**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 external (untrusted) data sources should be validated to be non-malicious before being used. Input, in this context, meaning any data originating from outside the system or data derived from variables outside of the system. |
| 1. Heed Compiler Warnings | Compile code using the highest available warning level and modify existing code to eliminate compiler warnings. Additionally, code analysis tools like code linters aid in additional code security. Compiler warnings are created for a reason and ignoring them can lead to compounding disastrous results. |
| 1. Architect and Design for Security Policies | Software should be architected and designed in a manner that eases the implementation and enforcement of security policies. If parts of a system have varying permissions required for certain functionality, consider fragmenting the system into several subsystems, each with its own privilege structure. |
| 1. Keep It Simple | Keep software design as simple as possible to avoid the confusion that can accompany complex or very large designs. In addition to causing confusion amongst developers in a large or complex system, such systems require much more effort to effectively secure. |
| 1. Default Deny | Outside variables or outside actors trying to access a system should be denied access by default. Data or actors from outside the system must be given explicit permission to access the system. |
| 1. Adhere to the Principle of Least Privilege | Only the lowest level of permission needed to complete a task should be given to any process. Any process that requires elevated (admin) privileges should be given such privileges *only* for as long as the elevated permissions are needed. This helps mitigate damage if an attacker can gain arbitrary code execution abilities. |
| 1. Sanitize Data Sent to Other Systems | Data intended to be used as specialized input (SQL query fields, CLI commands, etc.) should be sanitized/preprocessed before being passed to the external systems in which they are used as input. In C++, prepared SQL statements are an example of this security principle in action as query fields are type validated and escape characters ignored before being passed in an actual SQL query. This helps prevent a multitude of injection style attacks. |
| 1. Practice Defense in Depth | Use multiple layers of defense to reduce the number of system wide vulnerabilities. Each layer of security will have its own weaknesses or unique coverage cases. Stacking layers that have differing vulnerabilities and coverage areas can ensure that the system is as close to fully protected as possible. |
| 1. Use Effective Quality Assurance Techniques | In the modern era, there exist a multitude of proven quality assurance techniques to assist in measuring security and functionality coverage. Such techniques include code reviews, automated testing, and white hat testing methods. |
| 1. Adopt a Secure Coding Standard | Especially so for formal organizations, a secure coding standard should be developed or adopted to allows for greater levels of standardization and security across all products produced by the organization/individual. |

### C/C++ Ten Coding Standards

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

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Do not create incompatible declarations of the same function or object  <https://wiki.sei.cmu.edu/confluence/display/c/DCL40-C.+Do+not+create+incompatible+declarations+of+the+same+function+or+object> |

| **Noncompliant Code** |
| --- |
| A variable *i* is declared in the file b.cpp as a *short* type. Using the *extern* keyword, *i* is imported from b.cpp to the file a.cpp where it is declared as an *int* type. This causes undefined behavior. |
| /\* Code in a.cpp \*/  extern int i; //i imported from b.cpp here as int type  int increment\_i() {  return ++i;  }  /\* Code in b.cpp \*/  short i; //i declared here as short type |

| **Compliant Code** |
| --- |
| The variable *i* maintains the same type everywhere it is used across all files. |
| /\* Code in a.cpp \*/  extern int i; //i imported from b.cpp here as int type  int increment\_i() {  return ++i;  }  /\* Code in b.cpp \*/  int i; //i declared here as int type |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – Mismatched typing is an easy mistake to make, especially in a multifile projects, and effective quality assurance techniques like automated static code analysis and senior code reviews can mitigate such errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | type-compatibility | Fully checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.IF | Inconsistent function declarations |
| Klocwork | 2024.2 | MISRA.FUNC.NOPROT.DEF.2012 | Fully implemented |
| Polyspace Bug Finder | R2024a | CERT C: Rule DCL40-C | Checks for declaration mistmatch (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not dereference null pointers  <https://wiki.sei.cmu.edu/confluence/display/c/EXP34-C.+Do+not+dereference+null+pointers> |

| **Noncompliant Code** |
| --- |
| An int pointer, *ptr* is initialized as *nullptr*, passed to a function, and dereferenced to print the *ptr* data to console. Dereferencing the *nullptr* results in undefined behavior as *nullptr* does not represent a valid memory address. |
| #include <iostream>  void print\_ptr\_data(int\* ptr) {  // No checks if ptr is null or not  std::cout << \*ptr << std::endl;  }  int main() {  int\* ptr = nullptr;  print\_ptr\_data(ptr);  return 0;  } |

| **Compliant Code** |
| --- |
| Checks are implemented to only allow the dereferencing of a non-null pointer. |
| #include <iostream>  void print\_ptr\_data(int\* ptr) {  if (ptr != nullptr) {  std::cout << \*ptr << std::endl;  } else {  std::cout << “Cannot dereference null pointer.” << std::endl;  }  }  int main() {  int\* ptr = nullptr;  print\_ptr\_data(ptr);  return 0;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Adoption of a secure coding standard carries the implication of enforcement of the standards. In C++, any secure coding standard worth adopting will have rules for avoiding null pointer dereferencing. Enforcing this standard either through code reviews or automatic linting tools will help to prevent this error. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | null-dereferencing | Fully checked |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null pointer dereference |
| Cppcheck | 1.66 | nullPointer | CWE: 476  Possible null pointer dereference: ptr |
| Parasoft C/C++ test | 2023.1 | CERT\_C-EXP34-a | Avoid null pointer dereferencing |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer.  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR51-CPP.+Do+not+attempt+to+create+a+std%3A%3Astring+from+a+null+pointer> |

| **Noncompliant Code** |
| --- |
| A string type variable *my\_string* is created from the results of a call to *std::getenv(ENV\_VAR)*. If *ENV\_VAR* is not a valid environment variable, *std::getenv()* returns a null pointer. Attempting to use method calls on a null pointer will result in an error or undefined behavior. |
| #include <cstdlib>  #include <string>  void make\_str() {  std::string my\_string(std::getenv(“TMP”)); //Might be nullptr  if (!my\_string.empty()) {  // ...  }  } |

| **Compliant Code** |
| --- |
| A ternary check is implemented to assign *my\_string* an empty string if *std::getenv()* returns a null pointer. This allows *my\_string* to **always** have the *std::string* type that allows the use of method calls. |
| #include <cstdlib>  #include <string>    void make\_str() {  const char \*tmpPtrVal = std::getenv("TMP");     std::string my\_string(tmpPtrVal ? tmpPtrVal : "");     if (!my\_string.empty()) {       // ...     }  } |

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

| **Principles(s):** Heed Compiler Warnings – Building this code in Visual Studio gives a compiler warning that *std::getenv* might be unsafe if the named environment variable is not valid/available as it will result in a null pointer being returned instead of the expected value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Astree | 22.10 | assert\_failure | None |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024a | CERT C++: STR51-CPP | Checks for string operations on null pointer (rule partially covered). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL injection  <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection> |

| **Noncompliant Code** |
| --- |
| An SQL query is executed directly with user provided data with no additional checks. |
| #include <iostream>  sql::Connection \*con;  sql::Statement \*stmt;  driver = sql::mysql::get\_mysql\_driver\_instance();  con = driver->connect(localhost, user, passw);  std::string username;  cout << “Enter username” << endl;  cin >> usern;  stmt = con->createStatement();  stmt->execute(“SELECT \* FROM accounts WHERE username = ‘” + usern + “’”); |

| **Compliant Code** |
| --- |
| A prepared statement is used which sanitizes the user input before using the data in the SQL query. |
| #include <iostream>  sql::Connection \*con;  sql::PreparedStatement \*prep\_stmt;  driver = sql::mysql::get\_mysql\_driver\_instance();  con = driver->connect(localhost, user, passw);  std::string username;  cout << “Enter username” << endl;  cin >> usern;  prep\_stmt = con->prepareStatement(“SELECT \* FROM accounts WHERE username = ?”);  prep\_stmt->setString(1, usern);  prep\_stmt->execute(); |

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

| **Principles(s):** Sanitize Data Sent to Other Systems – SQL queries often require forwarding user provided data to specialized libraries to interact with databases. If not properly sanitized before being forwarded to the specialized libraries, the system is left vulnerable to SQL injection attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Klocwork | 2024.2 | SV.SQL | Implemented |
| SonarQube | 9.9 | S3649 | SQL queries should not be vulnerable to injection attacks |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory.  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory> |

| **Noncompliant Code** |
| --- |
| A struct, *S* has a member function *f()*. A new instance of *S* is created and then deleted. After deletion, *f()* is called from the instance. |
| #include <new>  struct S {  void f();  }  int main() {  S \*s = new S;  delete s;  s->f(); //s is no longer valid by this point  } |

| **Compliant Code** |
| --- |
| The instance of struct *S* is not deleted until after the last reference to the instance is used. |
| #include <new>  struct S {  void f();  }  int main() {  S \*s = new S;  s->f();  delete s;  } |

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

| **Principles(s):** Adopt a secure coding standard – Use after free is common error that is often a matter of doing needed operations in the wrong order which results in memory being accessed *after* it has been explicitly freed. Adopting and enforcing a secure coding standard can help ensure that operations are completed in the order that they ought to be. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | dangling\_pointer\_use | None |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |
| 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] | Incorporate diagnostic tests using assertions  <https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions> |

| **Noncompliant Code** |
| --- |
| An *assert()* statement is used as an error handling technique rather than a simple verification of internal state for debugging purposes. |
| #include <iostream>  #include <cassert>  void divide(int numerator, int denominator) {  assert(denominator != 0);  // No error handling + assert disabled for release build = bad code  int result = numerator / denominator;  std::cout << result << std::endl;  }  int main() {  int num;  int denom;  std::cout << “Enter numerator and denominator” << std::endl;  std::cin >> num >> denom;  divide(num, denom);  return 0;  } |

| **Compliant Code** |
| --- |
| The assert statement will cause program termination during debugging if the denominator is 0 but in a release build where the assert statement will not be present, the error handling immediately after will handle the invalid denominator gracefully and without program termination. |
| #include <iostream>  #include <cassert>  void divide(int numerator, int denominator) {  assert(denominator != 0);  if (denominator == 0) {  std::cerr << “Cannot divide by 0” << std::endl;  return;  }  int result = numerator / denominator;  std::cout << result << std::endl;  }  int main() {  int num;  int denom;  std::cout << “Enter numerator and denominator” << std::endl;  std::cin >> num >> denom;  divide(num, denom);  return 0;  } |

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

| **Principles(s):** Use effective quality assurance techniques – assert statements can be an extremely effective quality assurance tool when used to validate internal states during development. By *knowing* what variables or state values will be at any given time, the developer can more confidently implement more complex features without the worry of getting bogged down by a myriad of fine details. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | P1 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have a side effect. |
| Parasoft C/C++ test | 2023.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions.  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions> |

| **Noncompliant Code** |
| --- |
| A function, *throwing\_func()* throws an exception that is handled neither by function *f()* nor *main()*. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| A *try-catch* block is implemented in *main()* to gracefully handle the exception thrown from *throwing\_func()*. |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  try {  f();  }  catch (Error e) {  // Handle the error  }  } |

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

| **Principles(s):** Architect and Design for Security Policies – Unhandled exceptions can cause undefined behavior or unexpected program crashes. Every exception that could be thrown during a process should be handled using a catch-block. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | main-function-catch-all | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Klocwork | 2024.2 | MISRA.CATCH.ALL | None |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-ERR51-a | Always catch exceptions |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Iterators | [STD-008-CPP] | Use valid iterator ranges.  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges> |

| **Noncompliant Code** |
| --- |
| A vector passed to a function is iterated over with the starting point being set to a location **after** the ending point. This leads to infinite iteration beyond the bounds of the vector and thus, undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| The starting location for iteration is set to a location **before** the set end location. |
| #include <algorithm>  #include <iostream>  #include <vector>  void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):** Use effective quality assurance techniques – Using an invalid iterator range is an easy mistake to make as it only requires the start point to be further than the end point (inverted). Effective quality assurance techniques like static code analysis and code reviews can help mitigate this issue. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | overflow\_upon\_dereference | None |
| CodeSonar | 8.1p0 | LANG.MEM.BO | Buffer Overrun |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-CTR53-a | Do not use an iterator range that isn’t really a range |
| Polyspace Bug Finder | R2024a | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-009-CPP] | Ensure that unsigned integer operations do not wrap.  <https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap> |

| **Noncompliant Code** |
| --- |
| Addition of integer types of a fixed size may overflow if not explicitly prevented. |
| void func(unsigned int a, unsigned int b) {  unsigned int sum = a + b; //Overflows if summ > 4294967295 (uint::MAX)  } |

| **Compliant Code** |
| --- |
| An *if-else* block is used to ensure no overflow will occur before actually summing the integers. If *b* is larger than UINT\_MAX minus *a*, this means an overflow will occur if the integers are summed. This works because UINT\_MAX – a gives the max value of an integer that could be then added before overflow occurs. |
| #include <limits.h>  void func(unsigned int a, unsigned int b) {  unsigned int summ;  if (UINT\_MAX – a < b) {  //Error handling  } else {  summ = a + b;  }  } |

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

| **Principles(s):** Architect and design for security policies – integer wrapping is an easy mistake to make and to protect against. Systems should be designed in a way that if an integer has even the slightest chance at approaching it upper limit during any process, a higher capacity int type should be used instead. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 24.04 | integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 8.1p0 | ALLOC.SIZE.ADDOFLOW | Addition overflow of allocation size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Function returns | [STD-010-CPP] | Value-returning functions must return a value from all exit paths.  <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MSC52-CPP.+Value-returning+functions+must+return+a+value+from+all+exit+paths> |

| **Noncompliant Code** |
| --- |
| A function returns the absolute value of an integer parameter but fails to return anything if the integer is >= 0 due to oversight. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  //Nothing returned when a >= 0  } |

| **Compliant Code** |
| --- |
| A value is returned from each possible exit point of the function. |
| 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):** Heed compiler warnings – When compiled, the non-compliant code example gives a compiler warning that not all paths return a value. Failure to return a value when one is expected can cause unexpected or undefined behavior. |
| --- |

**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 |
| CodeSonar | 8.1p0 | LANG.STRUCT.MRS | Missing return statement |
| Polyspace Bug Finder | R2024a | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |
| RuleChecker | 22.10 | return-implicit | Fully checked |

### Defense-in-Depth Illustration

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



### Automation



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.

The process of transitioning from DevOps to DevSecOps carries with it many additional steps in the CI/CD loop that enhance security and ease its implementation. In the “Assess and plan” phase, the environment in which a system is to be built is thoroughly analyzed to garner what threats to the system may arise, what regulations might affect the system design or functionality, and bleeding edge threats are researched. Implementation of security in every layer of the system follows in the next loop stages. While in production, DevSecOps dictates that a system should undergo regular health checks by way of security audits and penetration tests. The DevSecOps loop also shows that in addition to active threat auditing, active threat monitoring should be used to maintain a watchful eye for any real threats that attempt to compromise the system. In traditional security, these two stages may take the form of building a thick steel door but still placing a security camera above it. If monitoring detects a threat in the system, there should be well documented response procedures so that any damage can be prevented/mitigated if the threat manages to bypass other security layers. Finally, all security layers should be maintained by constantly updating the knowledge base of new threats and ensuring all security tools are updated to the latest available versions.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Unlikely | Medium | 2 | 3 |
| STD-002-CPP | High | Likely | Medium | 18 | 1 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-CPP | High | Probable | High | 6 | 2 |
| STD-009-CPP | High | Likely | High | 9 | 2 |
| STD-010-CPP | Medium | Probable | Medium | 8 | 2 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Data at rest indicates that the data is not currently being used by a program and is not currently being transferred between systems. Encrypting data at rest can be done using a variety of techniques and is done to protect data that is kept in one place for a period of time where it may be vulnerable to attacks from those who know where to find the data. |
| Encryption in flight | Encryption in flight concerns the encryption of data as it is being transferred from system to system or from device to device. This is done to protect against packet sniffing (data being read/manipulated as it transfers across readable bandwidths) and man-in-the-middle attacks (an attacker mimicking a trusted connection point to gather data). |
| Encryption in use | Encryption in use is the practice of encrypting data during use by a program. For instance, a program might ask for a sensitive data value, perform some operations on it, and then free the memory containing the value. The data, while in use, should remain encrypted all the way until the point where its unencrypted use is required by the program, after which, it should be immediately re-encrypted. This does not protect against all data leakage for in-use sensitive data but it does mean that exfiltrating this sensitive data becomes extraordinarily harder and nigh impossible. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the step in which a user verifies their identity. This can be done through a variety of techniques but one of the most common authentication methods is the use of a unique identifying username and a strong password. Authentication is used to validate that people are who they present themselves as. |
| Authorization | Authorization dictates which users have access to certain parts or certain functionalities of a system. This access control for a specific user is called the user’s privilege. A user’s privilege determines what level of access they have to system functionality, to system data, and to alter the system in any way. |
| Accounting | Accounting is similar to monitoring in a system but tends to be more focused on tracking users of a system rather than monitoring a system’s vulnerable points. During accounting, a user’s actions, navigation route, time spent on page, and other analytics may be gathered to form a profile for how that specific user is interacting with the system. This is useful for if a user should interact with the system in an unauthorized manner, accounting ensures that there is a paper trail detailing the actions the user took prior to the unauthorized interaction. |

**\***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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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