**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

[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 for all developed applications. Consistent approaches and methodologies must be maintained through all uniformly defined policies, 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 procedures): [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 | Treat all input data as malicious until you can prove it is not. Ensure all data received from users or other systems is validated against the rules for data type, length, format, and all other appropriate rules. This should prevent vulnerabilities related to type mismatches, overflows, underflows, and injections. |
| 1. Heed Compiler Warnings | When finalizing builds, use the highest warning level available for your compiler. These warnings can help identify potential issues in your code. Eliminate all errors and warnings before finalizing the application. Warnings are typically issued for uninitialized variables, type mismatches, or unsafe constructs. Applications built with no warnings will be more robust and less vulnerable than applications where warnings remain. |
| 1. Architect and Design for Security Policies | Consider software architecture and design when implementing security policies, such as separating a system into sub-systems with different authorization or privilege levels. Using a zero-trust approach will guarantee that permissions are intentionally granted. Design systems with breaches in mind to minimize risk and damage should a security measure fail. |
| 1. Keep It Simple | Revise systems to remove complexity. The least complex version will likely have the fewest vulnerabilities. The simplest version will also benefit from others quickly understanding it and pointing out flaws. |
| 1. Default Deny | This security model denies all access to a system by default. It is the opposite of a traditional model, which allows everything until configured to deny something. If suitable, default deny has fewer security rules and should reduce the attack surface available to hackers. |
| 1. Adhere to the Principle of Least Privilege | Processes should be executed with the lowest possible privileges needed for the task. Elevated privileges should be active for only the tasks requiring them, then reduced to lower the chances of elevated privileges being able to execute code at an elevated privilege level. |
| 1. Sanitize Data Sent to Other Systems | Systems containing unused functions or out-of-context calls provide opportunities for injection attacks. Ensuring that all data passed to other systems is sanitized is crucial. This practice prevents injection attacks from being successful against command shells, relational databases, or commercial off-the-shelf components by supplying them with clean and safe data. |
| 1. Practice Defense in Depth | Build defensive strategies in multiple layers. Do not rely on a single layer of security to stop a threat. Employ a comprehensive approach that considers software, hardware, and people. Understand and leverage traditional tools and strategies while considering more modern and sophisticated techniques. A multi-layer defense may require both. |
| 1. Use Effective Quality Assurance Techniques | The quality assurance process should verify that software and systems meet established security requirements and standards. Testing should be executed with vulnerabilities, weaknesses, and potential threats in mind. This principle should also address regulatory compliance, legal requirements, and auditing. |
| 1. Adopt a Secure Coding Standard | Establish a standard for each language and platform used to set expectations and requirements for the development team. Standards prevent late-in-development confusion, questions, and oversights. |

### 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 from 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] | When choosing a data type, unsigned integer values should represent values that  can’t become negative, and signed integer values should be used for values that can become negative. In general, use the smallest signed or unsigned type that can  fully represent the range of possible values for the given variable, as these conserve memory. |

| **Noncompliant Code** |
| --- |
| Truncation can occur when a value is too small to represent the result, and conversions can result in values out of  range in the resulting type. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 sc = (signed char)ul; /\* cast eliminates warning \*/ |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 if (ul <= SCHAR\_MAX) {  4 sc = (signed char)ul; /\* use cast to eliminate warning \*/  5 }  6 else {  7 /\* handle error condition \*/  8 } |

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

| **Principles(s):** 1) Validate Input Data, 2) Heed compiler warnings, 3) Architect and Design for Security Policies, 9) Effective Quality Assurance techniques. Proper validation can prevent memory errors. Heeding compiler warnings will catch errors before production. A zero-trust approach will ensure data is from an authorized and authenticated user account. Using QA techniques will visualize this type of error in the pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.12 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Helix QAC | 2023.2 | C2800, C2801, C2802, C2803, C2860,  C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803,  C++2860, C++2861, C++2862, C++2863 | Implemented |
| Parasoft C/C++test | 2023.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow in  constant expression in '+', '-', '\*'  operator  Integer overflow or underflow in  constant expression in '<<' operator |
| TrustInSoft  Analyzer | 1.47 | Signed\_overflow | Exhaustively verified |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CP  P] | When choosing a data type, unsigned integer values should represent values that  can’t become negative, and signed integer values should be used for values that can become negative. Use the smallest signed or unsigned type to fully represent  the range of possible values for the given variable, as this conserves memory. |

| **Noncompliant Code** |
| --- |
| This code fails to consider that the unsigned integer value will result in an infinite loop. |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3 for (unsigned int i = cnt-2; i >= 0; i--) {  4 a[i] += a[i+1];  5 } |

| **Compliant Code** |
| --- |
| As size\_t is an unsigned type, this behavior is well defined by the standard to be modulo. |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3 for (size\_t i = cnt-2; i != SIZE\_MAX; i--) {  4 a[i] += a[i+1];  5 } |

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

| **Principles(s):** 1) Validate Input Data to catch these mistakes from the onset, and 9) Use Effective Quality Assurance  Techniques to mitigate these problems via analysis tools. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.12 | INTEGER\_OVERFLOW | Implemented |
| CodeSonar | 8.0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation  size  Subtraction underflow of allocation  size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Parasoft  C/C++test | 2023.2 | CERT\_C-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoid 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 INT30-C | Unsigned integer overflow  Unsigned integer constant overflow |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CP  P] | Incorrect string sizes and neglected buffer boundaries can lead to buffer overflows  and runtime errors. Never copy data from an unbounded source such as stdin into a fixed-length array. |

| **Noncompliant Code** |
| --- |
| C++: If a user inputs more than 11 characters, the result is out-of-bounds |
| 1 #include <iostream>  2 int main(void) {  3 char buf[12];  4 std::cin >> buf;  5 std::cout << "echo: " << buf << '\n';  6 } |

| **Compliant Code** |
| --- |
| Eliminates the overflow in the previous example by setting the field width member to the size of the character array buffer. |
| 1 #include <iostream>  2 int main(void) {  3 char buf[12];  4 std::cin.width(12);  5 std::cin >> buf;  6 std::cout << "echo: " << buf << '\n';  7 }  8 string str;  9 string::iterator i;  10 for (i = str.begin(); i != str.end(); ++i) {  11 cout << \*i  12 } |

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

| **Principles(s):** 1) Validate input with appropriate string function, 2) Heed compiler warnings of overflow, 7) Sanitize  data to prevent string attacks, and 9) QA Techniques to catch these errors on build in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that  report uses of library functions prone  to internal buffer overflows |
| Coverity | 2022.12 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Parasoft  C/C++test | 2023.2 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a  buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from  tainted data  Avoid using unsafe string functions  which may cause buffer overflows |
| TrustInSoft  Analyzer | 1.47 | Mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CP  P] | Attack vectors for strings include command-line arguments, environmental  variables, console input, text files, and network connections. These are all vectors  in which an attacker can launch strategic SQL injection-based attacks to cause overflow. Secure Coding in C and C++ states, “String concatenation is the primary point of entry for script injection.” Test the size / data type of input and enforce appropriate limits, rejecting entries containing binary data, escape sequences, and comment characters. Review all code that calls EXECUTE, EXEC, or sp\_executesql. Always use Parameterized Queries in SQL. |

| **Noncompliant Code** |
| --- |
| Unfiltered code is vulnerable to SQL injection via user input. |
| SqlDataAdapter myCommand =  new SqlDataAdapter("LoginStoredProcedure '" +  Login.Text + "'", conn); |

| **Compliant Code** |
| --- |
| Using the Parameters collection with Dynamic SQL |
| SqlDataAdapter myCommand = new SqlDataAdapter(  "SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id", conn);  SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",  SqlDbType.VarChar, 11);  Parm.Value = Login.Text; |

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

| **Principles(s):** 1) Input validation, use query parameterization, 3) Architect and design for security policies regarding code with knowledge of possible SQL injection attacks in mind, 8) Practice defense in depth, using a layered defensive strategy to prevent such attacks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes  that report uses of library  functions prone to internal buffer  overflows |
| Coverity | 2022.12 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully Implemented |
| Parasoft  C/C++test | 2023.2 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from  tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| TrustInSoft  Analyzer | 1.47 | Mem\_access | Exhaustively Verified |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CP  P] | When not using the new operator to allocate sufficient memory, memory is  thought to be manually managed and should be deallocated and destroyed in the same manual manner. An object used outside of its lifespan is undefined behavior and can lead to errors. |

| **Noncompliant Code** |
| --- |
| Manual memory management occurs due to the user-provided construction with a call to std::malloc(). The constructor for the object is never called, and this results in undefined behavior when the class is accessed later by  s->f(). |
| #include <cstdlib>  struct S {  S();  void f();  };  void g() {  S \*s = static\_cast<S \*>(std::malloc(sizeof(S)));  s->f();  std::free(s);  } |

| **Compliant Code** |
| --- |
| The constructor and destructor should both be explicitly called. |
| #include <cstdlib>  #include <new>  struct S {  S();  void f();  };  void g() {  void \*ptr = std::malloc(sizeof(S));  S \*s = new (ptr) S;  s->f();  s->~S();  std::free(ptr);  } |

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

| **Principles(s):** Architect for security involves training to write secure memory allocation routines, Adopting a Secure Coding standard to ensure this is consistent across the project, and Quality Assurance Techniques to test for these types of vulnerabilities on each scan in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2022.12 | CHECKED\_RETURN | Finds inconsistencies in how the function call return values are handled |
| Parasoft  C/C++test | 2023.2 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function  argument list because the order of  evaluation of a function's parameters  is undefined |
| Polyspace Bug  Finder | R2024a | CERT C++: MEM52-CPP | Checks for unprotected dynamic  memory allocation |
| Helix QAC | 2023.2 | C++3225, C++3226, C++3227, C++3228,  C++3229, C++4632 | It helps to target the most critical  defects using filters, suppressions,  and baselines. It delivers accurate  diagnostics and actionable results. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD--006-CPP] | Assert calls abort(); cleanup functions register with exit () are not called. This can  lead to errors in proper termination of the program leading to errors. |

| **Noncompliant Code** |
| --- |
| A function that is called before the program exits to clean up. The assert function (when it fails) will exit before cleanup. |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }  int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }  /\* ... \*/  assert(/\* Something bad didn't happen \*/);  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| If statements are used in place of assert to allow cleanup and ensure proper termination routines. |
| void cleanup(void) {  /\* Delete temporary files, restore consistent state, etc. \*/  }  int main(void) {  if (atexit(cleanup) != 0) {  /\* Handle error \*/  }  /\* ... \*/  if (/\* Something bad happened \*/) {  exit(EXIT\_FAILURE);  }  /\* ... \*/  } |

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

| **Principles(s):** Architect for security involves training to write secure memory allocation routines, adopting a Secure Coding standard to ensure consistency across the project, and Quality Assurance techniques to test for these vulnerabilities on each scan in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2022.12 | 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 | 2023.2 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CP  P] | All exceptions thrown must be caught by a matching exception handler or the stack may not unwind correctly due to std::abort() being called so destructors may  not be called. |

| **Noncompliant Code** |
| --- |
| Neither f() nor main() will catch exceptions thrown by throwing\_func(). |
| void throwing\_func() noexcept(false);  void f() {  throwing\_func();  }  int main() {  f();  } |

| **Compliant Code** |
| --- |
| Handles all exceptions, making sure the stack is unwound. |
| 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):** Architect for security involves training to use assertions in the development codebase, Adopting Secure Coding standards to ensure this is consistent across the project, and Quality Assurance Techniques to run such assertions on each scan in the CI/CD pipeline. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.2 | C++4035, C++4036, C++4037 | Implemented |
| Parasoft  C/C++test | 2023.2 | 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 | R2024a | CERT C++: ERR51-CPP | Checks for unhandled exceptions |
| RuleChecker | 21.04 | main-function-catch-all  early-catch-all | Partially checked |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-008-C PP] | Pertains to reading unintialized memory or relying on the value of a moved-from object. As local, automatic variables assume unexpected values if they are read  before initialization, which can lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>  void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| Initialize the object before printing its value. |
| #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):** Sequences of operators and operands that are used for one or more of these purposes |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 18.1.6 | Wdangling-initializer-list | Catches some lifetime issues related  to incorrect use of std::initializer\_list<> |
| CodeSonar | 8.0 | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Parasoft  C/C++test | 2023.2 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | Do not use resources that have been  freed  The address of an object with automatic storage shall not be returned from a function  The address of an object with  automatic storage shall not be  assigned to another object that may  persist after the first object has ceased to exist |
| Polyspace Bug  Finder | R2024a | CERT C++: EXP54-CPP | Non-initialized variable or pointer  Use of previously freed pointer  Pointer or reference to stack  variable leaving scope  Accessing object with temporary  lifetime |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Polymorphic  Object | [STD-009-CPP] | Deleting an object through pointer to a type without a virtual destructor results in  undefined behavior |

| **Noncompliant Code** |
| --- |
| The implicitly declared destructor is not declared as virtual in the presence of other virtual functions. |
| struct Base {  virtual void f();  };  struct Derived : Base {};  void f() {  Base \*b = new Derived();  // ...  delete b;  } |

| **Compliant Code** |
| --- |
| The destructor for the base is declared explicitly as a virtual destructor. This guarantees that the polymorphic delete  operation will have predictable behavior. |
| struct Base {  virtual ~Base() = default;  virtual void f();  };  struct Derived : Base {};  void f() {  Base \*b = new Derived();  // ...  delete b;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard. Using coding standards will avoid errors like not calling a virtual destructor when dealing with polymorphic types. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tools | 8.0 | 303 S | Partially implemented |
| Parasoft  C/C++test | 2023.2 | CERT\_CPP-OOP52-a | Define a virtual destructor in classes  used as base classes that have virtual functions |
| PRQA QA-C++ | 8.1 | 3402, 3403, 3404 | Do not use resources that have been  freed |
| Polyspace Bug  Finder | R2024a | CERT C++: OOP52-CPP | Checks for situations when a class has virtual functions but not a virtual  destructor |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integer Precision | [STD-010-CP  P] | An integer’s size is contributed by the padding bits, but this does not inherently  carry over to precision. This can lead to incorrect assumptions about the numeric range of these types. |

| **Noncompliant Code** |
| --- |
| If this code runs on a platform where unsigned int has one or more padding bits, it can result in a value for exp that is too large. |
| #include <limits.h>  unsigned int pow2(unsigned int exp) {  if (exp >= sizeof(unsigned int) \* CHAR\_BIT) {  /\* Handle error \*/  }  return 1 << exp;  } |

| **Compliant Code** |
| --- |
| Using popcount() function allows the code to determine the precision of any integer type, signed or unsigned. This  function will count the number of bits set on any unsigned integer. |
| #include <stddef.h>  #include <stdint.h>  /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num) {  size\_t precision = 0;  while (num != 0) {  if (num % 2 == 1) {  precision++;  }  num >>= 1;  }  return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):** Validate Input Data. Assumptions with input data can lead to errors. This principle ties in as the focus is  on using fewer assumptions with data handling and, in this case, integer precision. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | RuleChecker | Reports overflow due to insufficient precision |
| Helix QAC | 2023.2 | C0582  C++3115 | Implemented |
| Parasoft  C/C++test | 2023.2 | CERT\_C-INT35-a | Use correct integer precisions when  checking the right-hand operand of  the shift operator |
| Polyspace Bug  Finder | R2024a | CERT C: Rule INT35-C | Checks for situations when integer  precisions are exceeded |

### 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.

By integrating security measures into each step of the DevOps toolchain, DevOps becomes DevSecOps. Threat modeling is performed in the “Assess and Plan” phase. Integrated development environment security is addressed in the “Design” and “Build” phases. Static application testing and automated security scans are performed in the “Verify & Test” phase, along with unit, integration, and other tests.

Once in production, the automated testing continues with prevention by using integrity checks and defense-in-depth measures. Network monitoring, penetration testing, and performance logs are some methods supporting continuous threat detection. Like QA testing, security testing should be performed early and often.

### Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Likely | High | P9 | L2 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| 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 | Medium | P4 | L3 |
| STD-009-CPP | Low | Likely | Low | P9 | L2 |
| STD-010-CPP | Low | Unlikely | Medium | P2 | L3 |

### 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 protects stored data, which may include hard drives, phones, computers, and cloud assets. This data can be protected through encryption tools, disk encryption, and security for mobile devices and computers. |
| Encryption in flight | Encryption at flight is about protecting data that is moving. This data can be between two devices within a network or moving across the Internet. It can be protected using firewalls and vLANs. Based on current best practices, sensitive data moving across public infrastructure should leverage SSL or SSH encryption. System access should require authentication and authorization. Highly sensitive data should require multi-factor authentication. |
| Encryption in use | Encryption in use protects data that is created, edited, or otherwise defined as in-use. This data can be protected by ensuring data control and protection exists before use. Managing access rights and identity will also minimize risk to this data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the act of confirming one’s identity. This can cover a range of types, but often are examples such as static passwords, one-time passwords, certifications, and biometric credentials. These forms of identification work to ensure a person is who they claim to be. |
| Authorization | Authorization specifies a user's access rights and privileges and is an important part of information and computer security. Where authentication confirms an identity, authorization determines what a user can and cannot access within the system. Appropriate access rights reduce vulnerabilities by eliminating access to sensitive data the user does not need. |
| Accounting | Accounting is the process of tracking a user’s activity while interacting with a system. This logging may include timestamps, accessed resources, and data transfers. It is valuable in creating forensic “breadcrumbs” to analyze trends, exploits, or misused resources. |

**\***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 | 05/24/2024 | New Principles Added | Eric Farkas |  |
| 1.2 | 05/26/2024 | New Coding Standards Added | Eric Farkas |  |
| 1.3 | 06/10/2024 | Risk Assessments Added | Eric Farkas |  |
| 1.4 | 06/12/2024 | Encryption and Triple A Added | Eric Farkas |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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