**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 | Invalid input data can lead to unexpected behavior and program failure. For instance, if a program is expecting an integer as an input value, entering a letter would represent an error. Therefore, input data must be validated to ensure that programs behave correctly. |
| 1. Heed Compiler Warnings | Developers must not ignore compiler warnings. Although code can be compiled and executed in the absence of errors, compiler warnings are often a sign of bugs that can lead to security flaws. |
| 1. Architect and Design for Security Policies | An application’s architecture and design should be aligned with the security policies required. One example of architecture and design based on security policy could be compartmentalizing the software according to different levels of access privileges. |
| 1. Keep It Simple | The complexity of security mechanisms is directly correlated to the complexity of the application’s design. Therefore, keeping the design simple will mitigate the possibility of errors and security flaws. |
| 1. Default Deny | Controlling user access to a system based on exclusion, meaning allowing access by default and excluding specific users, increases the system’s vulnerability. Instead, users should be denied access by default unless they meet specific criteria for permission. |
| 1. Adhere to the Principle of Least Privilege | Programs should always execute processes with the least privilege possible for the operation. This approach prevents scenarios where attackers need elevated privileges to exploit a vulnerability. |
| 1. Sanitize Data Sent to Other Systems | Exploiting vulnerabilities in the communication between systems, such as SQL injection, is a common type of attack. Therefore, sanitizing the data sent to other systems is crucial to secure coding. A program should be able to identify and escape injection characters and commands before sending data to other systems. |
| 1. Practice Defense in Depth | Since there is no security mechanism that addresses all vulnerabilities and is completely secure, it is necessary to practice defense in depth. Combining multiple strategies and layers of defense mechanisms blocks and delays attackers in different levels. Examples of these layers are policy, configuration, authentication, and authorization management. |
| 1. Use Effective Quality Assurance Techniques | To verify that the mechanisms used to protect a system from attacks are working, it is necessary to use effective QA techniques. For example, tests can be performed to verify that defense mechanisms are effective against injection and overflow attacks, as well as unauthorized access to sensitive data. |
| 1. Adopt a Secure Coding Standard | Adopting secure coding standards is essential to developing secure programs. These standards help to mitigate errors and vulnerabilities, and to ensure that security concerns are taken into account during the development of the code. |

## 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** | **Never qualify a reference type with const or volatile** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Only non-reference types can be qualified as const or volatile. Otherwise, trying to qualify reference type with const or volatile will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In the code below, the reference type is qualified as const. This leads to undefined behavior and, depending on the compiler, the value referenced by b is modified. |
| int a = 1;  int& const b = a;  b = 2; |

| **Compliant Code** |
| --- |
| There are different ways to make this code compliant. First, if b should be modified, we can remove the const qualification. Otherwise, we can change the position of const to inform the compiler that the value referenced by b should be constant. |
| int a = 1;  int& b = a;  b = 2;  // Or: const int& b = a;  // Which is the same as: int const& b = a;  // if b should not be modified |

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

|  |
| --- |
| **Principles(s):**  2. Heed Compiler Warnings  10. Adopt a Secure Coding Standard  Depending on the compiler, a warning can be shown for this issue, which should not be ignored by the developer. Adopting a secure coding standard will prevent undefined behavior and unexpected results when using references. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++0014 |  |
| Klocwork | 2021.1 | CERT.DCL.REF\_TYPE.CONST\_OR\_VOLATILE |  |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-DCL52-a | Never qualify a reference type with 'const' or 'volatile' |
| Clang | 3.9 | [Insert text.] | Clang checks for violations of this rule and produces an error without the need to specify any special flags or options. |

### Coding Standard 2

| **Coding Standard** | **Label** | **Do not cast to an out-of-range enumeration value** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Enumerators are used to group and define a range to a specific set of integer values. Casting to an out-of-range enumeration value will result in unspecified value and lead to unspecified behavior. |

| **Noncompliant Code** |
| --- |
| In the noncompliant code below, the enum Position is unscoped and the range is only checked after the value is cast to the enumeration type. The value being cast can be out-of-range and result in unspecified value and unspecified behavior. |
| enum Position {  First,  Second,  Third  };    void f(int intVar) {  Position position = static\_cast<Position>(intVar);    if (position < First || position > Third) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| In the compliant code below, the enum Position is scoped, which makes it safer. Furthermore, the range check is done before the value is cast. |
| enum class Position {  First,  Second,  Third  };    void f(int intVar) {  if (intVar < Position::First || intVar > Position::Third) {  // Handle error  }  Position position = static\_cast<Position>(intVar);  } |

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

|  |
| --- |
| **Principles(s):**  1. Validate Input Data  10. Adopt a Secure Coding Standard  Validating input data, as shown in the compliant code, is necessary to prevent unspecified behavior from casting to an out-of-range enum value. Adopting a secure coding standard will prevent unspecified behavior and unexpected results when working with enumerations. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-INT50 |  |
| Helix QAC | 2021.2 | C++3013 |  |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-INT50-a | An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration |
| PVS-Studio | 7.13 | V1016 |  |

### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Not guaranteeing that a variable has enough space for characters and strings can result in a buffer overflow. |

| **Noncompliant Code** |
| --- |
| Since the input is unbounded in the code below, a buffer overflow could occur if the user enters more characters than the buf bounded variable can store. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| Bounding the input is one way of preventing from buffer overflow, however it can truncate the string and lose input information. Therefore, using std::string, as shown in the compliant code below, is the best option, as it guarantees enough storage for the string, preventing buffer overflow and avoiding truncation. |
| #include <iostream>  #include <string>    void f() {  std::string input;  std::string stringOne, stringTwo;  std::cin >> stringOne >> stringTwo;  } |

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

|  |
| --- |
| **Principles(s):**  4. Keep It Simple  10. Adopt a Secure Coding Standard  Buffer overflow is one of the most common vulnerabilities exploited by attackers. Adopting a secure coding standard is crucial to preventing buffer overflow from happening. Furthermore, the compliant code above uses std::string instead of char with a predefined length, which is safer and easier to implement, preventing buffer overflow as well as truncation. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | MISC.MEM.NTERM, LANG.MEM.BO, LANG.MEM.TO | No space for null terminator,  Buffer overrun,  Type overrun |
| Klocwork | 2021.1 | NNTS.MIGHT, NNTS.TAINTED |  |
| Parasoft | 2021.1 | CERT\_CPP-STR50-b, CERT\_CPP-STR50-c, CERT\_CPP-STR50-e, CERT\_CPP-STR50-f, CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string,  Avoid overflow when writing to a buffer,  Prevent buffer overflows from tainted data,  Avoid buffer write overflow from tainted data,  Do not use the 'char' buffer to store input from 'std::cin' |
| Helix QAC | 2021.2 | C++2835, C++2836, C++2839, C++5216 |  |

### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize query string to prevent SQL injection using “’ or 1=1” approach** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | A common SQL injection attack is inserting “’ or 1=1” as part of the input value. If this attack is not prevented, the query can return sensitive information. It is important to also note that not only 1=1 can be used, but any value such as ‘value’=’value’. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below sends the unaltered query string to the database. This vulnerability can be exploited, and sensitive data can be retrieved with an SQL injection attack. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  records.clear();  char\* error\_message;  if(sqlite3\_exec(db, sql.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  sqlite3\_free(error\_message);  return false;  }  return true;  } |

| **Compliant Code** |
| --- |
| The code below verifies if the query string contains “’ or “ and “=”, which are indications of an SQL attack. The suspicious characters are then removed, rendering the sanitized query string invalid and returning an error from the database. |
| bool run\_query(sqlite3\* db, const std::string& sql, std::vector< user\_record >& records)  {  records.clear();  std::string sanitizedSQL = sql;  bool isSQLinjection = false;  const char posOr = sanitizedSQL.find("' or ");  if (posOr < sanitizedSQL.length()) {  isSQLinjection = true;  sanitizedSQL.erase(posOr, 1);  const char posEqual = sanitizedSQL.substr(posOr).find('=') + posOr;  if (posEqual < sanitizedSQL.length()) {  sanitizedSQL.erase(posEqual, 1);  }  }  char\* error\_message;  if(sqlite3\_exec(db, sanitizedSQL.c\_str(), callback, &records, &error\_message) != SQLITE\_OK)  {  std::cout << "\nData failed to be queried from USERS table. ERROR = " << error\_message << std::endl;  if (isSQLinjection) {  std::cout << "Indication of SQL Injection: " << sql.substr(sql.find("'")) << std::endl;  }  sqlite3\_free(error\_message);  return false;  }  return true;  } |

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

|  |
| --- |
| **Principles(s):**  1. Validate Input Data  7. Sanitize Data Sent to Other Systems  8. Practice Defense in Depth  Validating input data and sanitizing data sent to other systems is essential in preventing SQL injection attacks, which are also one of the most common vulnerabilities exploited by attackers. Input data should be sanitized to prevent malicious commands to be executed by the database and sensitive information to be retrieved by attackers. This added layer of protection between the application and the database is an essential practice of defense in depth. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ Test | 2020.1 | CWE-89-a | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') |

### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Reading a pointer to memory that has been deallocated (dangling pointer) results in undefined behavior. Therefore, freed memory should not be accessed. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below tries to access the person’s hello() function after person was deleted and memory was freed. This results in undefined behavior. |
| #include <new>    struct Person {  void hello();  };    void f() noexcept(false) {  Person \*person = new Person;  delete person;  person->hello();  } |

| **Compliant Code** |
| --- |
| In the compliant code below, person is only deleted after the hello() function is accessed. |
| #include <new>    struct Person {  void hello();  };    void f() noexcept(false) {  Person \*person = new Person;  person->hello();  delete person;  } |

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

|  |
| --- |
| **Principles(s):**  2. Heed Compiler Warnings  10. Adopt a Secure Coding Standard  Depending on the compiler, a warning or error can be shown for this issue, which should not be ignored. Adopting a secure coding standard will prevent undefined behavior from accessing freed memory. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | dangling\_pointer\_use |  |
| Helix QAC | 2021.2 | C++4303, C++4304 |  |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Polyspace Bug Finder | R2021a | CERT C++: MEM50-CPP | Pointer access out of bounds,  Deallocation of previously deallocated pointer,  Use of previously freed pointer  Rule partially covered. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Expressions used in assertions must not produce side effects** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions are great mechanisms for ensuring that the program behaves as expected. However, since assertions are normally removed from the final code for release builds, expressions that have side effects will also be removed, which can lead to a bug. |

| **Noncompliant Code** |
| --- |
| The expression used in the assertion below has a side effect as it removes an item from the set. |
| #include <assert.h>  std::set<int> myset;  // insert some values:  for (int i=1; i<10; i++) myset.insert(i\*10);  assert(myset.erase(40) == 1); |

| **Compliant Code** |
| --- |
| The compliant expression below also compares the number of items erased from the set. However, a variable is used to store the data, which is then used in the assertion. |
| #include <assert.h>  std::set<int> myset;  // insert some values:  for (int i=1; i<10; i++) myset.insert(i\*10);  unsigned char num\_erased = myset.erase(40);  assert(num\_erased == 1); |

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

|  |
| --- |
| **Principles(s):**  9. Use Effective Quality Assurance Techniques  Quality Assurance techniques must be effective and not produce unintended consequences. The assertion technique used in the noncompliant code example would result in an unexpected bug that could lead to more serious issues. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Low | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.5 | CWE: 398, Id: assertWithSideEffect | Assert statements are removed from release builds so the code inside assert statement is not executed. If the code is needed also in release builds, this is a bug. |

### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | When an exception is not handled, the program will terminate abruptly. Attacks such as DoS commonly exploit abnormal termination of applications. Therefore, it is necessary to handle all exceptions thrown and terminate the program in a controlled manner. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below does not handle exceptions that can be thrown by throwing\_func(), which can result in the program terminating abnormally. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In the compliant code below, a try/catch statement is used to ensure that all exceptions are handled by the program. This allows for an improved management of any issues that may arise. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

|  |
| --- |
| **Principles(s):**  2. Heed Compiler Warnings  4. Keep It Simple  10. Adopt a Secure Coding Standard  Depending on the compiler, a warning can be shown for this issue, which should not be ignored. Handling all exceptions can be simple according to each case. Adopting a secure coding standard prevents the program from abruptly stop executing because of unhandled exceptions. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 20.10 | main-function-catch-all early-catch-all | Partially checked |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2021.2 | C++4035, C++4036, C++4037 |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | CERT\_CPP-ERR51-a, CERT\_CPP-ERR51-b | Always catch exceptions Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2021a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |

### Coding Standard 8

| **Coding Standard** | **Label** | **Value-returning functions must return a value from all exit paths** |
| --- | --- | --- |
| **Functions** | STD-008-CPP | An exit path that does not return a value in a value-returning function results in undefined behavior. Therefore, all exit paths must return a value in these types of functions. |

| **Noncompliant Code** |
| --- |
| In the example below, there is only one return statement, which is inside the if statement. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| This compliant example returns values in all exit paths, inside and outside of the if statement. |
| 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):**  2. Heed Compiler Warnings  4. Keep It Simple  10. Adopt a Secure Coding Standard  Similar to the previous standard, depending on the compiler, a warning can be shown for this issue, which should not be ignored. Moreover, returning values from all exit paths can be made simple and readable. Adopting a secure coding standard prevents undefined behavior. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | return-implicit | Fully checked |
| Clang | 3.9 | -Wreturn-type | Does not catch all instances of this rule, such as function-try-blocks |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MSC52-a | All exit paths from a function with non-void return type shall have an explicit return statement with an expression |
| Polyspace Bug Finder | R2021a | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |

### Coding Standard 9

| **Coding Standard** | **Label** | **Do not read uninitialized memory** |
| --- | --- | --- |
| **Memory Protection** | STD-009-CPP | Variables must be initialized before the memory can be read. Otherwise, reading uninitialized memory results in undefined behavior. |

| **Noncompliant Code** |
| --- |
| This noncompliant example declares a variable i but does not initialize it. Reading from it will result in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| The compliant code below assigns 0 to the variable i, therefore, there are no issues in reading the variable. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

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

|  |
| --- |
| **Principles(s):**  2. Heed Compiler Warnings  10. Adopt a Secure Coding Standard  Depending on the compiler, a warning or error can be shown for this issue, which should not be ignored. Adopting a secure coding standard will prevent undefined behavior that would result from accessing uninitialized memory. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Klocwork | 2021.1 | UNINIT.CTOR.MIGHT, UNINIT.CTOR.MUST, UNINIT.HEAP.MIGHT, UNINIT.HEAP.MUST, UNINIT.STACK.ARRAY.MIGHT, UNINIT.STACK.ARRAY.MUST, UNINIT.STACK.ARRAY.PARTIAL.MUST, UNINIT.STACK.MIGHT, UNINIT.STACK.MUST |  |
| Astrée | 20.10 | uninitialized-read | Partially checked |
| RuleChecker | 20.10 | uninitialized-read | Partially checked |

### Coding Standard 10

| **Coding Standard** | **Label** | **Detect errors when converting a string to a number** |
| --- | --- | --- |
| **Exceptions** | STD-010-CPP | Converting strings to numbers can result in errors; therefore, it is important to throw and handle exceptions when these issues happen. This prevents unexpected and undefined behavior. |

| **Noncompliant Code** |
| --- |
| The noncompliant code below defines two integer variables i and j, and stores the values read from the input stream in them. However, it does not take into account that errors can occur when converting the input string to an integer. |
| #include <iostream>    void f() {  int i, j;  std::cin >> i >> j;  } |

| **Compliant Code** |
| --- |
| The compliant code below also defines two integer variables to store the values read from the input stream. But, in this case, failbit and badbit throw exceptions and the try/catch statement handles these failures. |
| #include <iostream>    void f() {  int i, j;  std::cin.exceptions(std::istream::failbit | std::istream::badbit);  try {  std::cin >> i >> j;  } catch (std::istream::failure &E) {  // Handle error  }  } |

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

|  |
| --- |
| **Principles(s):**  1. Validate Input Data  9. Use Effective Quality Assurance Techniques  10. Adopt a Secure Coding Standard  Validating input data ensures that the program behaves as expected. Using a try/catch statement is an effective QA technique that accounts for expressions that might result in error and need to be handled to ensure correct behavior. Adopting a secure coding standard will prevent unexpected and undefined behavior that would result from failing to convert a string to a number. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-err34-c | Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2021.2 | **C++3161** |  |
| [Klocwork](https://www.securecoding.cert.org/confluence/display/cplusplus/Klocwork) | 2021.1 | [CERT.ERR.CONV.STR\_TO\_NUM](https://support.roguewave.com/documentation/klocwork/en/current/certcandcsecurecodingstandardidsmappedtoklocworkcandccheckers/) |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2021.1 | **CERT\_CPP-ERR62-a** | The library functions atof, atoi and atol from library stdlib.h shall not be used |

## 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.

DevOps is a great process that focuses on collaboration through development and deployment stages that allows for frequent and efficient product deliveries. DevSecOps extends DevOps by also addressing security concerns throughout the process instead of waiting until the end of the software development lifecycle. In pre-production, security factors are evaluated during planning, design, build, and testing stages. This involves analyzing the software’s threat landscape, following development best practices and processes such as test-driven development, performing secure build, using trusted repositories, as well as performing vulnerability scanning and security testing. In production, the application is deployed, and vulnerabilities are monitored, which is done through penetration testing, log collection event alerting, and intrusion detection. In case an attack is detected, a response is needed to block the attack and maintain and stabilize the application, restarting the process. Automation is an important part of DevSecOps, allowing the process to be efficient and developers to focus on more important aspects of security. Static Application Security Testing (SAST) is useful to analyzing vulnerabilities without having to compile the code, while Dynamic Application Security Testing (DAST) makes it possible to identify vulnerabilities while the application is running. Interactive Application Security Testing (IAST) combines SAST and DAST. Additionally, Runtime Application Self-Protection (RASP) is an automated tool that can detect and block attacks in real-time using data from the application. Assessing risks and vulnerabilities must be done on a regular basis to prevent threats and assess changes in the threat landscape.

## 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 | Low | Unlikely | Low | Low | 3 |
| STD-002-CPP | Medium | Unlikely | Medium | Medium | 3 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | Low | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Low | 3 |
| STD-008-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-009-CPP | High | Probable | Medium | High | 1 |
| STD-010-CPP | Medium | Medium | Medium | Medium | 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 at rest | The encryption of stored data (at rest) is essential to cyber security, and it can be achieved through file and disk encryption tools. This way, even if an attack captures the data, it will be unreadable unless the attacker also obtained the encryption key. Therefore, encryption keys must be stored in safe locations. |
| Encryption in flight | Data in transit between systems and networks should also be encrypted to avoid interception attacks from being able to read sensitive data. Protocols and encrypted tunnels, such as HTTPS, SSL, and VPNs must be used to protect the data in motion. |
| Encryption in use | To protect data while it is being processed and read from memory, encryption in use is needed. This requires new forms of technology, such as confidential computing, which functions as a black box that only allows authorized programs and users to read the encrypted data being processed. Encryption in use is ideal to prevent malicious attacks, internal threats, and business partners to access sensitive information in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Identifying users is an elementary step to ensure that systems are secure. User authentication is commonly implemented through unique usernames and passwords. |
| Authorization | Once authenticated, users should be authorized to only access predefined levels of information, such as authorized databases. Furthermore, users should only be allowed to perform specific actions. For instance, only authorized users should be allowed to add new users and grant them permissions. |
| Accounting | To avoid access violations and attacks, accounting should be used to monitor user activity. Generating log reports and monitoring activity help mitigate threats by identifying the resources being accessed, the users accessing them, the time of access, and actions being performed. |

**\***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 | 08/08/2021 | Updated Coding Standards, Risk Assessments and Policies | Diego Bez Zambiazzi |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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