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



Green Pace Secure Development Policy

Contents

[Overview 3](#_Toc68830069)

[Purpose 3](#_Toc68830070)

[Scope 3](#_Toc68830071)

[Module Three Milestone 3](#_Toc68830072)

[Ten Core Security Principles 3](#_Toc68830073)

[C/C++ Ten Coding Standards 4](#_Toc68830074)

[Coding Standard 1 5](#_Toc68830075)

[Coding Standard 2 7](#_Toc68830076)

[Coding Standard 3 10](#_Toc68830077)

[Coding Standard 4 12](#_Toc68830078)

[Coding Standard 5 14](#_Toc68830079)

[Coding Standard 6 17](#_Toc68830080)

[Coding Standard 7 19](#_Toc68830081)

[Coding Standard 8 21](#_Toc68830082)

[Coding Standard 9 23](#_Toc68830083)

[Coding Standard 10 25](#_Toc68830084)

[Defense-in-Depth Illustration 29](#_Toc68830085)

[Project One 29](#_Toc68830086)

[1. Revise the C/C++ Standards 29](#_Toc68830087)

[2. Risk Assessment 29](#_Toc68830088)

[3. Automated Detection 29](#_Toc68830089)

[4. Automation 29](#_Toc68830090)

[5. Summary of Risk Assessments 30](#_Toc68830091)

[6. Create Policies for Encryption and Triple A 30](#_Toc68830092)

[Figure 1. Data Phases 31](#_Toc68830093)

[Table 1. Secure Data-In-Transit Protocols 32](#_Toc68830094)

[7. Map the Principles 32](#_Toc68830095)

[Audit Controls and Management 34](#_Toc68830096)

[Enforcement 34](#_Toc68830097)

[Exceptions Process 34](#_Toc68830098)

[Distribution 35](#_Toc68830099)

[Policy Change Control 35](#_Toc68830100)

[Policy Version History 35](#_Toc68830101)

[Appendix A Lookups 35](#_Toc68830102)

[Approved C/C++ Language Acronyms 35](#_Toc68830103)

# 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 | All input shall be validated to prevent software vulnerabilities. All external data sources shall be considered unsafe including command line arguments, network interfaces, environmental variables and user-controlled files, for example. |
| 1. Heed Compiler Warnings | The highest compiler warning levels shall be used to compile code. All compiler warnings shall be addressed and eliminated by modifying code and using static and dynamic testing tools to detect and remove any additional security flaws. |
| 1. Architect and Design for Security Policies | Design and build software that enforces and heeds security policies. An example might be using private methods or classes. |
| 1. Keep It Simple | Code design shall be kept as small and simple as possible. The simpler the code, the less likelihood for errors and oversights. |
| 1. Default Deny | Access decisions shall be based on permission – not exclusion. Users, for example, shall not obtain access to a system without proper user credentials. |
| 1. Adhere to the Principle of Least Privilege | Each process shall execute with the least amount of privilege needed to perform that function. Least privilege pertains to users, programs, processes, networks, databases or entire systems. |
| 1. Sanitize Data Sent to Other Systems | Data shall be sanitized PRIOR to passing it to another system / subsystem. An example is removing SQL injection from input prior to passing it as an argument to another method. |
| 1. Practice Defense in Depth | Systems shall be protected in layers. Risk shall be managed with multiple strategies to prevent exploits of vulnerabilities. |
| 1. Use Effective Quality Assurance Techniques | Vulnerabilities can be identified and eliminated using good quality assurance techniques such as third-party security reviews. |
| 1. Adopt a Secure Coding Standard | A secure coding standard shall be developed for each individual programming language and platform as different languages come with different vulnerabilities. |

Source: <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

## 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-CLG] | Declare all variables, functions and return types prior to using them. Do not accept implicit declarations. |

Source: <https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them>

| **Noncompliant Code** |
| --- |
| Unspecified (implicit) variable type |
| foo = 0; |

| **Compliant Code** |
| --- |
| Specified variable type |
| int foo = 0; |

| **Noncompliant Code** |
| --- |
| Unspecified (implicit) function type |
| #include <limits.h>  #include <stdio.h>    foo(void) {  return UINT\_MAX;  }    int main(void) {  long long int c = foo();  printf("%lld\n", c);  return 0;  } |

| **Compliant Code** |
| --- |
| Specified function type |
| #include <limits.h>  #include <stdio.h>    unsigned int foo(void) {  return UINT\_MAX;  }    int main(void) {  long long int c = foo();  printf("%lld\n", c);  return 0;  } |

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

|  |
| --- |
| **Principles(s):**  #1: Input must match defined data types.  #2: Promptly correct all compiler warnings relating to data type declarations.  #4: Explicit variable declarations are easier to understand and track than implicit declarations.  #8: Variable, function and return declarations add a layer of defense |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | UNLIKELY | LOW | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Type-specifier  Function-return-type  Implicit-function-declaration  Undeclared-parameter | Fully checked |
| ECLAIR | 1.2 | [CERTC-DCL31](https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them) | Fully implemented |
| Parasoft C/C++test | 2020.2 | [CERTC-DCL31](https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them) | All functions shall be declared before use. |
| Polyspace Bug Finder | R2020a | [CERTC-DCL31](https://wiki.sei.cmu.edu/confluence/display/c/DCL31-C.+Declare+identifiers+before+using+them) | Types not explicitly specified;  Implicit function declaration. |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Do not read uninitialized variables. |

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory>

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

| **Compliant Code** |
| --- |
| Local variable is initialized prior to printing. |
| #include <iostream>    void f() {  **int** i = 0;    std::cout << i;  } |

| **Noncompliant Code** |
| --- |
| Uninitialized memory evaluated as part of print command |
| #include <iostream>    void f() {  int \*i = new int;  std::cout << i << ", " << \*i;  } |

| **Compliant Code** |
| --- |
| Memory initialized to value of 12 prior to printing its value. |
| #include <iostream>    void f() {  **int** \*i = new **int**(12);    std::cout << i << ", " << \*i;  } |

| **Noncompliant Code** |
| --- |
| Class member c is not initialized in the default constructor. |
| class S {  int c;    public:  int f(int i) const { return i + c; }  };    void f() {  S s;  int i = s.f(10);  } |

| **Compliant Code** |
| --- |
| Class S is given a default constructor that initializes class member c. |
| class S {  int c;    public:  S() : c(0) {}  int f(int i) const { return i + c; }  };    void f() {  S s;  int i = s.f(10);  } |

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

|  |
| --- |
| **Principles(s):**  #1: Input shall be within ranges defined by variable initialization.  #2: Promptly correct all compiler warnings related to uninitialized variables.  #8: Initializing all variables adds a layer of defense |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | MEDIUM | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Uninitialized-read | Partially checked |
| Parasoft C/C++test | 2020.2 | [EXP53-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory) | Avoid use before initialization |
| Polyspace Bug Finder | R2020a | [EXP53-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP53-CPP.+Do+not+read+uninitialized+memory) | Non-initialized variable;  Non-initialized pointer; |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Guarantee that string storage has enough space for null terminator. |

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator>

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, buffer overflow can occur. |
| #include <iostream>    void f() {  char buf[12];  std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| By using std::string, buffer overflow and data truncation is prevented. |
| #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):**  #1: Storage for input string must allow for null terminator space.  #7: String lengths bounded by null terminator prevent occurrences of SQL injection and buffer overruns.  #8: String length verification provides a layer of defense against vulnerabilities. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | [STR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | No space for null terminator;  Buffer overrun;  Type overrun. |
| Parasoft C/C++test | 2020.2 | [STR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | * 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' |
| Polyspace Bug Finder | R2020a | [STR50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation   Rule partially covered. |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CLG] | Always use prepared statements for querying. |

| **Noncompliant Code** |
| --- |
| Without a prepared statement, the user may alter the SQL statement directly. |
| uName = getRequestString("username");  uPass = getRequestString("userpassword");  sql = “SELECT \* FROM Users WHERE Name = " + uName + " AND Pass = " +  uPass + ” |

| **Compliant Code** |
| --- |
| With a prepared statement, the user does not have control of the SQL statement, thus limiting the potential for injection. |
| PreparedStatement pStmt = PreparedStatement();  std::cin >> username;  std::cin >> userpassword;  sql = “SELECT \* FROM Users WHERE Name = %s AND Pass = %s;”, username,  userpassword}; |

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

|  |
| --- |
| **Principles(s):**  #1: Validate input to prevent SQL injection.  #3: Use prepared SQL statements to prevent injection.  #7: Usage of prepared statements allow data to be sanitized as they reduce likelihood / add a layer of defense against SQL injection attacks. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | [FIO30-C](https://wiki.sei.cmu.edu/confluence/display/c/FIO30-C.+Exclude+user+input+from+format+stringsFIO30-C) | Never use unfiltered data from an untrusted user as the format parameter; |
| Astre’e | 20.10 | - | Supported via stubbing / taint analysis |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory. |

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory>

| **Noncompliant Code** |
| --- |
| Here, s is dereferenced after it has been deleted resulting in a write-after-free vulnerability. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| Memory storage s is not deallocated until after it is no longer needed. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Noncompliant Code** |
| --- |
| A pointer is pointing to released memory once the std::string object is destroyed at the end of the expression, resulting in undefined behavior. |
| #include <string>    std::string str\_func();  void display\_string(const char \*);    void f() {  const char \*str = str\_func().c\_str();  display\_string(str); /\* Undefined behavior \*/  } |

| **Compliant Code** |
| --- |
| A local copy of string returned by the std::string object is made to ensure that the string str will be valid when the call to display\_string() is made. |
| #include <string>    std::string str\_func();  void display\_string(const **char** \*s);    void f() {    std::string str = str\_func();    const **char** \*cstr = str.c\_str();    display\_string(cstr);  /\* ok \*/  } |

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

|  |
| --- |
| **Principles(s):**  #2: Use static tools to detect, identify, and mitigate freed memory issues.  #9: Use effective QA techniques to identify freed memory occurrences that could be easily overlooked if testing is bypassed. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | MEDIUM | P18 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Dangling\_pointer\_use | - |
| Parasoft C/C++test | 2020.2 | [MEM50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) | Do not use resources that have been freed |
| Polyspace Bug Finder | R2020a | [MEM50-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory) | * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Use assertions to test assumptions such as if a pointer is NULL or not. If the expression evaluates to false, the abort() function is called preventing unexpected behavior. |

| **Noncompliant Code** |
| --- |
| Here the pointer may be NULL which can result in program exploitation. |
| /\* assert example \*/  #include <stdio.h> /\* printf \*/  #include <assert.h> /\* assert \*/  void print\_number(int\* myInt) {  printf ("%d\n",\*myInt);  }  int main ()  {  int a = 10;  int \* b = NULL;  int \* c = NULL;  b=&a;  print\_number (b);  print\_number (c);  return 0;  } |

| **Compliant Code** |
| --- |
| The assert() function is used added to prevent printing from dereferenced memory. |
| /\* assert example \*/  #include <stdio.h> /\* printf \*/  #include <assert.h> /\* assert \*/  void print\_number(int\* myInt) {  assert (myInt!=NULL);  printf ("%d\n",\*myInt);  }  int main ()  {  int a = 10;  int \* b = NULL;  int \* c = NULL;  b=&a;  print\_number (b);  print\_number (c);  return 0;  } |

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

|  |
| --- |
| **Principles(s):**  #9: Use assertions to test / check code throughout the program. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | [ERR56-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR56-CPP.+Guarantee+exception+safety) | Partially implemented |

### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| There is no handler in either f() nor main() for the exception thrown in throwing\_func(), thus std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| Here there is a try/catch block in main() to handle exceptions. |
| 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):**  #3: Design code using try/catch/throw to prevent code from stopping abruptly.  #9: Continually test throughout code and throw exceptions to prevent code from behaving unexpectedly. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| LOW | PROBABLE | MEDIUM | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Catch\_all  Catch\_early | Partially checked |
| Parasoft C/C++test | 2020.2 | [ERR51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) | 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 | R2020a | [ERR51-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions) | Checks for unhandled exceptions (partially covered) |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input / Output | [STD-008-CPP] | Close files when they are no longer needed. |

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>

| **Noncompliant Code** |
| --- |
| The constructor calls open(), however close() is not called, resulting in the default std::terminate\_handler to be abort(), which does not call destructors and thus the file is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| Here, file.close() is called, ensuring that file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

|  |
| --- |
| **Principles(s):**  #8: Closing open files provides a layer of defense by restricting unpermitted data access.  #10: Closing open files is best practice. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| MEDIUM | UNLIKELY | MEDIUM | P4 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Klocwork | 2018 | RH.LEAK | - |
| Parasoft C/C++test | 2020.2 | [FIO51a-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed) | Ensure resources are freed |
| Polyspace Bug Finder | R2020a | [FIO51a-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed) | Checks for resource leaks (rule partially covered) |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Use valid iterator ranges. |

Source: <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges>

| **Noncompliant Code** |
| --- |
| The incorrect values are passed to the for\_each() function. As long as the two iterators are not equal, c.end will keep advancing resulting in a past-the-end element which leads to undefined behavior. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| Here the two iterator values are passed correctly to the for\_each() function. |
| #include <algorithm>  #include <iostream>  #include <vector>    void f(const std::vector<int> &c) {  std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

|  |
| --- |
| **Principles(s):**  #3: Protect iterators from generating overflow errors.  #4: Explicitly define iterator ranges so code is simpler to understand.  #8: Preventing iterator overflow is a layer of defense and best practice. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | PROBABLE | HIGH | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Overflow\_upon\_dereference | - |
| Parasoft C/C++test | 2020.2 | [CTR53a/b-CPP](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR53-CPP.+Use+valid+iterator+ranges) | Do not use an iterator range that isn't really a range Do not compare iterators from different containers |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Integers | [STD-010-CLG] | Ensure that operations on signed integers do not result in overflow. |

Source: <https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>

| **Noncompliant Code** |
| --- |
| This addition example can result in overflow. |
| void func(signed int si\_a, signed int si\_b) {  signed int sum = si\_a + si\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This addition example tests for overflow prior to calculating the sum. |
| #include <limits.h>    void f(signed int si\_a, signed int si\_b) {  signed int sum;  if (((si\_b > 0) && (si\_a > (INT\_MAX - si\_b))) ||  ((si\_b < 0) && (si\_a < (INT\_MIN - si\_b)))) {  /\* Handle error \*/  } else {  sum = si\_a + si\_b;  }  /\* ... \*/  } |

| **Noncompliant Code** |
| --- |
| This subtraction example can result in overflow / underflow. |
| void func(signed int si\_a, signed int si\_b) {  signed int diff = si\_a - si\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This subtraction example guarantees no possibility of overflow. |
| #include <limits.h>    void func(signed int si\_a, signed int si\_b) {  signed int diff;  if ((si\_b > 0 && si\_a < INT\_MIN + si\_b) ||  (si\_b < 0 && si\_a > INT\_MAX + si\_b)) {  /\* Handle error \*/  } else {  diff = si\_a - si\_b;  }    /\* ... \*/  } |

| **Noncompliant Code** |
| --- |
| This multiplication example can result in overflow. |
| void func(signed int si\_a, signed int si\_b) {  signed int result = si\_a \* si\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This multiplication example prevents an overflow result. |
| #include <stddef.h>  #include <assert.h>  #include <limits.h>  #include <inttypes.h>    extern size\_t popcount(uintmax\_t);  #define PRECISION(umax\_value) popcount(umax\_value)    void func(signed int si\_a, signed int si\_b) {  signed int result;  signed long long tmp;  assert(PRECISION(ULLONG\_MAX) >= 2 \* PRECISION(UINT\_MAX));  tmp = (signed long long)si\_a \* (signed long long)si\_b;    /\*  \* If the product cannot be represented as a 32-bit integer,  \* handle as an error condition.  \*/  if ((tmp > INT\_MAX) || (tmp < INT\_MIN)) {  /\* Handle error \*/  } else {  result = (int)tmp;  }  /\* ... \*/  } |

| **Noncompliant Code** |
| --- |
| This division example can result in overflow. |
| void func(signed long s\_a, signed long s\_b) {  signed long result;  if (s\_b == 0) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This division example eliminates the possibility of overflow. |
| #include <limits.h>    void func(signed long s\_a, signed long s\_b) {  signed long result;  if ((s\_b == 0) || ((s\_a == LONG\_MIN) && (s\_b == -1))) {  /\* Handle error \*/  } else {  result = s\_a / s\_b;  }  /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):**  #1: Validate input that may contribute/cause overflow.  #9: Always surround computations with tests for integer overflow and handle exceptions. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| HIGH | LIKELY | HIGH | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astre’e | 20.10 | Integer-overflow | Fully checked |
| Parasoft C/C++test | 2020.2 | [INT32a/b/c-C](https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow) | Avoid integer overflows Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2020a | [INT32a/b/c-C](https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow) | * Integer overflow * Tainted division operand * Tainted modulo operand |

## 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 transforms to DevSecOps by integrating security measures into each step of the DevOps toolchain. Within the pre-production phases, threat modeling and security tool training and selections are added to the “Assess and Plan” segment. In the “Design” and “Build” phases, IDE security is addressed. Static application testing and automated security scans are added to 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, network monitoring and performance logs are some methods of continuous threat detection. Just as with testing in the QA sense, testing in the security sense should also 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-CLG | LOW | UNLIKELY | LOW | 3 | 3 |
| STD-002-CPP | HIGH | PROBABLE | MEDIUM | 12 | 1 |
| STD-003-CPP | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-004-CLG | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-005-CPP | HIGH | LIKELY | MEDIUM | 18 | 1 |
| STD-006-CPP | HIGH | LIKELY | HIGH | 9 | 2 |
| STD-007-CPP | LOW | PROBABLE | MEDIUM | 4 | 3 |
| STD-008-CPP | MEDIUM | UNLIKELY | MEDIUM | 4 | 3 |
| STD-009-CPP | HIGH | PROBABLE | HIGH | 6 | 2 |
| STD-010-CLG | HIGH | LIKELY | HIGH | 9 | 2 |

## Create Policies for Encryption and Triple A

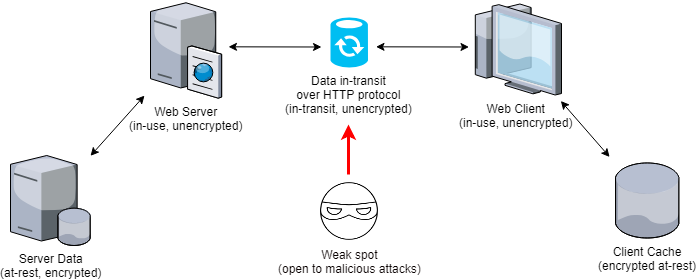
Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Protects data where it is stored (eg: on a computer/phone, a database, in cloud). This may include physical sources such as hard drives, or logical sources such as databases or cloud assets. There are several options for encryption tools, such as VeraCrypt, AxCrypt or InnoDB tablespace encryption. These tools are necessary to protect data from being physically stolen, logically stolen, or otherwise be breached. |
| Encryption at flight | Protects data as it is moved from one location to another (eg: sending an email, browsing the internet). Email encryption tools such as S/MIME or PGP shall be used for email transmission. Web traffic shall only be sent over a Secure Sockets Layer (SSL) such as Transport Layer Security (TLS) by obtaining a SSL/TLS HTTPS certificate from authorities such as GoDaddy or DigiCert. Refer to Table 1 below for suggested secure network protocols. Using these tools prevents network layer attacks, such as eavesdropping, and tampering-based attacks, such as third-party communication hijacking. |
| Encryption in use | Protects data as it is being created, edited, accessed, processed, or viewed. This state occurs in between the at-rest and at-flight states when, say for instance, accessing a website on a server, or whenever the CPU is in use processing applications. It is important to encrypt data-in-use because memory can be hacked, and encryption keys for data-at-rest may be exposed. CPU manufacturer AMD offers full memory encryption, called Secure Memory Encryption (SME), and Intel offers Total Memory Encryption (TME) in order to protect CPU based key storage. There are also cryptographic tools that can be used to protect data during computation. |

Source: <https://www.ryadel.com/en/data-encryption-in-transit-at-rest-definitions-best-practices-tutorial-guide/>



Source: <https://www.ryadel.com/en/data-encryption-in-transit-at-rest-definitions-best-practices-tutorial-guide/>

### Figure 1. Data Phases

### Table 1. Secure Data-In-Transit Protocols

|  |  |  |
| --- | --- | --- |
| Transfer Type | What to avoid (insecure) | What to use (secure) |
| Web Access | HTTP | HTTPS |
| E-Mail Servers | POP3, SMTP, IMAP | POP3S, IMAPS, SMTPS |
| File Transfer | FTP, RCP | FTPS, SFTP, SCP, WebDAV over HTTPS |
| Remote Shell | telnet | SSH2 |
| Remote Desktop | VNC | radmin, RDP |

Source: <https://www.ryadel.com/en/data-encryption-in-transit-at-rest-definitions-best-practices-tutorial-guide/>

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Who are you? Authentication verifies a user’s identity credentials. This can be achieved in several ways, including username/password verification, single sign-on (SSO) systems, biometrics, I and/or digital certificates, for example. All users, new and existing, shall be verified. Identity theft and unauthorized system access can result if user credentials are not authenticated. |
| Authorization | What can you use? Authorization is set for each existing and new user and defines the level of access to files, directories and/or applications for each individual user. Each user or group shall be permitted to have read, write and/or execution permission, depending on the user’s role within an organization. For example, a data entry clerk may have read / write access to view or edit a particular file while a manager may warrant administrative permissions. |
| Accounting | What happened and when? Accounting refers to records, or log files, which detail things like user logins; new user profile creations; file access events; database updates; data transfers or access. All actions are date and timestamped as a record of occurrence. Usage information can track events such as authorization or resource utilization which can be used in turn for system wide planning. Accounting tracks who is doing what at all times. |

Source: <https://codebots.com/application-security/aaa-security-an-introduction-to-authentication-authorisation-accounting>

**\***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 | 03/20/2021 | Draft | Marta Taylor |  |
| 2.0 | 04/08/2021 | Amendment | Marta Taylor |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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