**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 | All input data by command line, application input, file reads should be validated to ensure that the data is usable and safe for processing. Check for any fields required to ensure processing can function as appropriately**.** |
| 1. Heed Compiler Warnings | Addressing all concerns of compiler warnings can help prevent security issues up front as well as correcting any software processing issues before they are found during testing phase. |
| 1. Architect and Design for Security Policies | By designing for security and architecture up front before starting the development phase can help future refactoring once a security or vulnerability is found during the development or testing phase. |
| 1. Keep It Simple | By building smaller simple module components that do one thing, it can help with better code reuse and future understanding by engineers for addressing future issues as well as implementing newer features. Also better allows patterns to be used with design and development. |
| 1. Default Deny | Everyone at first is considered to not be allowed to access the system unless the user can be authenticated and authorized. |
| 1. Adhere to the Principle of Least Privilege | Only assign the least significant access to a process needed to perform. Don’t over grant permissions just because it seems like the normal (default) thing to do. Take time to evaluate every permission and assign only what’s needed. |
| 1. Sanitize Data Sent to Other Systems | Ensure that data sent to other systems via communication or file transfer methods are only sent with data formatted as required and not any extra sensitive secure data is sent by accident. |
| 1. Practice Defense in Depth | Ensure that multiple security mechanism, policies and tools are in place for protecting external and internal systems to ensure protection of data and processing. |
| 1. Use Effective Quality Assurance Techniques | Testing procedures must adopt an industry wide standard for unit, regression, system, integration and final release candidate testing to ensure the highest standards of testing are done for each application. |
| 1. Adopt a Secure Coding Standard | Engineers must understand the rules regarding secure coding principles for the applications they develop. While some technologies may allow certain features of coding we must still restrict use of said features that are deemed as unsecured from being used for our application. |

### 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 objects with appropriate storage durations. |

| **Noncompliant Code** |
| --- |
| This example returns a pointer which once exits the scope is no longer valid due to the scope of the declaration. |
| **char** \*init\_array(**void**) {  **char** array[10];    /\* Initialize array \*/  **return** array;  } |

| **Compliant Code** |
| --- |
| This example ensures that any variables (such as arrays) are declared separately so that they will still be valid when later referenced with a pointer |
| void init\_array(char \*array, size\_t len) {  /\* Initialize array \*/  return;  }    int main(void) {  char array[10];  init\_array(array, sizeof(array) / sizeof(array[0]));  /\* ... \*/  return 0;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – By use of such a pattern, we should uphold proper use of pointers in a way that ensures the intended outcomes. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | pointered-deallocation return-reference-local | Fully checked |
| Axivion Bahhaus Suite | 7.2.0 | CertC-DCL30 | Fully implemented |
| Klocwork | 2023.4 | LOCRET.ARG LOCRET.GLOB LOCRET.RET | Fully implemented |
| LDRA tool suite | 9.71 | 42 D, 77 D, 71 S, 565 S | Enhanced Enforcement |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CLG | Do not dereference null pointers. |

| **Noncompliant Code** |
| --- |
| This example doesn’t check for a null pointer for instance if malloc failed and returned null. |
| void f(const char \*input\_str) {  size\_t size = strlen(input\_str) + 1;  char \*c\_str = (char \*)malloc(size);  memcpy(c\_str, input\_str, size);  /\* ... \*/  free(c\_str);  c\_str = NULL;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This example validates the conditional that the pointers used aren’t null before deferencing. |
| void f(const char \*input\_str) {  size\_t size;  char \*c\_str;    if (NULL == input\_str) {  /\* Handle error \*/  }    size = strlen(input\_str) + 1;  c\_str = (char \*)malloc(size);  if (NULL == c\_str) {  /\* Handle error \*/  }  memcpy(c\_str, input\_str, size);  /\* ... \*/  free(c\_str);  c\_str = NULL;  /\* ... \*/  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Ensure proper checking of pointers before being dereferenced to ensure proper secure actions without unintended consequences. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Null-dereferencing | Fully checked |
| CodeSonar | 8.0p0 | LANG.MEM.NPD LANG.STRUCT.NTAD LANG.STRUCT.UPD | Null pointer dereference Null test after dereference Unchecked parameter dereference |
| Helix QAC | 2023.4 | DF2810, DF2811, DF2812, DF2813 | Fully implemented |
| TrustinSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CLG | Guarantee that storage for strings has sufficient space for character data and the null terminator. |

| **Noncompliant Code** |
| --- |
| This example uses the gets( ) function which should never be used because it can allow writing of data past the allocated size of the character string. |
| void func(void) {  char buf[BUFFER\_SIZE];  if (gets(buf) == NULL) {  /\* Handle error \*/  }  } |

| **Compliant Code** |
| --- |
| This example makes use of an explicit buffer size to be used when declaring and using with the fgets( ) function to ensure writing of data stops at the specific size and manipulates the newline to a null terminator for the string. |
| enum { BUFFERSIZE = 32 };    void func(void) {  char buf[BUFFERSIZE];  int ch;    if (fgets(buf, sizeof(buf), stdin)) {  /\* fgets() succeeded; scan for newline character \*/  char \*p = strchr(buf, '\n');  if (p) {  \*p = '\0';  } else {  /\* Newline not found; flush stdin to end of line \*/  while ((ch = getchar()) != '\n' && ch != EOF)  ;  if (ch == EOF && !feof(stdin) && !ferror(stdin)) {  /\* Character resembles EOF; handle error \*/  }  }  } else {  /\* fgets() failed; handle error \*/  }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – The size of the buffer should be checked to ensure that it can hold any buffer data source plus the null terminator character. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported Astree reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold data. |
| CodeSonar | 8.0p0 | LANG.MEM.BO LANG.MEM.TO MISC.MEM.NTERM BADFUNC.BO.\* | Buffer overrun Type overrun No space for null terminator A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Coverity | 2017.07 | STRING\_OVERFLOW BUFFER\_SIZE OVERRUN STRING\_SIZE | Fully implemented |
| TrustinSoft Analyzer | 1.38 | mem\_access | Exhaustively verified |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CLG | Sanitize data passed to complex subsystems. |

| **Noncompliant Code** |
| --- |
| This example shows that without validation, the expected parameter could be anything that could manipulate the intended system call by executing other processes by passing a “;” in the string. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| This example uses a whitelisting approach that ensures that any bad characters are manipulated to ensure subsystems are protected from exploits. |
| 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):** Sanitize Data Sent to Other Systems – We should ensure that the data intended to be transmitted to another system should have the proper data it expects. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Klockwork | 2023.4 | NNTS.TAINTED SV.TAINTED.INJECTION |  |
| Parasoft C/C++test | 2023.1 | CERT\_C-STR02-a CERT\_C-STR02-b CERT\_C-STR02-c | Protect against command injection Protect against file name injection Protect against SQL injection |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CLG | Allocate and free memory in the same module, at the same level of abstraction. |

| **Noncompliant Code** |
| --- |
| This example shows an example of double free scenario where memory is allocated in the parent function and then freed in a function called by the parent. |
| enum { MIN\_SIZE\_ALLOWED = 32 };    int verify\_size(char \*list, size\_t size) {  if (size < MIN\_SIZE\_ALLOWED) {  /\* Handle error condition \*/  free(list);  return -1;  }  return 0;  }    void process\_list(size\_t number) {  char \*list = (char \*)malloc(number);  if (list == NULL) {  /\* Handle allocation error \*/  }    if (verify\_size(list, number) == -1) {  free(list);  return;  }    /\* Continue processing list \*/    free(list);  } |

| **Compliant Code** |
| --- |
| This example corrects the double free error by having the function called by the parent return a value to be understood as an error and allows the parent to free the memory allocated within it. |
| enum { MIN\_SIZE\_ALLOWED = 32 };    int verify\_size(const char \*list, size\_t size) {  if (size < MIN\_SIZE\_ALLOWED) {  /\* Handle error condition \*/  return -1;  }  return 0;  }    void process\_list(size\_t number) {  char \*list = (char \*)malloc(number);    if (list == NULL) {  /\* Handle allocation error \*/  }    if (verify\_size(list, number) == -1) {  free(list);  return;  }    /\* Continue processing list \*/    free(list);  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Standard introduced to effectively prevent double freeing in which pointers or allocated memory is in use across different modules of the application. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.0p0 | ALLOC.DF ALLOC.LEAK | Double free Leak |
| Coverity | 6.5 | RESOURCE\_LEAK | Fully implemented |
| Parasoft Insure++ |  |  | Runtime analysis |
| PC-lint Plus | 1.4 | 449, 2434 | Partially supported |

#### Coding Standard 6

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

| **Noncompliant Code** |
| --- |
| This example uses memory mapped code with the assert. This will fail at compile time since it requires the sizes to be calculated at runtime. |
| 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** |
| --- |
| This example corrects the previous by using static\_assert to ensure the condition is true at compile time. |
| struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** Use Effective Quality Assurance Techniques – all asserts should only use values that had already been calculated or processed before the assert can be called. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Checked by clang-tidy |
| CodeSonar | 8.0p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 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-CLG | Do not throw an exception across execution boundaries. |

| **Noncompliant Code** |
| --- |
| This example shows a declared library function that throws an exception across execution boundaries. |
| // library.h  void func() noexcept(false); // Implemented by the library    // library.cpp  void func() noexcept(false) {  // ...  if (/\* ... \*/) {  throw 42;  }  }    // application.cpp  #include "library.h"    void f() {  try {  func();  } catch(int &e) {  // Handle error  }  } |

| **Compliant Code** |
| --- |
| This code shows the library function returning an error value that the caller can understand, instead of throwing an exception. |
| // library.h  int func() noexcept(true); // Implemented by the library    // library.cpp  int func() noexcept(true) {  // ...  if (/\* ... \*/) {  return 42;  }  // ...  return 0;  }    // application.cpp  #include "library.h"    void f() {  if (int err = func()) {  // Handle error  }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – throwing exceptions can cause unintended repercussions of the applications and libraries that it integrates with. These unintended repercussions can cause security concerns for the application(s). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.3 | C++3809, C++3810 |  |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-ERR59a | Do not throw an exception across execution boundaries |
| Polyspace Bug Finder | R2023b | CERT C++: ERR59-CPP | Checks for exceptions raised from library interfaces (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| [Student Choice] | STD-008-CPP | Do not invoke virtual functions from constructors or destructors. |

| **Noncompliant Code** |
| --- |
| This example shows the based class calling a virtual function in its constructor and destructor which will never be executed when used by the inherited object by the methods called being virtual. |
| struct B {  B() { seize(); }  virtual ~B() { release(); }    protected:  virtual void seize();  virtual void release();  };    struct D : B {  virtual ~D() = default;    protected:  void seize() override {  B::seize();  // Get derived resources...  }    void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| This example corrects the situation by using non virtual functions which can be later overloaded by inherited classes but still use base class features within its owns class. |
| class B {  void seize\_mine();  void release\_mine();    public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }    protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };    class D : public B {  void seize\_mine();  void release\_mine();    public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }    protected:  void seize() override {  B::seize();  seize\_mine();  }    void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** Adopt a Secure Coding Standard – Virtual functions are functions that let derived classes define. Using within the constructor or destructor can cause unintended reprecussions. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Virtual-call-in-constructor invalid\_function\_pointer | Fully checked |
| Clan | 3.9 | Clang-analyuzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| LDRA Tool suite | 9.71 | 467 S, 92 D | Fully implemented |
| Polyspace Bug Finder | R2023b | CERT C++: OOP50-CPP | Checks for virtual function call from constructors and destructors (rule fully covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File I/O | STD-009-CLG | Do not copya FILE object. |

| **Noncompliant Code** |
| --- |
| This example passes the FILE object by address which allows fputs to overwrite the FILE object and could result in an access violation at runtime. |
| int main(void) {  FILE my\_stdout = \*stdout;  if (fputs("Hello, World!\n", &my\_stdout) == EOF) {  /\* Handle error \*/  }  return 0;  } |

| **Compliant Code** |
| --- |
| This example corrects the scenario by instead passing the pointer to fputs which then copies the values correctly to the FILE object. |
| int main(void) {  FILE \*my\_stdout = stdout;  if (fputs("Hello, World!\n", my\_stdout) == EOF) {  /\* Handle error \*/  }  return 0;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – Copying a FILE object can cause access violations to trigger at runtime. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | File-dereference | Partially checked |
| Clang | 3.9 | Misc-non-copyable-objects | Checked with clang-tidy |
| Coverity | 2017.07 | MISRA C 2012 Rule 22.5 | Partially implemented |
| LDRA toll suite | 9.71 | 591 S | Fully implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Error Handling | STD-010-CLG | Detects errors when converting a string to a number. |

| **Noncompliant Code** |
| --- |
| This code blindly assumes the buff character pointer contains numbers which can be understood and fit as an integer. |
| void func(const char \*buff) {  int si;    if (buff) {  si = atoi(buff);  } else {  /\* Handle error \*/  }  } |

| **Compliant Code** |
| --- |
| This example corrects the scenario where it explicitly checks the buff character pointer to be assured before converting the string to an integer. |
| void func(const char \*buff) {  char \*end;  int si;    errno = 0;    const long sl = strtol(buff, &end, 10);    if (end == buff) {  (void) fprintf(stderr, "%s: not a decimal number\n", buff);  } else if ('\0' != \*end) {  (void) fprintf(stderr, "%s: extra characters at end of input: %s\n", buff, end);  } else if ((LONG\_MIN == sl || LONG\_MAX == sl) && ERANGE == errno) {  (void) fprintf(stderr, "%s out of range of type long\n", buff);  } else if (sl > INT\_MAX) {  (void) fprintf(stderr, "%ld greater than INT\_MAX\n", sl);  } else if (sl < INT\_MIN) {  (void) fprintf(stderr, "%ld less than INT\_MIN\n", sl);  } else {  si = (int)sl;    /\* Process si \*/  }  } |

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

| **Principles(s):** Adopt a Secure Coding Standard – All functions called should check return value for unexpected errors especially when converting or casting between multiple data type variables. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err34-c | Checked by clang-tidy |
| CodeSonar | 8.0p0 | BADFUNC.ATOF BADFUNC.ATOI BADFUNC.ATOL BADFUNC.ATOLL (customization) | Use of atof Use of atoi Use of atoll Users can add custom checks for uses of other undesirable conversion functions |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |
| PC-lint Plus | 1.4 | 586 | Assistance provided |

### 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.

Automation should start in the Build phase to ensure quick finding of issues while the build takes place. This can occur on the engineer’s development environment first and then also be ran through a build pipeline during the Verify and Test phase. Later, Transition and health check can also run automation tools again that ensure that no missed issues are found in the release candidate build for proper coverage of the secure coding standards. Later, these automation tools again are used in the Maintain and Stabilize plan to ensure proper coverage based on regression when modifying the source code for developing solutions to issues.

Automation tools can include code analysis checkers as well as regression testing suites. By utilizing both in the previously mentioned phases of pre-production and production can ensure that security and reliance of the application systems are top notch and well ensured.

### 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 | High | Probably | High | P6 | L2 |
| STD-002-CLG | High | Likely | Medium | P18 | L1 |
| STD-003-CLG | High | Likely | Medium | P18 | L1 |
| STD-004-CLG | High | Likely | Medium | P18 | L1 |
| STD-005-CLG | High | Probable | Medium | P12 | L1 |
| STD-006-CLG | Low | Unlikely | High | P1 | L3 |
| STD-007-CLG | High | Probably | Medium | P12 | L1 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CLG | Low | Probable | Medium | P4 | L3 |
| STD-010-CLG | Low | Probable | Medium | P4 | L3 |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

### 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 stored in any database or data warehouse should be done with hardware encryption preferred rather than software encryption. All devices that read/write data to/from the following data sources must also be encrypted to ensure data restriction and that said data is protected. |
| Encryption at flight | Data being transmitted to external company locations must be encrypted. This includes any secure file transfers, emails or uploading/downloading from third party API services. Internal transfers however should be protected by network and hardware policies and as such can be transmitted without being encrypted. |
| Encryption in use | User accounts that use or depend on encrypted data must have security standards as proper authentication and authorizations to ensure decrypting the data before use. Remote networking requires VPN and as well as some authentication may require multifactor authentication before decrypting of data for use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | With use of authentication, users must prove by various mechanism that they are who they say they are. Some types of network authentication at the basic level involve username/password however this obsolete mechanism is of much concern since users can enter simple or re-used passwords. New mechanisms such as multi factor authentication (when users have access via communication mechanism, they can respond back with access codes to prove who they are) or keychain authentication which synchronizes codes across various user devices. |
| Authorization | Authorization defines what access or rights a user may have for various date sources. A user may be authorized to write however not read, update or delete data at the data source or via an API service. Authorization should be used and defined for each user who is setup to be involved with user groups (or departments). Then authorization rights can be simply defined for groups which applies to users assigned to groups (however, exceptions can occur where users may need more rights than other users within the group). |
| Accounting | Accounting is the basic auditing of user activity. User activity must be recorded in some log data source (that has only top-level rights for access by admins and other log auditors). These logs can say then when unauthorized attempted access occurs or various other actions a user may take (including recording key strokes for various protected systems as well as images from the session if including use of a GUI desktop/web/mobile application). |

**\***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 | 01/27/2024 | 3-2 Milestone: Coding Standards | Jason Farrell |  |
| 1.2 | 02/14/2024 | 6-2 Project One: Security Policy | Jason Farrell |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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