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



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## Overview

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

## Purpose

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

## Scope

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

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Input validation is used to ensure that any received input conforms to expected forms. This validation includes numerous methods, including enforcing character limits, validating that input matches expectations of character type, etc. This prevents against damage from both accidents and attacks, such as SQL injection and buffer overflows. |
| 1. Heed Compiler Warnings | Compilers display warnings to alert programmers to issues within the code that are not necessarily errors but still may need to be addressed. Adjust compiler settings to use the highest level of warnings, and modify code accordingly to address all possible warnings. |
| 1. Architect and Design for Security Policies | Design and develop code with security policies in mind, in order to have security built-in from the beginning. Doing so helps to prevent security from being added in as an afterthought, or requiring massive code overhauls to bring the code in line with security policies. |
| 1. Keep It Simple | Avoid unnecessary complexity by keeping code as simple as is reasonable. This helps with the ease of maintaining, updating, and modifying the code, which will necessarily aid in keeping code secure and up-to-date with security standards. |
| 1. Default Deny | Deny by default is a form of access control that, as a rule, denies access requests that are not specifically allowed. For instance, if an administrator creates a new user account, that account should have minimal or no access until the account is configured to receive access. |
| 1. Adhere to the Principle of Least Privilege | The principle of least privilege requires that users, programs, and processes should have only the privileges that they require to achieve their tasks, and nothing more. No unnecessary privileges should be held. This helps to prevent both accidental and malicious abuse of privileges. |
| 1. Sanitize Data Sent to Other Systems | Data sanitization refers to the practice of sanitizing data such that it conforms to the expectations of the receiving system. This practice also involves verifying that no unnecessary sensitive information is being passed inappropriately. This way, the receiving system does not inappropriately receive sensitive information, but also receives data in a form it can accept and parse. |
| 1. Practice Defense in Depth | Defense in depth is the practice of layering distinct security measures, so that their vulnerabilities do not overlap. That is, if one layer is vulnerable to a certain method of attack, the next layer is not. Thus attackers have to penetrate several layers of security before gaining access to a system. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques, such as third-party reviews of systems and security and multiple rigorous testing phases, can greatly aid in the delivery of high-quality and secure code. |
| 1. Adopt a Secure Coding Standard | Define a standard for developing secure code that addresses vulnerabilities inherent in the code and keep both the code and the coding standard up to date with changes in security. |

### 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 |

| **Noncompliant Code** |
| --- |
| The following code declares an integer x in file 1, but also declares a string x in file 2. This results in undefined behavior. |
| // In file 1  int x;  int func()  {  return x;  }  // In file 2  string x; |

| **Compliant Code** |
| --- |
| The following code declares an integer x in file 1, and declares an integer x again in file 2. |
| // In file 1  int x;  int func()  {  return x;  }  // In file 2  int x; |

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

| **Principles(s):** 4. Keep it simple, and 10. Adopt a secure coding standard. Coding best practices include using descriptive names for variables, and if a variable is defined or declared in more than one location, it should be the same type. This will help to prevent undefined behavior and keep the code simple and secure. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | type-compatibility  type-compatibility-link  distinct-extern | Fully checked |
| Helix QAC | 2024.2 | C0776, C0778, C0779, C0789, C1510  C++1510 | Fully implemented |
| RuleChecker | 24.04 | type-compatibility  type-compatibility-link  distinct-extern | Fully checked |
| TrustInSoft | 1.38 | incompatible declaration | Exhaustively verified. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Ensure that operations on signed integers do not result in overflow |

| **Noncompliant Code** |
| --- |
| The following code can result in an overflow error when adding x and y. |
| int func(signed int x, signed int y)  {  signed int total = x + y; // Could result in an overflow error  // ...  } |

| **Compliant Code** |
| --- |
| In the following code, if adding x and y would result in an overflow, the code handles the error instead of allowing it. If not, x and y are added as normal. |
| #include <limits.h>  int func(signed int x, signed int y)  {  // Check for overflow condition  if (((x > 0) && (y > (INT\_MAX - x)) ||  ((x < 0) && (y < (INT\_MAX - x))))  {  // Overflow error handler here  } else {  signed int total = x + y;  }  // ...  } |

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

| **Principles(s):** 2. Heed compiler warnings, and 3. architect and design for security policies. Beware of logic that could result in overflow errors, and prevent them by employing robust error-handling. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | integer-overflow | Fully checked |
| Parasoft C/C++test | 2023.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid signed integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2024a | CERT C: Rule INT32-C | Checks for:  Integer overflow  Tainted division operand  Tainted modulo operand  Rule partially covered. |
| TrustInSoft Analyzer | 1.38 | signed\_overflow | Exhaustively verified |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Range check element access |

| **Noncompliant Code** |
| --- |
| The following code results in undefined behavior if mystring is empty. |
| #include <string>  void capitalizeAString(std::string &mystring)  {  std::capitalized\_string = toUpper(mystring);  } |

| **Compliant Code** |
| --- |
| The following code verifies that a string is not empty before attempting to capitalize the string. |
| #include <string>  void capitalizeAString(std::string &mystring)  {  if (mystring.empty()) {  return;  }  std::capitalized\_string = toUpper(mystring);  } |

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

| **Principles(s):** 3. Architect and design for security policies, and 10. adopt a secure coding standard. When implementing logic that accesses indices within a string, ensure that an out of range error will not occur by wrapping potentially error-prone code in error-handling logic. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | assert\_failure |  |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2024a | CERT C++: STR53-CPP | Checks for:  Array access out of bounds  Array access with tainted index  Pointer dereference with tainted offset  Rule partially covered. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Exclude user input from format strings |

| **Noncompliant Code** |
| --- |
| This code prints whatever string is called alongside it when the code is executed. This can result in the user passing in format string modifiers that access memory. |
| #include <stdio.h>    void main(int argc, char \*\*argv)  {  printf(argv[1]);  } |

| **Compliant Code** |
| --- |
| The following code does not interpret arguments in the input string, so the code is safe. |
| #include <stdio.h>  void main(int argc, char \*\*argv)  {  printf("%s\n", argv[1]);  } |

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

| **Principles(s):** 1. Validate input data. User input is dangerous and should not be trusted by default. Prevent potentially dangerous user input from damaging systems by implementing strong user input validation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| LDRA tool suite | 9.7.1 | 86 D | Partially Implemented |
| Parasoft C/C++test | 2023.1 | CERT\_C-FIO30-a  CERT\_C-FIO30-b  CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable  Never use unfiltered data from an untrusted user as the format parameter |
| Polyspace Bug Finder | R2024a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| The following example attempts to create a new allocation in a try block. However the allocation never occurs because an error is thrown before. The catch block then executes and attempts to delete unallocated memory. |
| #include <new>  void myfunc()  {  int \*xyz;  try {  throw;  \*xyz = new int;  }  catch (...) {  delete \*xyz;  }  } |

| **Compliant Code** |
| --- |
| In this code, xyz has been initialized to nullptr, so that when the catch block is executed, the delete function will delete memory that has been allocated. |
| #include <new>  void myfunc()  {  int \*xyz = nullptr;    try {  throw;  \*xyz = new int;  }  catch (...) {  delete \*xyz;  }  } |

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

| **Principles(s):** 10. Adopt a Secure Coding Standard. Memory allocation and deallocation should be handled with care. When allocating and deallocating memory, be careful to do so securely, and avoid situations where deletions are attempted erroneously. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks  -Wmismatched-new-delete  clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 8.1p0 | ALLOC.FNH  ALLOC.DF  ALLOC.TM  ALLOC.LEAK | Free non-heap variable  Double free  Type mismatch  Leak |
| LDRA tool suite | 9.7.1 | 232 S, 236 S, 239 S, 407 S, 469 S, 470 S, 483 S, 484 S, 485 S, 64 D, 112 D | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b  CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Use the same form in corresponding calls to new/malloc and delete/free  Always provide empty brackets ([]) for delete when deallocating arrays  Both copy constructor and copy assignment operator should be declared for classes with a nontrivial destructor  Properly deallocate dynamically allocated resources |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use a static assertion to test the value of a constant expression |

| **Noncompliant Code** |
| --- |
| Because the assertion has been placed in a function, it will only run if the function is called. |
| #include <assert.h>  struct mynumber {  int x;  };  void myfunc(int x) {  assert(sizeof(struct mynumber) == sizeof(int));  } |

| **Compliant Code** |
| --- |
| The example below includes a static\_assert, which will be checked at compile time, removing the possibility of a silent malfunction or runtime error. |
| #include <assert.h>    struct mynumber {  int x;  };  static\_assert(sizeof(struct mynumber) == sizeof(int)); |

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

| **Principles(s):** 9. Use Effective Quality Assurance Techniques. When using assertions, ensure that they are implemented correctly by employing static\_assert for constant expressions. These assertions will be verified, even if the specific block of code that they are in is not executed (such as an unused if-else branch or function call). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| main() does not catch thrown exceptions, so when my\_throwing\_func() throws an exception, it will not be caught and the program will instead call terminate(). |
| void my\_throwing\_func()  {  throw;  }  int main()  {  my\_throwing\_func();  } |

| **Compliant Code** |
| --- |
| In the following code, main() catches any exceptions and handles them, so the program will not terminate needlessly. |
| void my\_throwing\_func()  {  throw;  }  int main()  {  try {  my\_throwing\_func();  }  catch (...) {  // Error handler  }  } |

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

| **Principles(s):** 8. Practice Defense in Depth. Implement robust and flexible error-handling throughout the code to ensure that the program does not terminate unexpectedly and instead handles errors gracefully. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | main-function-catch-all  early-catch-all | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| RuleChecker | 22.10 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Prefer special member functions and overloaded operators to C Standard Library functions |

| **Noncompliant Code** |
| --- |
| The following code attempts to use the equality operator to check whether the values of two floats are equal. This is bad practice for comparing float values. |
| #include <cstring>    class MyClass  {  float x;    public:  virtual void func();    // ...  };  void func(MyClass &c1, MyClass &c2)  {  if (c1 == c2)  {  // ...  }  } |

| **Compliant Code** |
| --- |
| In the following code, the equality operator has been overloaded to set a threshold for considering the two floats to be equal. |
| #include <cstring>    class MyClass  {  float x;    public:  virtual void func();    bool operator==(const MyClass &c) const  {  if ((c.x - x) < 0.0001)  {  return true;  }  return false;  }  // ...  };  void func(MyClass &c1, MyClass &c2)  {  if (&c1 == &c2)  {  // ...  }  } |

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

| **Principles(s):** 4. Keep it simple, and 7. sanitize data sent to other systems. When interfacing between different parts of the program, ensure that objects are being instantiated properly. When possible, use special member functions and overloaded operators to guarantee correct instantiation and use of objects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use-ato  stdlib-use  stdlib-use-getenv  stdlib-use-system  include-time  stdlib-use-string-unbounded | Partially checked |
| CodeSonar | 8.1p0 | BADFUNC.MEMCMP  BADFUNC.MEMSET | Use of memcmp  Use of memset |
| LDRA tool suite | 9.7.1 | 44 S | Enhanced Enforcement |
| Polyspace Bug Finder | R2024a | CERT C++: OOP57-CPP | Checks for bytewise operations on nontrivial class object (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Guarantee that container indices and iterators are within the valid range |

| **Noncompliant Code** |
| --- |
| The function myfunc() accepts an integer pos, which could be a negative value, resulting in writing outside the bounds of memory referenced by the array. |
| void myfunc(int size, int pos)  {  int myarray[size];  if (pos <= size)  {  myarray[pos] = pos;  }  } |

| **Compliant Code** |
| --- |
| Here, the argument pos has been declared as size\_t, which prevents negative arguments from being passed. |
| void myfunc(int size, std::size\_t pos)  {  int myarray[size];  if (pos <= size)  {  myarray[pos] = pos;  }  } |

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

| **Principles(s):** 3. Architect and design for security policies, and 10. adopt a secure coding standard. When attempting to access a certain index within a container, ensure that the index is valid before attempting to access it to prevent referencing memory that is out of bounds. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TO  LANG.MEM.TU  LANG.MEM.TBA  LANG.STRUCT.PBB  LANG.STRUCT.PPE  LANG.STRUCT.PARITH | Buffer overrun  Buffer underrun  Type overrun  Type underrun  Tainted buffer access  Pointer before beginning of object  Pointer past end of object  Pointer Arithmetic |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2024a | CERT C++: CTR50-CPP | Checks for:  Array access out of bounds  Array access with tainted index  Pointer dereference with tainted offset  Rule partially covered. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | Do not read uninitialized memory |

| **Noncompliant Code** |
| --- |
| The following code attempts to output the value of mystring, but mystring has only been declared, not initialized with a value. |
| #include <iostream>    int main()  {  string mystring;  std::cout << "Here is my string: " << mystring << std::endl;  } |

| **Compliant Code** |
| --- |
| In the following code, my has been initialized prior to the expression attempting to reference its value. |
| #include <iostream>    int main()  {  string mystring = "";  std::cout << "Here is my string: " << mystring << std::endl;  } |

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

| **Principles(s):** 2. Heed compiler warnings, and 3. architect and design for security policies. Ensure that accessed memory has been initialized prior to access to prevent memory errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | uninitialized-read | Partially checked |
| Clang | 3.9 | -Wuninitialized  clang-analyzer-core.UndefinedBinaryOperatorResult | Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory. |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-EXP53-a | Avoid use before initialization |
| Polyspace Bug Finder | R2024a | CERT C++: EXP53-CPP | Checks for:  Non-initialized variable  Non-initialized pointer  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.

Just as security should be an aspect that is considered from the start and baked-in to every part of the designing, building, and releasing of a project, automation tools can assist the DevSecOps process at virtually every step of the way, throughout the software development lifecycle. Many programmers do not have much experience in software security, and may be under the mistaken impression that security is a separate issue to be addressed at the end of a project. DevSecOps automation can ensure that security is a foundational consideration of the project from the very beginning. As code is being written, automation tools can identify and help correct security vulnerabilities, so that issues can be addressed as soon they arise rather than in the last minute push before release.

Consider static application security testing (SAST) tools and dynamic application security testing (DAST) tools. SAST tools are wonderful for statically assessing written code to check for known vulnerabilities. They are capable of being used very early in the SDLC because the code does not even need to be runnable for SAST tools to be used. Because of this, static testing should be performed early and often throughout the DevSecOps cycle. On the other hand, DAST tools require a running application, and as such are more useful for a mature project with an executable program. Unlike SAST tools, DAST tools do not examine the source code directly, but rather test the application from the perspective of a hacker trying to penetrate the software. These methods can be useful for finding run-time or environment-based vulnerabilities. Both SAST and DAST are necessary for comprehensive security testing, and both should be deployed as early as possible in the development process.

Finally, even after code has been deployed, it is crucial to keep up with the ever-changing landscape or software security. This means applying security patches in a timely manner, and monitoring activity within the system to catch any suspicious behavior as early as possible.

### 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 | Medium | P2 | L3 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Unlikely | Medium | P6 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Likely | High | P9 | L2 |
| STD-010-CPP | High | Probable | Medium | P12 | L1 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the practice of securely encrypting stored data and decrypting the data upon access. Often this is accomplished with the use of a symmetric encryption key. Anyone with the key can encrypt data for storage, and access and decrypt stored data. Anyone without the key would be unable to read the encrypted data, even if they managed to access it. This practice should be utilized for any stored data that could be classified as sensitive, especially if that data is accessible through a network. |
| Encryption in flight | Encryption in flight refers to the practice of securely encrypting data for transmission. This practice helps defend against the data being intercepted while in transit. The practice involves encrypting the data before transmission, authentication at both endpoints (such as by security certificates), and decryption upon receipt. Secure communication is imperative to verify that data was not intercepted or altered while in transit, and that both the sender and receiver remain secure. |
| Encryption in use | Encryption in use refers to the practice of securely encrypting data while currently in use, and this practice is also called in-memory encryption or runtime encryption. The data is encrypted and decrypted in real time, while the data is being processed or modified. While this process is computationally expensive, it is valuable in order to protect data from being vulnerable while in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication refers to the practice of identifying an entity, and verifying that the identification is accurate. Often this is accomplished by giving users of all types (clients, admins, etc.) an account with a username or some other form of ID, and securing the account with a password or PIN. Then, in order to access the system, an entity first must provide their identity (username or ID) and verify that it is indeed them by providing a secret password or other secure token. |
| Authorization | Authorization refers to the practice of determining what permissions an account has. These permissions include what data an account is allowed to access and what it is allowed to do to that data (create, read, modify, delete, etc.). One common method of authorizing accounts is through the use of role-based access controls (RBAC), whereby roles are defined (such as user and administrator) and each role is given only the permissions necessary for that role. These roles are then applied to accounts, creating a classification system for all accounts within the system. This method works best when used in conjunction with the principle of least privilege, which states that entities should have the minimum amount of permissions required for the execution of their required tasks. In essence, an account should be given an appropriate role, and roles should not have greater access or permissions than required. |
| Accounting | Accounting refers to the process of monitoring and logging activity within the system. This includes keeping records of what account performed what activity on what data or systems, and when this activity occurred. Rigorous accounting enables thorough auditing of system activity. Suspicious activity (such as an account attempting to access data it is forbidden to access) can then be investigated and action can be taken accordingly. |

**\***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 |  |
| 1.1 | 07/28/2024 | Completed module 4 sections | Taryn Brownfield |  |
| 1.2 | 08/11/2024 | Completed module 6 sections | Taryn Brownfield |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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