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



# Green Pace Secure Development Policy

# Steven Anderson

## 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 | Validating input data is making sure that the data being processed fits program. Making sure that the inputs follow class rules and fit the type and/or numeric ranges of the program input. |
| 1. Heed Compiler Warnings | Compiler warnings are important and you should not ignore them or try and hide them. Comprehending the reason for warnings can help you develop a better strategy to fix the problem. Eliminating compiler headers through modification to code should be a goal. |
| 1. Architect and Design for Security Policies | When developing and designing your software make sure the design the architecture to best implement security policies. This can be achieved through the use of subsystems if different privileges are needed. Making subsystems that each have their privilege set. |
| 1. Keep It Simple | Don’t make your program too complicated. Keep your design simple and limited in size. Overly complex designs can cause more errors with implementation and use. Also making your program more complex means that proving your program is secure becomes more complex. |
| 1. Default Deny | Access should by default be denied. You should design the program to allow access based off permissions and decisions. There should be conditions that need to be met for any access allowance to users. |
| 1. Adhere to the Principle of Least Privilege | Processes should only execute with the least set of privileges to run. And any permission about least set should only be allowed for limited time. It will help make sure that unauthorized access is kept down and limits the time any hackers would have to execute their code with the higher privileges. |
| 1. Sanitize Data Sent to Other Systems | Make sure information being set between subsystems is sanitized. This would include any information set to command shells, and any commercial components. It will help cut down on the chances of SQL injection or other kind of injection attacks. Additionally the subsystem being called doesn’t need to know the context for which it is called it is up the calling process to ensure sanitization of data is ensured. |
| 1. Practice Defense in Depth | Layers in cybersecurity defense work. Having multiple layers of defense can prevent security flaws from being exploited. If one layer of defense doesn’t catch or stop a security breach the next layers afterword increases the chances of catching it. So having the layers can help prevent the attack or at the least minimize the damage. |
| 1. Use Effective Quality Assurance Techniques | Testing techniques can be used to help find and remove system vulnerabilities. Things like penetration tests and source code audits should be employed to ensure the quality of the program. And having testing done both internally and with external testers can both help in finding any vulnerabilities. Having outside eyes might find problems you can’t see or didn’t notice. |
| 1. Adopt a Secure Coding Standard | Develop in a way that matches the secure coding standards or the language and platform you are developing in. If python has different standards than C++ use the standards defined by the language you are actively using. |

### 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** | **Do not modify the standard namespaces** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | The namespaces provide regions for declarations and decrease the chances of confliction between identifiers using other declarative regions. The standard library provides the namespace std for most of the standard declarations. Adding declarations or definitions to std or posix will lead to unresolved issues and is considered undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this code example the user declared x to the namespace std which is processed as undefined behavior. |
| namespace std {  int x;  } |

| **Compliant Code** |
| --- |
| The compliant solution here is not trying to declare x within the namespace std but instead into a namespace using a non-reserved name. |
| namespace nonstd {  int x;  } |

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

| **Principles(s):**   * Principle 9 use effective quality assurance techniques   + Modifying standard namespaces can lead to vulnerabilities within the code. * Principle 2 head compiler warnings   + If you modify the standard namespaces this will lead to complier warnings and can be fixed by changing the name |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | 6 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-DCL58 |  |
| CodeSonar | 8.1p0 | LANG.STRUCT.DECL.SNM | Modification of Standard Namespaces |
| Parasoft C/C++test | 2023.1 | CERT\_CPP-DCL58-a | Do not modify the standard namespaces ‘std’ and ‘posix’ |
| Polyspace Bug Finder | R2024a | CERT C++: DCL58-CPP | Checks for modification of standard namespaces (rule fully covered) |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Do not read uninitialized memory** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Local variables will assume unexpected values when they are told to read values before initialization, resulting in default-initialization. So, if it gets called later without being initialized it will still remain an indeterminate value. Reading these uninitialized variables will create problems with removing memory accesses by the compiler optimization. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example the code creates i without initializing it and then calls I for printing. This use of an uninitialized variable will result in undefined behavior. |
| void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| In this compliant example the code creates i and initializes it with the value 0 before the variable is called for printing. |
| void f() {  int I = 0;  std::cout << i;  } |

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

| **Principles(s):**   * Principle 1 validate input data   + By reading from uninitialized memory, you are not verifying the input data and an attacker can add a false value to the memory * Principle 10 adopt a secure coding standard   + Reading the uninitialized memory will cause a vulnerability in the code that can be used for attacks and is against the coding standards for secure coding |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | 12 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Uninitialized-local-read | Fully Checked |
| CodeSonar | 8.1p0 | LANG.MEM.UVAR | Uninitialized variable |
| Coverity | 2017.07 | UNINIT | Implemented |
| LDRA tool suite | 9.7.1 | 53 D, 69 D, 631 S, 652 S | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not attempt to create a std::string from a null pointer** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | When a std::string is used it uses the traits design pattern in order to implement the details of the string types. When passing a null pointer into implementations like length() it will be undefined behavior because it would dereference the null pointer. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example the code creates the std::string object with std::getenv. But it returns a null pointer leading to undefined behavior since the variable doesn’t exist. |
| void f() {  std::string temp(std::getenv(“Temp”));  If (!temp.empty()) {  }  } |

| **Compliant Code** |
| --- |
| In this compliant example the code creates the char \*tempPtrVal first to check for a null before the std::string is created. |
| void f() {  const char \*tempPtrVal = std::getenv(“Temp”);  std::string temp(tempPtrVal ? tempPtrVal : “”);  if (!temp.empty()) {  }  } |

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

| **Principles(s):**   * Principle 1 validate input data   + By creating a string out of a null pointer, you are not validating what data is being created into a string and it will cause undefined behavior. * Principle 3 architect and design for security policies   + Creating a string from a null pointer can cause problems within the code that fall under this policy since an attack could assign a value to what is the null pointer and the string will then take that as its data input. * Principle 10 adopt a secure coding standard   + In CPP you don’t want to create a string out of a null pointer since this will cause undefined behavior from dereferencing the null pointer |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | LANG.MEM.NPD | Null Pointer Dereference |
| Astrée | 22.10 | Assert\_failure |  |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-STR51-a | Avoid null pointer dereferencing |
| Polyspace Bug Finder | R2024a | CERT C ++: STR51-CPP | Checks for string operations on null pointer (rule partially covered) |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | When passing string data to complex subsystems they may contain special characters that when reaching the subsystem will initiate a command or action. This is a software vulnerability and would be a decrease in overall security. To prevent this kind of activity it is important for the calling system to sanitize the string data being passed to prevent these characters. The kind of subsystems can include things like the command processor, relational databases, external programs, and 3rd party components. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example the code the vulnerability is in Sun Sloaris TELNET daemon and allows attackers to gain access to the system with elevated privileges. This is because the call doesn’t sanitize the data before passing it. |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :    getenv("USER")),    0); |

| **Compliant Code** |
| --- |
| In this compliant version of the code the code adds the – to the end of the argument before the call. The – causes the getopt() from POSIX to stop interpreting options from the argument list. |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :    getenv("USER")), 0); |

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

| **Principles(s):**   * Principle 3 architect and design for security policies   + Like shown in the example when not sanitizing data between systems you can create problems where access can be granted with the wrong credentials by hijacking from attackers who will look for this vulnerability. If properly designed to avoid this it removes that risk and adheres to proper security policy. * Principle 7 sanitize data sent to other systems   + Other than the obvious both have sanitizing data as their focus this follows the principle since it is saying that all data that is being sent from one system to another complex system should be sanitized to avoid attackers from gaining access to the system. * Principle 8 practice defense in depth   + This principle also applies to this since by not sanitizing the data you create a vulnerability that could have been fixed through sanitizing and remove a crucial layer of defense in the system. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| 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** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | When trying to evaluate a pointer into memory if it has already been deallocated will result in undefined behavior. It is called a dangling pointer and pointers like this can create vulnerabilities. When memory is freed the pointers to it become invalid, so the data from the data’s location can appear as valid but be changed. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example the code s is called as a new instance of struct S but is then deleted freeing up the memory. It is then called after the deletion creating a vulnerability where attackers can exploit it by running their own code. |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| In this compliant code the code calls s as a new instance of Struct S but then calls s before deleting it. This prevents creating the vulnerability and ensures that s is called before the memory is freed. |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

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

| **Principles(s):**   * Principle 1 validate input data   + This standard is about accessing memory that has already been freed, if the input had been validated before trying to access it you would avoid the vulnerability of creating a dangling pointer. * Principle 3 architect and design for security policies   + This principle applies since by not properly designing the code you have created a dangling pointer and with proper access design in the code it can be avoided. * Principle 9 use effective quality assurance techniques   + Through testing the program using testing techniques you can find the dangling pointers and find out that it was caused by the attempt to access freed memory. This would identify the major vulnerability and allow it to be fixed. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | 18 | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 24.04 | Dangling\_pointer\_use | Supported  Astrée reports all accesses to freed allocated memory |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses, after its deallocation and double memory deallocations |
| Coverity | 2017.07 | 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 | 2023.1 | CERT\_C-MEM30-a | Do not use resources that have been freed |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are diagnostic tools used to find and remove software faults that could create vulnerabilities. If assert() or static\_assert() finds problems it will send out an error message. Also while not perfect the assert() macro can also cause a runtime error but is not meant to be use and shouldn’t be used as a runtime error checker. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code the assert() macro is used within a function instead of where the definition of the structure. Meaning the diagnostic check is only done while the program runs and only if the function is executed. |
| struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };    int func(void) {    assert(sizeof(struct timer) == sizeof(unsigned char)  + sizeof(unsigned it) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| In this compliant code example, the code uses static\_assert in order to run the diagnostic check during the compiling phase instead. Resulting in no runtime cost and not requiring an executing to run diagnostics. |
| **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):**   * Principle 3 architect and design for security policies   + When properly designing the code for security the design will place the macros in their proper place instead of inside the function. * Principle 9 use effective quality assurance techniques   + Using proper testing techniques will help prevent breaking this standard and find the false use of macros. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | 1 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| Axivion Bauhaus suite | 7.2.0 | CertC-DCL03 |  |
| ECLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Using functions like abort() and Exit() will terminate the program but without calling exit handlers and without executing destructors. This leaves open streams where the buffer data might not flush, they could not close and temporary files may not be removed. Leaving these files and network communications in an unspecified state. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example code there is a call to f which includes inside it throwing\_func(). When f is called it is recorded as an exit handler. |
| void throwing\_func() noexcept(false);  void f() {  Throwing\_func();  }  int main() {  if (0 != std::atexit(f)) {  }  } |

| **Compliant Code** |
| --- |
| In this compliant example code the code uses try and catch to run threw all exceptions thrown by throwing\_func() |
| void throwing\_func() noexcept(false);  void f() {  try {  throwing\_func();  }  Catch (…) {  }  }  int main() {  if (0 != std::atexit(f)) {  }  } |

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

| **Principles(s):**   * Principle 4 keep it simple   + This principle applies since by keeping it simple and not using early terminations you keep the system more secure and avoid attackers taking advantage of these early terminations to cause problems within the system. * Principle 8 practice defense in depth   + Not having the early terminations works with defense in depth by avoiding open buffer and data streams running that can be exploited by attackers hoping to damage the system. * Principle 9 use effective quality assurance techniques   + This principle applies since using proper tests you could find where open data or buffers from terminations are occurring. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Medium | 4 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | stdlib-use | Partially checked |
| CodeSonar | 8.1p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| LDRA tool suite | 9.7.1 | 122 S | Enhanced Enforcement |
| ParaSoft C/C++ test | 2023.1 | CERT\_CPP-ERR50-a  CERT\_CPP-ERR50-b  CERT\_CPP-ERR50-c  CERT\_CPP-ERR50-d  CERT\_CPP-ERR50-e  CERT\_CPP-ERR50-f  CERT\_CPP-ERR50-g  CERT\_CPP-ERR50-h  CERT\_CPP-ERR50-i  CERT\_CPP-ERR50-j  CERT\_CPP-ERR50-k  CERT\_CPP-ERR50-l  CERT\_CPP-ERR50-m  CERT\_CPP-ERR50-n | The execution of a function registered with 'std::atexit()' or 'std::at\_quick\_exit()' should not exit via an exception  Never allow an exception to be thrown from a destructor, deallocation, and swap  Do not throw from within destructor  There should be at least one exception handler to catch all otherwise unhandled exceptions  An empty throw (throw;) shall only be used in the compound-statement of a catch handler  Exceptions shall be raised only after start-up and before termination of the program  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point  Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s)  Function called in global or namespace scope shall not throw unhandled exceptions  Always catch exceptions  Properly define exit handlers  The 'abort()' function from the 'stdlib.h' or 'cstdlib' library shall not be used  Avoid throwing exceptions from functions that are declared not to throw  The 'quick\_exit()' and '\_Exit()' functions from the 'stdlib.h' or 'cstdlib' library shall not be used |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Obey the one-definition rule** |
| --- | --- | --- |
| Declaration and Initialization | [STD-008-CPP] | Don’t use the same definition for more than one non-inline function or variable. It will cause undefined behavior. The ODR rule only has one exception and that is with uses of #include. |

| **Noncompliant Code** |
| --- |
| Noncompliant example code shows two translation units defining classes within them of the same name but with different definitions. It will result in undefined behavior. |
| //a.cpp  struct S {  int a;  };  //b.cpp  class S {  int a;  }; |

| **Compliant Code** |
| --- |
| In this compliant example the ODR wasn’t violated as in both class definitions they are unique. By using the public declaration, the int in b.cpp isn’t the same as the one in a.cpp. |
| // a.cpp  namespace {  struct S {    int a;  };  }    // b.cpp  namespace {  class S {  public:    int a;  };  } |

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

| **Principles(s):**   * Principle 4 keep it simple   + This principle applies to this standard since keeping it simple would imply having one definition means if you need to look it up you will find it easier. Will make looking up each instance easier by following where they are defined based off what they do. * Principle 5 adhere to the principle of least privilege   + This principle applies to this standard since having multiple definitions would allow processes that rely on those to operate in more than one location if it doesn’t throw up undefined behavior first. Also could possibly allow someone with knowledge of these multiple definitions to insert their own to subvert the program logic. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | High | 3 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | type-compatibility  definition-duplicate  undefined-extern  undefined-extern-pure-virtual  external-file-spreading  type-file-spreading | Partially checked |
| CodeSonar | 8.1p0 | LANG.STRUCT.DEF.FDH  LANG.STRUCT.DEF.ODH | Function defined in header file  Object defined in header file |
| LDRA tool suite | 9.7.1 | 286 S, 287 S | Fully implemented |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-DCL60-a | A class, union or enum name (including qualification, if any) shall be a unique identifier |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use valid iterator ranges** |
| --- | --- | --- |
| Containers | [STD-009-CPP] | Iterating over elements of a container requires that the iterator must iterate over the valid range. This range is two iterators that refer to the first and then past-the-end elements of the range. Empty iterator ranges are valid but using two iterators which are invalid or don’t refer to the same container will result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example one each of the iterations of the loop the for\_each() compares iterator one with iterator two to find if they are equal, if they aren’t it will continue to increment the first one. Leading the end to past the begin and causing an undefined behavior. |
| void f(const std::vector<int> &c) {    std::for\_each(c.end(), c.begin(), [](int i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In the compliant example the begin comes before the end in the iterations within the loop. This will cause each iterations of the loop to increment the first iterator until it is equal to the second iterator. |
| void f(const std::vector<int> &c) {    std::for\_each(c.begin(), c.end(), [](int i) { std::cout << i; });  } |

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

| **Principles(s):**   * Principle 1 validate input data   + The principle use in this standard would be that if the range had been checked before trying to access you would have avoided the undefined behavior trying to access outside the range. Therefor validating how big a range you have for a container input will help avoid calling outside it. * Principle 9 use effective quality assurance techniques   + This principle is great for this since testing software would find this error and be able to flag and alert the user to the problem. In some cases an attacker could use this vulnerability if left unchecked to cause a system to fall into a runtime error. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | 6 | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | overflow\_upon\_dereference |  |
| CodeSonar | 8.1p0 | LANG.MEM.BO | Buffer Overrun |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn’t really a range  Do not compare iterator from different containers |
| Polyspace Bug Finder | R2024a | CERT C++: CTR53-CPP | Checks for invalid iterator range (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not delete an array through a pointer of the incorrect type** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | Do not delete an array object with a static pointer if that differs from the dynamic pointer type of the object. If an array is deleted with an incorrect pointer type it will lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| The noncompliant code shows the code trying to delete an object stored array Derived after storing it in pointer Base, a virtual. This will not work and result in undefined behavior. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Base \*b = new Derived[10];     // ...     delete [] b;  } |

| **Compliant Code** |
| --- |
| In this compliant code example the void f deletes []b successfully. This is because static type b is Derived and the pointer is Derived instead of the virtual Base. |
| struct Base {    virtual ~Base() = default;  };    struct Derived final : Base {};    void f() {     Derived \*b = new Derived[10];     // ...     delete [] b;  } |

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

| **Principles(s):**   * Principle 1 validate input data   + For this standard validating the input data would help prevent the problems that could arise. If the coder stops to make sure that where they are assigning input data is being stored the right way and in a correct type, they would avoid undefined behavior. * Principle 4 keep it simple   + This would also apply to the standard since keeping it simple would mean not having too many pointer types would limit the possibility of using the wrong one for deletion. Or having your pointer types properly separated in the code can prevent trying to use one in a part of the code that it wont work in. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | 2 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -analyzer-checker=cplusplus | Checked with clang – cc1 or (preferably) scan-build |
| CodeSonar | 8.1p0 | ALLOC.TM | Type mismatch |
| Parasoft C/C++ test | 2023.1 | CERT\_CPP-EXP51-a | Do not treat arrays polymorphically |
| Polyspace Bug Finder | R2024a | CERT C++: EXP51-CPP | Checks for delete operator used to destroy downcast object of different type |

### 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



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 tools should be used throughout the development process to best enforce the coding standards. If used throughout the development process any instance of noncompliance will be caught early and be fixed before it becomes more ingrained in the software and requires larger rewrites to fix. If used only at the end then automated tools could create a list of found problems that may be harder to work through than if used often. There should also be documentation of each automated test to see where any trends of noncompliance show up to fix these problems so they don’t keep reoccurring.

In regards to the above graphic the automated tools should be used in the pre-production verify and test section and, in the production, should be used in the monitor and detect step to ensure the security of the program. In the production side of the process, it is important to use the automated tools to check the code since the tools go through regular updates and may be better adapted to finding problems that earlier versions would have missed. Can also help to ensure formatting of the program remains consistent and no undetected changes have been made.

### 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 | 6 | 2 |
| STD-002-CPP | High | Probable | Medium | 12 | 1 |
| STD-003-CPP | High | Likely | Medium | 18 | 1 |
| STD-004-CPP | High | Likely | Medium | 18 | 1 |
| STD-005-CPP | High | Likely | Medium | 18 | 1 |
| STD-006-CPP | Low | Unlikely | High | 1 | 3 |
| STD-007-CPP | Low | Probable | Medium | 4 | 3 |
| STD-008-CPP | High | Unlikely | High | 3 | 3 |
| STD-009-CPP | High | Probable | High | 6 | 2 |
| STD-010-CPP | Low | Unlikely | Medium | 2 | 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 at rest | This form of encryption is the process of making sure that data and information is encrypted while being stored. This could be on a server or on a physical storage medium. Keeping the data encrypted ensures that attackers will not be able to access the data should they obtain access to the physical storage medium or physical server. |
| Encryption in flight | This is the form of encryption is the process of encrypting data and information while it is in transit. Meaning if the data is being moved from one source to another it should be encrypted during that movement to avoid attackers from hijacking the data in transit and accessing it. |
| Encryption in use | This is the form of encryption that means data is encrypted while being processed. This can be encryption used in memory for programs or to encrypt the runtime processes. While this form of encryption is very intensive on the network and system it protects data from being accesses by attackers while it is being used. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication in the Triple-A framework is the identification of a user and the process of giving them access to the network. Can be done through credential login systems or more complex methods such as biometrics and fingerprint scanning. This is a very important part of the Triple-A since it blocks unrecognized or unauthorized users from gaining access to sensitive data or network activity. |
| Authorization | Authorization in the Triple-A framework is access controls. Deciding what a user is allowed to access within the network based on what permissions they are authorized to have. It also determines what level of control they have over the data, if they can only view or if they can add, delete and modify as well. As a part of Triple-A this is important since it enforces the least privilege practice to limit what a would-be attacker can do with certain credentials. Helping to prevent cascading attacks into other parts of the network. |
| Accounting | Accounting is the final A in the Triple-A framework and has to deal with monitoring of the network. It is the part that logs all activity from users and will include data such as how long they accessed and what they accessed. As a part of Triple-A this is important since it can be used to track any attacks or review any changes made to data within the network. It can also help with allocation of capacity if certain parts of the network see a higher traffic flow then others to help keep the network running smooth. |

**\***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 | 09/21/2024 | First Revision | Steven Anderson |  |
| 1.2 | 10/12/2024 | Final Revision | Steven Anderson |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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

References:

* Martinez, J. (2024, September 27). What is AAA Security? Authentication, Authorization, and Accounting. *Strongdm*. <https://www.strongdm.com/blog/aaa-security>
* Dinic, M., & Dinic, M. (2024, April 29). *Understanding encryption - data at rest, in motion, in use*. Jatheon Technologies Inc. <https://jatheon.com/blog/data-at-rest-data-in-motion-data-in-use/>
* *OWASP Developer Guide | Principles of Security | OWASP Foundation*. (n.d.-b). <https://owasp.org/www-project-developer-guide/draft/foundations/security_principles/>
* *SEI External Wiki Home - Homepage - Confluence*. (n.d.). <https://wiki.sei.cmu.edu/confluence/>