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



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc69073463)

[Purpose 2](#_Toc69073464)

[Scope 2](#_Toc69073465)

[Module Three Milestone 2](#_Toc69073466)

[Ten Core Security Principles 2](#_Toc69073467)

[C/C++ Ten Coding Standards 3](#_Toc69073468)

[Coding Standard 1 4](#_Toc69073469)

[Coding Standard 2 6](#_Toc69073470)

[Coding Standard 3 8](#_Toc69073471)

[Coding Standard 4 10](#_Toc69073472)

[Coding Standard 5 12](#_Toc69073473)

[Coding Standard 6 14](#_Toc69073474)

[Coding Standard 7 16](#_Toc69073475)

[Coding Standard 8 18](#_Toc69073476)

[Coding Standard 9 20](#_Toc69073477)

[Coding Standard 10 22](#_Toc69073478)

[Defense-in-Depth Illustration 24](#_Toc69073479)

[Project One 24](#_Toc69073480)

[1. Revise the C/C++ Standards 24](#_Toc69073481)

[2. Risk Assessment 24](#_Toc69073482)

[3. Automated Detection 24](#_Toc69073483)

[4. Automation 24](#_Toc69073484)

[5. Summary of Risk Assessments 25](#_Toc69073485)

[6. Create Policies for Encryption and Triple A 26](#_Toc69073486)

[7. Map the Principles 27](#_Toc69073487)

[Audit Controls and Management 28](#_Toc69073488)

[Enforcement 28](#_Toc69073489)

[Exceptions Process 28](#_Toc69073490)

[Distribution 29](#_Toc69073491)

[Policy Change Control 29](#_Toc69073492)

[Policy Version History 29](#_Toc69073493)

[Appendix A Lookups 29](#_Toc69073494)

[Approved C/C++ Language Acronyms 29](#_Toc69073495)

[Resources 29](#_Toc69073496)

# 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 | The validate input data principle states to validate input from all untrusted data sources. It also states to be suspicious of most external data sources and this includes network interfaces, command line arguments, and user controlled files. |
| 1. Heed Compiler Warnings | The heed compiler warnings principle states to compile your code using the highest warning level available on your compiler and to work to eliminate these warnings by modifying the code. |
| 1. Architect and Design for Security Policies | The architect and design for policies principle states to create a software architecture that is geared towards enforcing security policies. This also includes designing your software to implement these security policies. |
| 1. Keep It Simple | The keep it simple principle is as it says, to keep the design of your system as small and simple as possible. |
| 1. Default Deny | The default deny principle states to essentially default your system to deny access. It also mentions to base access decisions on permission rather than on exclusion. |
| 1. Adhere to the Principle of Least Privilege | The adhere to the principle of least privilege principle states to write your code where every process should be able to execute with the least set of privileges necessary in order to complete the job. It also states that any elevated privilege should only be accessed for the least amount of time required to complete a task. |
| 1. Sanitize Data Sent to Other Systems | The sanitize data sent to other systems principle is also as it states. It mentions to sanitize the data being sent to complex subsystems. |
| 1. Practice Defense in Depth | The practice defense in depth principle states to create layers of security so that if one layer is broken another is in its place to prevent further attack. |
| 1. Use Effective Quality Assurance Techniques | The use effective quality assurance techniques principle states to include various forms of quality assurance techniques such as fuzz testing, penetration testing, and source code audits in order to identify and eliminate vulnerabilities. |
| 1. Adopt a Secure Coding Standard | The adopt a secure coding standard principle states to develop and/or apply a secure coding standard for your target development language and platform. |

## 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** | DCL12-C | Implement abstract data types using opaque types |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is based on the managed string library developed by CERT [Burch 2006]. In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file as follows: |
| struct string\_mx {  size\_t size;  size\_t maxsize;  unsigned char strtype;  char \*cstr;  };    typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ |

| **Compliant Code** |
| --- |
| This compliant solution reimplements the string\_mx type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two header files: an external string\_m.h header file that is included by the user of the data type and an internal file that is included only in files that implement the managed string abstract data type.  In the external string\_m.h file, the string\_mx type is defined to be an instance of struct string\_mx, which in turn is declared as an incomplete type:  In the internal header file, struct string\_mx is fully defined but not visible to a user of the data abstraction: |
| struct string\_mx;  typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/  struct string\_mx {  size\_t size;  size\_t maxsize;  unsigned char strtype;  char \*cstr;  }; |

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

|  |
| --- |
| **Principles(s):** 3, 10 This coding standard maps to the core principles 3 and 10. The reason for this is that this standard mentions to use abstract data types instead of opaque ones. This matches the third principle of architecting and designing for security policies and the tenth principle of adopting a secure coding standard. This standard should be used throughout the design process in order to prevent unnecessary security issues due to incorrect data types. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | CERTC-DCL12 |  |
| LDRA tool Suite | 9.7.1 | 104 D | Partially implemented |
| Polyspace Bug Finder | R2020a | CERT C: Rec. DCL12-C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| Parasoft C/C++ test | 2020.2 | CERT\_C-DCL12-a | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | INT31-C | Ensure that integer conversions do not result in lost or misinterpreted data |

| **Noncompliant Code** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most implementations: |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type: |
| #include <limits.h>    void func(void) {  unsigned long int u\_a = ULONG\_MAX;  signed char sc;  if (u\_a <= SCHAR\_MAX) {  sc = (signed char)u\_a; /\* Cast eliminates warning \*/  } else {  /\* Handle error \*/  }  } |

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

|  |
| --- |
| **Principles(s):** 9. I feel that this standard best reflects security principle number nine. The reason for this is that principle nine focuses on using effective quality assurance techniques. This standard is in place to prevent the unnecessary lose or misinterpretation of data due to integer conversions. The standard also shows how to validate ranges in order to prevent the damage of data. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| LDRA tool suit | 9.7.1 | 93 S, 433 S, 434 S | Partially implemented |
| Polyspace Bug Finder | R2020a | CERT C: Rule INT31-C | Checks for:   * Integer conversion overflow * Call to memset with unintended value * Sign change integer conversion overflow * Tainted sign change conversion * Unsigned integer conversion overflow   Rule partially overed. |
| TrustInSoft Analyzer | 1.38 | Signed\_downcast | Exhaustively verified |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | EXP58-CPP | Pass an object of the correct type to va\_start |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the object passed to va\_start() will undergo a default argument promotion, which results in undefined behavior. |
| #include <cstdarg>    extern "C" void f(float a, ...) {  va\_list list;  va\_start(list, a);  // ...  va\_end(list);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() accepts a double instead of a float. |
| #include <cstdarg>    extern "C" void f(double a, ...) {  va\_list list;  va\_start(list, a);  // ...  va\_end(list);  } |

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

|  |
| --- |
| **Principles(s):** 4. This standard I had a hard time mapping as it didn’t quite fit many of the core principles. However, principle four seemed to be the best principle for this standard as it focuses on keeping things simple. This standard mentions to use the correct type to pass to va\_start. To do that is as simple as it gets and changing the data type to double. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wvarargs | Does not catch the violation in the third noncompliant code example (it is conditionally supported by Clang) |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-EXP58-a | Use macros for variable arguments correctly |
| Polyspace Bug Finder | R2020a | CERT C++: EXP58-CPP | Checks for incorrect data types for second argument of va\_start (rule fully covered) |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STR02-C | Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system() [Viega 2003]: |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, shows the whitelisting approach: |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {  \*cp = '\_';  } |

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

|  |
| --- |
| **Principles(s):** 7. This standard is a near identical match to principle 7, sanitize data sent to other systems. This standard focuses on how to properly sanitize data and provides code examples of how to do this. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Polyspace Bug Finder | R2020a | CERT C: Rec. STR02-C | Checks for:   * Execution of externally controlled command * Command executed from externally controlled path * Library loaded from externally controlled path   Rec. partially covered. |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | CON52-CPP | Prevent data races when accessing bit-fields from multiple threads |

| **Noncompliant Code** |
| --- |
| Adjacent bit-fields may be stored in a single memory location. Consequently, modifying adjacent bit-fields in different threads is undefined behavior, as shown in this noncompliant code example. |
| struct MultiThreadedFlags {  unsigned int flag1 : 2;  unsigned int flag2 : 2;  };    MultiThreadedFlags flags;    void thread1() {  flags.flag1 = 1;  }    void thread2() {  flags.flag2 = 2;  } |

| **Compliant Code** |
| --- |
| This compliant solution protects all accesses of the flags with a mutex, thereby preventing any data races. |
| #include <mutex>    struct MultiThreadedFlags {  unsigned int flag1 : 2;  unsigned int flag2 : 2;  };    struct MtfMutex {  MultiThreadedFlags s;  std::mutex mutex;  };    MtfMutex flags;    void thread1() {  std::lock\_guard<std::mutex> lk(flags.mutex);  flags.s.flag1 = 1;  }    void thread2() {  std::lock\_guard<std::mutex> lk(flags.mutex);  flags.s.flag2 = 2;  } |

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

|  |
| --- |
| **Principles(s):** 10. This standard is more similar to the tenth principle of adopting a secure coding standard. This standard discusses how to prevent data races when accessing bit fields. When it comes to developing code to prevent security issues this standard provides example code and brief descriptions to guide a user. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Read\_write\_data\_race  Write\_write\_data\_race | Supported |
| Axivion Bauhaus Suite | 6.9.0 | CERTC++-CON52 |  |
| Coverity | 6.5 | RACE\_CONDITION | Fully implemented |
| PRQA QA-C++ | 4.4 | 1774, 1775 | Enforced by MTA |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | CTR54-CPP | Do not subtract iterators that do not refer to the same container |

| **Noncompliant Code** |
| --- |
| This noncompliant code example attempts to determine whether the pointer test is within the range [r, r + n]. However, when test does not point within the given range, as in this example, the subtraction produces undefined behavior. |
| #include <cstddef>  #include <iostream>    template <typename Ty>  bool in\_range(const Ty \*test, const Ty \*r, size\_t n) {  return 0 < (test - r) && (test - r) < (std::ptrdiff\_t)n;  }    void f() {  double foo[10];  double \*x = &foo[0];  double bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates a fully portable, but likely inefficient, implementation of in\_range() that compares test against each possible address in the range [r, n]. A compliant solution that is both efficient and fully portable is currently unknown. |
| #include <iostream>    template <typename Ty>  bool in\_range(const Ty \*test, const Ty \*r, size\_t n) {  auto \*cur = reinterpret\_cast<const unsigned char \*>(r);  auto \*end = reinterpret\_cast<const unsigned char \*>(r + n);  auto \*testPtr = reinterpret\_cast<const unsigned char \*>(test);    for (; cur != end; ++cur) {  if (cur == testPtr) {  return true;  }  }  return false;  }    void f() {  double foo[10];  double \*x = &foo[0];  double bar;  std::cout << std::boolalpha << in\_range(&bar, x, 10);  } |

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

|  |
| --- |
| **Principles(s):** 3. I feel that this standard best reflects principle three. The reason for this is in how the standard points out how to structure the code. This standard also discusses how an improper structure can lead to issues and unknown behavior. Unknown behavior can have all kinds of effects. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | P8 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Invalid\_pointer\_subtraction  Invalid\_pointer\_comparison |  |
| LDRA tool suite | 9.7.1 | 70 S, 87 S, 437 S, 438 S | Enhanced Enforcement |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-CTR54-a  CERT\_CPP-CTR54-b  CERT\_CPP-CTR54-c | Do not compare iterators from different containers  Do not compare two unrelated pointers  Do not subtract two pointers that do not address elements of the same array |
| PRQA QA-C++ | 4.4 | 2668, 2761, 2762, 2763, 2766, 2767, 2768 | Enforced by QA-CPP |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | Honor exception specifications |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) noexcept(true) {  v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions. |
| #include <cstddef>  #include <vector>    void f(std::vector<int> &v, size\_t s) {  v.resize(s); // May throw, but that is okay  } |

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

|  |
| --- |
| **Principles(s):** 2. This standard is similar to the second principle, heed compiler warnings. It discusses being mindful of exceptions and actually points out that it may be best to design code that will throw an exception when you least expect to see one. This can lead to much safer code and the standard does a great job of providing a noncompliant code example that points out the consequences well. Some functions may be flawed that will not display an exception and in those cases it is best to plan ahead and ask the code to verify just in case. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Unhandled-throw-noexcept | Partially checked |
| LDRA tool suite | 9.7.1 | 56 D | Partially implemented |
| Parasoft C/C++ Test | 2020.2 | CERT\_CPP-ER55-a | Where a function’s declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |
| RuleChecker | 20.10 | Unhandled-throw-noexcept | Partially checked |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Memory Management | MEM56-CPP | Do not store an already-owned value in an unrelated smart pointer |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction. When the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. This compliant solution also calls std::make\_shared() instead of allocating a raw pointer and storing its value in a local variable. |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

|  |
| --- |
| **Principles(s):** 3,4, 10. This standard matches to both principle 3 and 10. This standard is also a great example of structuring code to prevent issues. This standard may also be a great example of principle 4 in that it shows ways of simplifying this process of how to properly remove pointers without adding large amounts of code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Dangling\_pointer\_use |  |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-MEM56-a | Do not store an already-owned pointer value in an unrelated smart pointer |
| PVS-Studio | 7.07 | V1006 |  |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Characters and Strings | STR50-CPP | Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the unformatted input function std::basic\_istream<T>::read() is used to read an unformatted character array of 32 characters from the given file. However, the read() function does not guarantee that the string will be null terminated, so the subsequent call of the std::string constructor results in undefined behavior if the character array does not contain a null terminator. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  char buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }    std::string str(buffer);  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution assumes that the input from the file is at most 32 characters. Instead of inserting a null terminator, it constructs the std::string object based on the number of characters read from the input stream. If the size of the input is uncertain, it is better to use std::basic\_istream<T>::readsome() or a formatted input function, depending on need. |
| #include <fstream>  #include <string>    void f(std::istream &in) {  char buffer[32];  try {  in.read(buffer, sizeof(buffer));  } catch (std::ios\_base::failure &e) {  // Handle error  }  std::string str(buffer, in.gcount());  // ...  } |

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

|  |
| --- |
| **Principles(s):** 3, 9. This standard matches to the third principle. This standard discusses how to guarantee the storage for strings so they have enough space for characters and the null pointer. This falls in the realm of architecting and designing code for security policies. Now that I am thinking about it, this standard also falls in the realm of principle 9. The reason for this is that the standard points out that it is necessary that there is enough space for strings and that applies to quality assurance techniques. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| codeSonar | 6.0p0 | MISC.MEM.NTERM  LANG.MEM.BO  LANG.MEM.TO | No space for null terminator  Buffer overrun  Type overrun |
| Klocwork | 2018 | NNTS.MIGHT  NNTS.TAINTED |  |
| LDRA tool suite | 9.7.1 | 489 S, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Avoid overflow due to reading a not zero terminated string  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Do not use the ‘char’ buffer to store input from ‘std::cin’ |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | FIO51-CPP | Close files when they are no longer needed |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object 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** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the 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):** 8,10. This standard is a great example of principles eight and ten. I want to include principle eight in this standard because this is a form of defense in depth. It is small but if files are not closed then then access remains open. This standard is also a great example of adopting a secure coding standard as every project, if it is accessing files, must have a close operation. |

**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 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Polyspace Bug Finder | R2020a | CERT C++: FIO51-CPP | Checks for resource leak (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.

In order to implement this process, I am going to be using the above infographic to explain the process of DevSecOps. As with the usual process of DevOps, everything starts with the assessment and planning phase. What is different about this stage is the inclusion of security into the planning process. From there the process moves into the design phase where security driven design is implemented. Following that is the build phase where measures are taken to use secure open-source materials and trusted repositories. After this is the verification and testing phase. This is where the project up to this point will be vulnerability scanned and security tested before moving into production.

Once in production the system will be monitored through transition and health checks to makes sure the move was smooth and successful. After this, monitoring measures such as logging, event alerting, and analytics will be used to keep an eye on the system. Last of the cycles for Production involve responding to attacks if there are any and maintaining and stabilizing the system if an event occurs.

## 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 | Unlikely | Medium | High | 2 |
| DCL12-C | Low | Unlikely | High | P1 | 3 |
| INT31-C | High | Probable | High | P6 | 2 |
| EXP58-CPP | Medium | Unlikely | Medium | P4 | 3 |
| STR02-C | High | Likely | Medium | P18 | 1 |
| CON52-CPP | Medium | Probable | Medium | P8 | 2 |
| CTR54-CPP | Medium | Probable | Medium | P8 | 2 |
| ERR55-CPP | Low | Likely | Low | P9 | 2 |
| MEM56-CPP | High | Likely | Medium | P18 | 1 |
| STR50-CPP | High | Likely | Medium | P18 | 1 |
| FIO51-CPP | Medium | Unlikely | Medium | P4 | 3 |

## Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption in rest is encryption of data when it is on the disk. This means that when data is about to be used it is currently encrypted. This policy applies at its focus is to prevent security leaks due to data being exposed on the disk. |
| Encryption at flight | Encryption at flight or in flight is when data is encrypted while in use or in transit. This is something similar to encrypting messages between a server and client. This policy applies to prevent security and data leaks when data is being transmitted from one point to another. |
| Encryption in use | Encryption in use is when data is encrypted in the memory. This you would probably see most often on a database or other memory storage. The policy applies to prevent data and security leaks to stored data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the focus of verifying the identity of a call and return from server and client or other communication event. The policy applies in that it is meant to provide security around system calls that are made and no call should be made that cannot be verified to match the key or other pass into the system. Authentication can also be used for encrypting and decrypting messages. Looking at the list down below authentication can be used for a number of the tasks including user logins, changes to the database, addition of new users, and files accessed by the users. |
| Authorization | Authorization is the focus of providing a limit to the access a user has to the system. The policy applies by limiting what a user is able to do and only requiring higher level access for tasks that can be done briefly. The more time a user has access to higher level locations the greater there is a chance of the user’s access being used to breach the system. Authorization also adds a layer of security. |
| Accounting | Accounting is the focus of taking note when something has occurred and taking measures to prevent further attack. The policy applies to things like routine security checks and other security audits. |

**\***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 | 03/21/2021 | Filled In the security principles | Nathanael Burke | [Insert text.] |
| 1.2 | 04/11/2021 | Completed the security policy and added additional resources. | Nathanael Burke | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

# Resources

Ballman, A. & modified by Gangopadhyay, A. (2019, October) EXP58-CPP. Pass an object of the correct type to va\_start. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP58-CPP.+Pass+an+object+of+the+correct+type+to+va_start>

Ballman, A. & modified by Herter, J. (2020, October) MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM56-CPP.+Do+not+store+an+already-owned+pointer+value+in+an+unrelated+smart+pointer>

Keaton, D. & modified by Herter, J. (2020, October) CON52-CPP. Prevent data races when accessing bit-fields from multiple threads. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON52-CPP.+Prevent+data+races+when+accessing+bit-fields+from+multiple+threads>

Pincar, J. & modified by Gangopadhyay, A. (2019, October) FIO51-CPP. Close files when they are no longer needed. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO51-CPP.+Close+files+when+they+are+no+longer+needed>

Pincar, J. & modified by Rozenau, M. (2021, February) CTR54-CPP. Do not subtract iterators that do not refer to the same container. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CTR54-CPP.+Do+not+subtract+iterators+that+do+not+refer+to+the+same+container>

Pincar, J. & modified by Rozenau, M. (2021, February) STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator. *Carnegie Mellon University.* Retrieved from <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>

Seacord, R & modified by Gangopadhyay, A. (2019, August) DCL12-C. Implement abstract data types using opaque types. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/c/DCL12-C.+Implement+abstract+data+types+using+opaque+types>

Seacord, R & modified by Schiela, R. (2018, May) Top 10 Secure Coding Practices. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

Seacord, R. & modified by Herter, J. (2020, August) STR02-C. Sanitize data passed to complex subsystems. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems>

Seacord, R. & modified by Rozenau, M. (2021, February) INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/c/INT31-C.+Ensure+that+integer+conversions+do+not+result+in+lost+or+misinterpreted+data>

Svoboda, D. & modified by Herter, J. (2020, September) ERR55-CPP. Honor exception specifications. *Carnegie Mellon University.* Retrieved from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR55-CPP.+Honor+exception+specifications>