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



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

# Overview

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

# Purpose

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

# Scope

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

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate untrusted input data. Most of the major software vulnerabilities can be prevented by validating input data from untrusted sources. Not validating input data can allow exploits such as buffer overflows. |
| 1. Heed Compiler Warnings | Use the highest warning level your compiler has available. Eliminate warnings by modifying the code rather than turning off warnings. Use additional tools such as static analysis to further scrutinize your code. For C/C++ compilers the ‘-Wall’ flag should be used to enforce all warnings. |
| 1. Architect and Design for Security Policies | When designing your software architecture, be sure to implement and enforce strict security policies. |
| 1. Keep It Simple | Write clean, simple code to avoid the security implications of a large and complex code base. Try to use the least amount of code as is necessary to get the task accomplished for security, performance, and ease of maintenance. |
| 1. Default Deny | When authorizing users, the default option should always be to deny access. The access standard should be based on permission rather than exclusion. |
| 1. Adhere to the Principle of Least Privilege | Processes should always have the least amount of privileges necessary to execute the task. Permissions shall only be elevated temporarily for the duration of a task that requires elevated permissions. |
| 1. Sanitize Data Sent to Other Systems | Data that needs to be sent to subsystems must be sanitized to prevent manipulation of the subsystems by bad actors. Not sanitizing data sent to subsystems such as a database can allow exploits through SQL injection. |
| 1. Practice Defense in Depth | Practice defense in depth with multiple layers of security, so when one layer of defense is defeated, there are additional layers to prevent security vulnerabilities. An example of implementing defense in depth would be using a firewall, encrypting data flowing through the network, encrypting data at rest, and securing the physical location of the server itself through locked doors and security personnel or using a cloud server. |
| 1. Use Effective Quality Assurance Techniques | Using quality assurance techniques greatly increases the chances of security vulnerabilities to be discovered and patched leading to a more secure system. This can be accomplished implementing multiple testing phases and using both internal and external security reviews. |
| 1. Adopt a Secure Coding Standard | Adopt a secure coding standard for development for both the language and platform you choose to use. |

## C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Pointer to integer or integer to pointer conversion.  Pointer-to-integer and integer-to-pointer conversions are implementation defined and may have undesired consequences when converting between integers and pointers. |

| **Noncompliant Code** |
| --- |
| The size of a pointer can be greater than the size of an integer, the 64-bit pointer cannot be represented by the 32 bit pointer: |
| void f(void) {  char \*ptr;  /\* ... \*/  unsigned int number = (unsigned int)ptr;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| If the implementation supports the uintptr\_t type, any valid pointer to void can be converted to intptr\_t or uintptr\_t and back with no change in value: |
| #include <stdint.h>    void f(void) {  char \*ptr;  /\* ... \*/  uintptr\_t number = (uintptr\_t)ptr;  /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** (2, 4) Heed Compiler Warnings and Keep It Simple, data may be corrupted on some systems but work fine on others. Enable the highest compiler warnings and use additional static analysis to scrutinize your code. Keep it simple by utilizing standard data types. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wint-to-pointer-cast, -Wint-conversion | Detects some instances of this rule but not all |
| CodeSonar | 6.1p0 | LANG.CAST.PC.CONST2PTR  LANG.CAST.PC.INT | Converts: integer constant to pointer, pointer/integer |
| Parasoft C/C++test | 2021.1 | CERT\_C-INT36-b | Only 'uintptr\_t' or 'intptr\_t' |
| RuleChecker | 20.10 | pointer-integral-cast  pointer-integral-cast-implicit  function-pointer-integer-cast  function-pointer-integer-cast-implicit | Fully checked |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Guarantee indices and iterators are within valid range.  Array and other containers (such as std::vector) references must be within the bounds of the array. |

| **Noncompliant Code** |
| --- |
| The function attempts to do a range check to ensure it does not exceed bounds of the array, but it only checks the upper bound and does not check the lower bound which could possibly lead to accessing a negative index: |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, int pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

| **Compliant Code** |
| --- |
| The parameter pos is declared as size\_t rather than a plain int, which prevents the passing of negative arguments ensuring the array does not attempt to access a negative element: |
| #include <cstddef>    void insert\_in\_table(int \*table, std::size\_t tableSize, std::size\_t pos, int value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

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

|  |
| --- |
| **Principles(s):** (1, 2, 7) Validate Input Data, Heed Compiler Warnings, and Sanitize Data Sent to Other Systems. This can lead to a vulnerability causing buffer overflows and underflows allowing the possibility of arbitrary code execution. Enabling the highest compiler warnings and using static code analysis could be used to help detect noncompliant code. Sanitizing data sent to other systems can help prevent harmful code injection through this vulnerability. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TO  LANG.MEM.TU  LANG.MEM.TBA  LANG.STRUCT.PBB  LANG.STRUCT.PPE | Buffer overrun  Buffer underrun  Type overrun  Type underrun  Tainted buffer access  Pointer before beginning of object  Pointer past end of object |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2021a | CERT C++: CTR50-CPP | Checks for:  - Array access out of bounds  - Array access with tainted index  - Pointer dereference with tainted offset  Rule partially covered. |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Never attempt to modify string literals.  String literals are a pointer to a fixed-size array of characters. Attempting to modify a string literal leads to undefined behavior or an access violation because string literals are usually stored in read-only memory. |

| **Noncompliant Code** |
| --- |
| Attempting to modify a string literal is undefined behavior and usually leads to an access violation: |
| char \*str = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| Using an array initializer creates a copy of the string literal in the space allocated to the character array which can be modified safely: |
| char str[] = "string literal";  str[0] = 'S'; |

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

|  |
| --- |
| **Principles(s):** (3, 2) Architect and Design for Security Policies and Heed Compiler Warnings. Choosing a good compiler collection for your project will warn you or even flat out fail to compile if you attempt to modify a string literal. Enabling all compiler warnings by is a great way to implement and enforce strict security policies. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | string-literal-modfication  write-to-string-literal | Fully checked |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| RuleChecker | 20.10 | string-literal-modfication | Partially checked |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Exclude user input from format strings.  Never call a formatted I/O function with a format string as it creates a vulnerability for an attacker to fully or partially control the contents of a format string. The attacker can utilize this vulnerability to execute arbitrary code with the permissions of the vulnerable process. |

| **Noncompliant Code** |
| --- |
| This function creates an error message that is output to stderr using the C standard fprintf() function creating a format-string vulnerability: |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const char \*user) {  int ret;  /\* User names are restricted to 256 or fewer characters \*/  static const char msg\_format[] = "%s cannot be authenticated.\n";  size\_t len = strlen(user) + sizeof(msg\_format);  char \*msg = (char \*)malloc(len);  if (msg == NULL) {  /\* Handle error \*/  }  ret = snprintf(msg, len, msg\_format, user);  if (ret < 0) {  /\* Handle error \*/  } else if (ret >= len) {  /\* Handle truncated output \*/  }  fprintf(stderr, msg);  free(msg);  } |

| **Compliant Code** |
| --- |
| This solution passes the untrusted user input as variadic arguments to fprintf() rather than a format string eliminating the vulnerability: |
| #include <stdio.h>    void incorrect\_password(const char \*user) {  static const char msg\_format[] = "%s cannot be authenticated.\n";  fprintf(stderr, msg\_format, user);  } |

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

|  |
| --- |
| **Principles(s):** (1, 6, 7, 2) Validate Input Data, Adhere to the Principle of Least Privilege, Sanitize Data Sent to Other Systems, and Heed Compiler Warnings. This can lead to a vulnerability allowing attackers who fully or partially control the contents of a format string to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location. Processes should always have the least amount of privileges necessary to execute the task to prevent the attacker from executing arbitrary code with the permissions of the vulnerable process. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-FIO30 | Partially implemented |
| CodeSonar | 6.1p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| PC-lint Plus | 1.4 | 592 | Reports only non-literal format strings |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Do not access freed memory.  Dangling pointers are pointers to memory that has been deallocated. Accessing a dangling pointer can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this example, p is freed before p->next is executed; therefore, causing p->next to point to memory that has already been freed: |
| #include <stdlib.h>    struct node {  int value;  struct node \*next;  };    void free\_list(struct node \*head) {  for (struct node \*p = head; p != NULL; p = p->next) {  free(p);  }  } |

| **Compliant Code** |
| --- |
| This is corrected by storing a reference to p->next in q before freeing p to prevent accessing memory that has already been freed: |
| #include <stdlib.h>    struct node {  int value;  struct node \*next;  };    void free\_list(struct node \*head) {  struct node \*q;  for (struct node \*p = head; p != NULL; p = q) {  q = p->next;  free(p);  }  } |

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

|  |
| --- |
| **Principles(s):** (2, 9) Heed Compiler Warnings and Use Effective Quality Assurance Techniques. Dangling pointers can lead to security vulnerabilities, abnormal terminations, race conditions, and other unexpected behaviors. Quality Assurance techniques can be utilized to test thoroughly and discover any dangling pointers. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 6.1p0 | ALLOC.UAF | Use after free |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Use a static assertion to test the value of a constant expression.  The assert() macro is useful for identifying incorrect assumptions but not for runtime error checking. The assert() macro also creates a runtime overhead and calls abort(), meaning it is not suitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This code uses the assert() macro to assert a constant expression: |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| Use a preprocessor conditional statement instead of assert() for constant expressions: |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))  #error "Structure must not have any padding"  #endif |

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

|  |
| --- |
| **Principles(s):** (4, 3) Keep It Simple, Architect and Design for Security Policies. Assertions are a simple, clean, and quick way to add little sanity checks to your code. The overhead can be completely removed from release builds negating any performance hits. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| CodeSonar | 6.1p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Handle all exceptions.  An uncaught exception calls std::terminate() which calls std::abort() and abnormally terminates the process, which is not good for servers or embedded systems. Abnormal process terminations are a typical vector for denial-of-service (DDOS) attacks. |

| **Noncompliant Code** |
| --- |
| Neither f() nor main() catch exceptions thrown by throwing\_func() causing std::terminate() to be called: |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| The main entry point handles all 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, 4, 9, 2) Architect and Design for Security Policies, Keep It Simple, Use Effective Quality Assurance Techniques, and Heed Compiler Warnings. Utilizing exceptions is an excellent, simple, and clean way to handle errors when done properly, but not catching an exception can lead to abnormal termination causing destructors to not be called leaving external resources in an indeterminate state. Uncaught exceptions can be mitigated by heeding compiler warnings and using effective quality assurance techniques. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | main-function-catch-all  early-catch-all | Partially checked |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Polyspace Bug Finder | R2021a | CERT C++: ERR51-CPP | Checks for unhandled exceptions |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output | STD-008-CPP | Close files as soon as they are no longer needed.  A file open function must be matched with a file close call before the return value pointer of the call has ended or before normal program termination. |

| **Noncompliant Code** |
| --- |
| The constructor opens the file and calls std::terminate(), calling std::abort(), which does not call destructors causing the object to not be 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** |
| --- |
| File close is called before std::terminate(), 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):** (4, 2) Keep It Simple and Heed Compiler Warnings. Failing to close opened files can leave external resources in an indeterminate state causing a memory leak. Memory leaks can cause running programs to run out of memory and slow down to a halt or crash. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Klocwork | 2021.1 | RH.LEAK | Leak |
| Polyspace Bug Finder | R2021a | CERT C++: FIO51-CPP | Checks for resource leak |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | STD-009-CPP | Do not destroy a mutex while it is locked.  For threading to function properly, mutex objects are used to protect shared data from being accessed concurrently. So if a mutex object is destroyed while a thread is blocked waiting for the lock, the shared data is no longer protected. |

| **Noncompliant Code** |
| --- |
| This code contains a race condition allowing the mutex to be destroyed while it is still owned: |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {  std::lock\_guard<std::mutex> lk(\*pm);    // Access data protected by the lock.  }    void start\_threads() {  std::thread threads[maxThreads];  std::mutex m;    for (size\_t i = 0; i < maxThreads; ++i) {  threads[i] = std::thread(do\_work, i, &m);  }  } |

| **Compliant Code** |
| --- |
| This compliant solution eliminates the race condition by extending the lifetime of the mutex. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {  std::lock\_guard<std::mutex> lk(\*pm);    // Access data protected by the lock.  }    std::mutex m;    void start\_threads() {  std::thread threads[maxThreads];    for (size\_t i = 0; i < maxThreads; ++i) {  threads[i] = std::thread(do\_work, i, &m);  }  } |

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

|  |
| --- |
| **Principles(s):** (9, 2) Use Effective Quality Assurance Techniques and Heed Compiler Warnings. If a mutex object is destroyed while a thread waiting for the lock, sections of shared data are no longer protected leading to data corruption and undefined behavior. Using effective quality assurance techniques and heeding compiler warnings can help detect and patch noncompliant code. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++4961, C++4962 | Do not destroy another thread's mutex |
| Klocwork | 2021.1 | CERT.CONC.MUTEX.DESTROY\_WHILE\_LOCKED | Do not destroy another thread's mutex |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-CON50-a | Do not destroy another thread's mutex |
| Polyspace Bug Finder | R2021a | CERT C++: CON50-CPP | Checks for destruction of locked mutex |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | STD-010-CPP | Do not use std::rand() for generating pseudorandom numbers.  Pseudorandom number generators use a seed and math to generate a seemingly random number sequence, but these numbers are not genuinely random. |

| **Noncompliant Code** |
| --- |
| Using std::rand() produces ID numbers that are predictable and have very little randomness: |
| #include <cstdlib>  #include <string>    void f() {  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed  // by a random integer in the range [0-10000].  id += std::to\_string(std::rand() % 10000);  // ...  } |

| **Compliant Code** |
| --- |
| This compliant solution uses the Mersenne Twister algorithm as the engine for generating random values: |
| #include <random>  #include <string>    void f() {  std::string id("ID"); // Holds the ID, starting with the characters "ID" followed  // by a random integer in the range [0-10000].  std::uniform\_int\_distribution<int> distribution(0, 10000);  std::random\_device rd;  std::mt19937 engine(rd());  id += std::to\_string(distribution(engine));  // ...  } |

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

|  |
| --- |
| **Principles(s):** (4, 2) Keep It Simple and Heed Compiler Warnings. Usage of rand() does not generate truly random numbers and should be avoided. Not using a true random function can create guessable and repeatable numbers and patterns that can be exploited. Using a true random number generator and heeding compiler warnings to detect usages can mitigate the issue. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | bad-function (AUTOSAR.26.5.1A) | Fully checked |
| Clang | 4.0 | cert-msc50-cpp | Checked by clang-tidy |
| CodeSonar | 6.1p0 | BADFUNC.RANDOM.RAND | Use of rand |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MSC50-a | Do not use the rand() function |

## 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.

Diagram, timeline

Description automatically generated

As seen in the DevSecOps diagram above, the overall flow of the DevOps process remains the same while security automation is integrated into key areas throughout the entire process. For example: at the beginning during the creation phase, security tools are integrated into the IDE to provide security features such as automated static code analysis. Security tool training is added to the planning phase as well to ensure proper usage of security tools. Throughout the entire process, monitoring and analytics is implemented. Upon release, software signing takes place, and several verifications and integrity checks occur. Lastly, penetration testing occurs, and the process repeats as needed.

## Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Probable | High | P2 | L3 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | Low | Likely | Low | P9 | L2 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STD-009-CPP | Medium | Probable | High | P4 | L3 |
| STD-010-CPP | Medium | Unlikely | Low | P6 | L2 |

## 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 | Data at rest is data that is stored for an extended period of time and is usually protected by a firewall or anti-virus software. Encryption at rest prevents an attacker from accessing the data in the event of an intrusion by encrypting data on disc. All data at rest should be encrypted at all times. |
| Encryption at flight | Data at flight is data that is being transmitted from one location to another. Protecting data while it is being transmitted via encryption prevents it from being intercepted and read during transmission. Data should be encrypted at flight using protocols such as SSH and HTTPS (SSL/TLS.) |
| Encryption in use | Data in use is data that is stored in memory (RAM) or CPU caches/registers. Data in use encryption allows you to use the data while it remains encrypted preventing unauthorized access to unencrypted data from an attacker using a memory access exploit. An example would be using a hash of the original data when making comparisons such as password verifications. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is a process that confirms if a user has access to the system. This is typically accomplished through a login system with a username and password prompt. Security is further improved by using a multi-tier or 2-step authentication method. |
| Authorization | Authorization determines what resources a user has access to in the system. For example, you may want an administrator to have the authority to add or delete user accounts, but you would not want a standard user to have access to those administrative features. |
| Accounting | Accounting is the process of monitoring and logging user activities. For example, you may want to keep track of what databases a user has accessed and shell commands a user has executed. Typically, you would log and keep track of the who (user) what (action) when (time) and where (database.) |

**\***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 | 7/27/2021 | Revision: Added core security principles and coding standards. | Cameron Winningham |  |
| 1.2 | 8/7/2021 | Final Revision: Added risk assessment, automated detection, and policies. | Cameron Winningham |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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