP51-CPP. Do not delete an array through a pointer of the incorrect type*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP51-CPP.+Do+not+delete+an+array+through+a+pointer+of+the+incorrect+type

Carnegie Mellon University. (2025, January 31). *IDS00-J. Prevent SQL injection*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+Injection

Carnegie Mellon University. (2025, January 30). *STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator*. Retrieved from Confluence: https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator

Lindemulder, G., & Kosinski, M. (2024, June 20). *What is Zero Trust?* Retrieved from IBM: https://www.ibm.com/think/topics/zero-trust

Seacord, R. C. (2013). *Secure Coding in C and C++ (2nd ed.).* Pearson Technology Group.