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



Green Pace Secure Development Policy

Contents

[Overview 2](#_heading=h.30j0zll)

[Purpose 2](#_heading=h.1fob9te)

[Scope 2](#_heading=h.3znysh7)

[Module Three Milestone 2](#_heading=h.4d34og8)

[Ten Core Security Principles 2](#_heading=h.2s8eyo1)

[C/C++ Ten Coding Standards 3](#_heading=h.3rdcrjn)

[Coding Standard 1 4](#_heading=h.26in1rg)

[Coding Standard 2 5](#_heading=h.lnxbz9)

[Coding Standard 3 6](#_heading=h.35nkun2)

[Coding Standard 4 7](#_heading=h.1ksv4uv)

[Coding Standard 5 8](#_heading=h.2jxsxqh)

[Coding Standard 6 9](#_heading=h.z337ya)

[Coding Standard 7 10](#_heading=h.28h4qwu)

[Coding Standard 8 11](#_heading=h.nmf14n)

[Coding Standard 9 13](#_heading=h.37m2jsg)

[Coding Standard 10 14](#_heading=h.1mrcu09)

[Defense-in-Depth Illustration 15](#_heading=h.46r0co2)

[Project One 15](#_heading=h.2lwamvv)

[1.](#_heading=h.111kx3o) Revise the C/C++ Standards 15

[2.](#_heading=h.3l18frh) Risk Assessment 15

[3.](#_heading=h.206ipza) Automated Detection 15

[4.](#_heading=h.4k668n3) Automation 15

[5.](#_heading=h.2zbgiuw) Summary of Risk Assessments 16

[6.](#_heading=h.1egqt2p) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.3ygebqi) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.2dlolyb)

[Enforcement 18](#_heading=h.sqyw64)

[Exceptions Process 18](#_heading=h.3cqmetx)

[Distribution 19](#_heading=h.1rvwp1q)

[Policy Change Control 19](#_heading=h.4bvk7pj)

[Policy Version History 19](#_heading=h.2r0uhxc)

[Appendix A Lookups 19](#_heading=h.1664s55)

[Approved C/C++ Language Acronyms 19](#_heading=h.3q5sasy)

# 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 | This principle is focused on the verification of user input to determine if any malicious input is being passed into the system. This could be in the form of an input that causes a buffer overflow, or even a SQL Injection. |
| 1. Heed Compiler Warnings | This principle focuses on ensuring that we utilize the highest warning level that the compiler has available and modifying the codebase to eliminate any and all warnings. Compiler warnings tell us when there are potential or actual issues with our code, focusing on avoiding these warnings will help to ensure a stronger code base. |
| 1. Architect and Design for Security Policies | This principle focuses on creating the software architecture and designing the software to enforce your security policies. One example would be if you have a system heavily divided into different privileged components, you could divide said system into different subsystems that have different privileges assigned to them. |
| 1. Keep It Simple | The principle of keeping it simple refers to building systems without excess complexity as a system built with excess complexity has a higher chance that errors will occur during the implementation, configuration, and use of the said system. The more complex the system, the more resources required to ensure a satisfactory level of security to the users and the organization. |
| 1. Default Deny | This is a very interesting principle. It is based on the idea of basing access on permission instead of exclusion. This is a beneficial principle because we don’t need to exclude every possibility, but instead include only the situations where access to the system is permitted. |
| 1. Adhere to the Principle of Least Privilege | This principle involves the concept of limiting the amount of time that users spend at certain privilege levels. Each process in the system should execute at the lowest level/set of privileges required to complete the task. Attackers will be more limited as they won’t have elevated privileges for no reason, only when they are required for the system to execute a function. |
| 1. Sanitize Data Sent to Other Systems | The sanitization of data that is passed to subsystems is essential. This is directly connected to the SQL Injection attacks as improperly sanitized data could contain potentially malicious code that could negatively impact the system or the security of the data within the system. |
| 1. Practice Defense in Depth | Defense in Depth refers to layering your security approach so if one layer fails, another layer will be there to protect the system or prevent a flaw in one layer from becoming a full-fledged vulnerability. There are trade-offs with the time and resources needed to build these systems, so you must find a balance that you are comfortable with |
| 1. Use Effective Quality Assurance Techniques | This principle focuses on using quality assurance techniques to identify and eliminate vulnerabilities. This means to not limit yourself to one technique, but instead use a variety of techniques to better cover your application or system as a whole. Quality assurance can be done with external reviewers or as independent security reviews. |
| 1. Adopt a Secure Coding Standard | This principle is another essential as having a standard to build your secure systems is crucial to the overall security of the said system. This principle involves the creation and/or application of the aforementioned standard. |

## 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] | I will say that using the proper data type when performing calculations is crucial as some calculations may result in a number with decimal points. |

|  |
| --- |
| **Noncompliant Code** |
| This code divides two integers and stores the result in another integer. The result of the division is 0 because integers truncate the decimal value. |
| int a = 1;  int b = 2;  int c = a / b; |

|  |
| --- |
| **Compliant Code** |
| This code instead stores the value of the division in a double. This means our answer will be accurate to the decimal point. |
| int a = 1;  int b = 2;  double c = a / b; |

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

|  |
| --- |
| **Principles(s):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like proper data types are handled. The two I find are intertwined for that reason. Our coding standard should outline data type usage and cases. This will ensure that it is upheld to the required standards. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Low | Likely | Low | Medium | 8 |

**Automation**

|  |
| --- |
| **Tool** |
| I wasn’t able to find any tools for this. This would be an issue that would require static analysis of the code or unit testing to identify. |

### Coding Standard 2

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Value** | [STD-002-CPP] | When returning values from functions, they must return from all exit paths. |

|  |
| --- |
| **Noncompliant Code** |
| Only the if statement returns a value in the noncompliant code. If the value of **a** is not negative, the function will not return anything even though the function needs to return an integer. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

|  |
| --- |
| **Compliant Code** |
| The compliant code has all paths returning a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

|  |
| --- |
| **Principles(s):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like returning values from all paths are resolved. The two I find are intertwined for that reason. With our coding standard, we need to make clear that every path needs to return a value in order to be in compliance. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Medium | Likely | Medium | High | 3 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | return-implicit | Fully checked |
| CodeSonar | 6.0p0 | LANG.STRUCT.MRS | Missing return statement |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-MSC52-a | All exit paths from a function with non-void return type shall have an explicit return statement with an expression |
| Polyspace Bug Finder | R2020a | CERT C++: MSC52-CPP | Checks for missing return statements (rule partially covered) |

### Coding Standard 3

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **String Correctness** | [STD-003-CLG] | We want to avoid attempting to modify string literals in our code. Instead, the string stored in an array will allow us to modify that string. |

|  |
| --- |
| **Noncompliant Code** |
| In this code, we have the char pointer str which is initialized to the address of a string literal. We attempt to modify the string literal which is undefined behavior. |
| char \*str = "string literal";  str[0] = 'S'; |

|  |
| --- |
| **Compliant Code** |
| We initialize as an array in the compliant code. This creates a copy of the string literal and thus allows us to modify str 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):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like attempting to modify string literals are mitigated by initializing as an array. The two I find are intertwined for that reason. Proper QA would involve not only static analysis and some automated tools but also unit testing to ensure that we are in fact able to modify strings. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Likely | Low | High | 5 |

**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 \*" |
| Parasoft C/C++test | 2020.2 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified  Do not modify string literals |
| RuleChecker | 20.10 | string-literal-modfication | Partially checked |

### Coding Standard 4

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **SQL Injection** | [STD-004-CPP] | This standard deals with employing countermeasures to mitigate the risk of potential SQL Injections. |

|  |
| --- |
| **Noncompliant Code** |
| The below code shows a system that takes user input and calls a function that will then allow for the execution of a SQL statement on the system without checking if the statement is possibly an injection attack. Structure\_query adds the user input to the standard SQL query of the system and outputs **SQL.** run\_queries executes said query. |
| cout << “Enter User”;  cin >> user;  structure\_query(user);  run\_queries(sql); |

|  |
| --- |
| **Compliant Code** |
| The compliant code will have an additional function that checks to see if the user input contains certain characters that could indicate a potential injection attack. Structure\_query adds the user input to the standard SQL query of the system and outputs **SQL.** run\_queries executes said query. |
| bool run\_query(string sql)  {  // If the SQL command contains a " or ", it will fail and print the error message  // This is more of a blanket coverage for SQL Injection  if (sql.find(" or ") != std::string::npos)  {    std::cout << "Data failed to be queried from USERS table. ERROR = " << "Suspected SQL Injection" << std::endl;  sqlite3\_free(error\_message);  return false;  }  }  cout << “Enter User”;  cin >> user;  structure\_query(user);  run\_query(sql)  if(run\_query == true){  run\_queries(sql);  }  else  {  return 0;  } |

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

|  |
| --- |
| **Principles(s):** 1, 7, 8, 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like mitigating the risk of SQL injections. The two I find are intertwined for that reason. Within the coding standard, we need to outline the proper handling of input that is going to be executed as an SQL Query. In the above example, we looked for an “ or ” because the user was only expected to type in one input. This avoids them being able to enter another statement to execute. This is another standard where QA is crucial. We need to use the principle of validating input data as outlined above by checking if it’s valid, as well as sanitizing the data before it is sent to other systems, in turn, avoiding sending malicious data further into our systems. Since we are validating and sanitizing, I believe that we also are supported by the defense in depth principle as we are creating multiple levels of security to protect ourselves against attack. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Unlikely | Medium/High | High | 1 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 6.0p0 | IO.INJ.SQL | SQL injection |
| Parasoft C/C++test | 2020.2 | CERT\_C-STR02-c | Protect against SQL injection |

### Coding Standard 5

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Memory Protection** | [STD-005-CPP] | This standard is meant to address allocation failures from the default memory allocation operator. |

|  |
| --- |
| **Noncompliant Code** |
| In this code, two allocations are performed in the same expression. This will throw an exception because one of the calls to new could result in memory leakage. |
| struct A { /\* ... \*/ };  struct B { /\* ... \*/ };    void g(A \*, B \*);  void f() {  g(new A, new B);  } |

|  |
| --- |
| **Compliant Code** |
| In the compliant code below, we use std::unique\_ptr to manage the resources for both A and B objects. |
| #include <memory>    struct A { /\* ... \*/ };  struct B { /\* ... \*/ };    void g(std::unique\_ptr<A> a, std::unique\_ptr<B> b);  void f() {  g(std::make\_unique<A>(), std::make\_unique<B>());  } |

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

|  |
| --- |
| **Principles(s):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like memory leakage are avoided at all costs. The two I find are intertwined for that reason. Our coding standard should include things like using unique\_ptr’s to manage resources. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Likely | Medium | High | 3 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| LDRA tool suite | 9.7.1 | 45 D | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |
| Polyspace Bug Finder | R2020a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |

### Coding Standard 6

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Assertions** | [STD-006-CLG] | Assertions allow us to perform tests on the value of constant expressions. |

|  |
| --- |
| **Noncompliant Code** |
| The code is noncompliant because for the assert statement to be checked, it will only happen during runtime and if the code path containing the assertion is actually executed. |
| #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** |
| In the following code that is compliant, we use static\_assert. We could also use a preprocessor conditional statement. This is because static\_assert allows for incorrect assumptions to be made at compilation. Thus avoiding runtime errors. |
| #include <assert.h>    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):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like improper assertion usage are not found in our codebase. The two I find are intertwined for that reason. This means we need to properly outline different usages when it comes to assert and static\_assert to avoid unnecessary runtime errors. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Medium | Likely | Low | Medium | 6 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 6.0p0 | (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 |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |

### Coding Standard 7

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Exceptions** | [STD-007-CPP] | We need to avoid programs from abruptly terminating due to errors. |

|  |
| --- |
| **Noncompliant Code** |
| In the non-compliant code, the call to function f could result in a call to std::terminate() because throwing\_function() can result in an exception being thrown. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

|  |
| --- |
| **Compliant Code** |
| In this compliant code, function f handles all exceptions with throwing\_func() and doesn’t rethrow. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

|  |
| --- |
| **Principles(s):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and we are able to avoid abruptly terminating our program due to any potential errors. The QA for this one will focus on tools to check if we abruptly terminate while the coding standard will focus on not building our program to abruptly terminate. The combination of checks and balances is going to help us ensure that the system fits our requirements of functionality. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Medium | Unlikely | Low | Medium | 7 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 6.0p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| Parasoft C/C++test | 2020.2 | 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 |
| Polyspace Bug Finder | R2020a | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |

### Coding Standard 8

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Sanitization** | [STD-008-CLG] | This involves ensuring that certain characters are “Sanitized” from the input before it is moved into different subsystems to avoid passing potentially harmful inputs further into a system. |

|  |
| --- |
| **Noncompliant Code** |
| In this noncompliant code example, an address is inputted into a buffer and the string is an argument in a call to the system. This is fine for people without malicious intent but could be easily used to provide malicious code to the system. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

|  |
| --- |
| **Compliant Code** |
| In the compliant code we have outlined characters that are white listed by us. Removing specific characters from use allows us to block another way that an attacker could potentially manipulate the system. |
| 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):** 1, 7, 8, 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and that we properly sanitize data being sent to subsystems. QA would involve using tools to check if we are at risk for any type of injection attack, using unit testing to see if we properly sanitize our data, and with an area so special, I believe having static analysis would be crucial. This standard also is supported by two other principles. Those are, sanitize data sent to other systems and validate input data. I feel those two principles are closely related. Those two are present because we need to not only validate input data as it comes in but also sanitize it before it gets to subsystems on our network. That is why I believe that defense in depth also applies here. We are taking multiple precautions to prevent attacks. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Unlikely | Medium/High | High | 1 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 6.0p0 | IO.INJ.COMMAND  IO.INJ.FMT  IO.INJ.LDAP  IO.INJ.LIB  IO.INJ.SQL  IO.UT.LIB  IO.UT.PROC | Command injection  Format string injection  LDAP injection  Library injection  SQL injection  Untrusted Library Load  Untrusted Process Creation |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++test | 2020.2 | 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 |
| Polyspace Bug Finder | R2020a | CERT C: Rec. STR02-C | Checks for:  Execution of externally controlled command  Command executed from externally controlled path  Library loaded from externally controlled path  Rec. partially covered. |

### Coding Standard 9

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Dangling Pointer** | [STD-009-CLG] | This is connected to memory but is such an important topic so I decided to give it its own standard. This is a standard around not accessing memory that has been freed. |

|  |
| --- |
| **Noncompliant Code** |
| In this noncompliant example, p is freed before p->next is executed. This means that p->next tries to read 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** |
| In the compliant example, we store a reference to p->next in q before we free p. This solves the issue fo attempting to access freed memory. |
| #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):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and issues like dangling pointers are properly managed. For the QA side, we need to focus on using tools to verify that we aren’t attempting to access any freed memory. This should be accompanied by unit testing as well. The combination of proper QA in this situation with a coding standard that should help us avoid creating the dangling pointer in the first place should set us up for success. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | dangling\_pointer\_use | Supported  Astrée reports all accesses to freed allocated memory. |
| Axivion Bauhaus Suite | 6.9.0 | CertC-MEM30 | Detects memory accesses after its deallocation and double memory deallocations |
| CodeSonar | 6.0p0 | ALLOC.UAF | Use after free |
| 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 |

### Coding Standard 10

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Proper Naming** | [STD-010-CPP] | This standard deals with properly naming variables, objects, etc. Proper naming helps yourself and fellow engineers to understand a program much better at a glance because the names are understandable to them. |

|  |
| --- |
| **Noncompliant Code** |
| The non-compliant code has variables named with random letters and numbers, this is difficult for someone new to the codebase to understand and can even be difficult for you to understand if you spend some time away from it. |
| int a = 2340534;  int x = 52;  int y = 26;  double one = a / x;  double two = a / y; |

|  |
| --- |
| **Compliant Code** |
| The compliant code is much easier to follow. You can tell we have integers of salary, weeks, and paychecks. And we have doubles of weekly and biweekly which can lead us to extrapolate that this small snippet of code calculates your weekly and biweekly pay based on your salary. |
| int salary = 2340534;  int weeks = 52;  int paychecks = 26;  double weekly = salary / weeks;  double biweekly = salary / paychecks; |

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

|  |
| --- |
| **Principles(s):** 9, 10: The principles of using effective quality assurance techniques and adopt a secure coding standard apply here because, regarding the first one, we need to have proper QA to ensure that our secure coding standard has been executed properly and that everyone is utilizing proper naming conventions. This may seem like a small issue, and in the grand scheme of things it is, but it is also important. Having proper naming of variables and functions makes it easier for employees to understand the code that’s being written by coworkers. By setting a coding standard around naming, you are able to streamline processes and make the lives of those in QA much easier. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Low | Likely | Low | Low | 10 |

**Automation**

|  |
| --- |
| **Tool** |
| This is another standard where I don’t believe a tool is beneficial. I especially don’t believe so here because I couldn’t find one and I don’t believe one could exist for this standard. Naming conventions are however something that is very important to the company, so we will need a heavy amount of static analysis from employees to uphold this standard. |

## 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.

I believe I am understanding this correctly, so here it goes. Given the current DevOps process, I believe there are some slight modifications that can be made in order to automate the enforcement of the standards outlined in this policy. I believe that there should be thought put into the planning and design process to adapt the code to the standards and that we should keep a close eye on them as we are building, but crucially, using automation during and after the build is completed to check for any standards being broken should suffice as a way to uphold our system to the secure coding standards. This means we will be using automation in both the build step of DevSecOps and the Verify and test step to ensure that we meet and exceed all of our standards.

## 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 | Likely | Low | Medium | 8 |
| STD-002-CPP | Medium | Likely | Medium | High | 3 |
| STD-003-CLG | High | Likely | Low | High | 5 |
| STD-004-CPP | High | Unlikely | Medium/High | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 3 |
| STD-006-CLG | Medium | Likely | Low | Medium | 6 |
| STD-007-CPP | Medium | Unlikely | Low | Medium | 7 |
| STD-008-CLG | High | Unlikely | Medium/High | High | 1 |
| STD-009-CLG | High | Likely | Medium | High | 1 |
| STD-010-CPP | Low | Likely | Low | Low | 10 |

## Create Policies for Encryption and Triple A

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

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

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

|  |  |
| --- | --- |
| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| Encryption in rest | Encryption in rest is a way of protecting the local storage of our data through encryption. The purpose of encryption in rest is to protect the data of the application from attackers who can access the storage device but that doesn’t have access to the application itself, making it so they can’t actually read the data. This policy applies because, in order to have a truly defense-in-depth strategy, we need to protect our data in all states. |
| Encryption at flight | Encryption at flight is essentially just encryption in transit. As the data is being transmitted, the data should be sent through encrypted channels. One example would be using HTTPS. This policy applies because where we already have our data encrypted in rest, why would we decrypt the data to transmit it, opening ourselves up to attack and breaking down our defense in depth strategy? We shouldn’t, which is why we need encryption at flight. |
| Encryption in use | Finally, we have encryption in use which is essentially having the data encrypted until you authenticate a user has access to said data. This policy applies because like before if we open up one of the DiD components, we make the whole system potentially vulnerable. This is one of the most important in my eyes because it could be a user-facing application and if data isn’t properly stored, you could potentially release unencrypted data to a non-malicious user. Encrypting data until it is needed is a fantastic idea and a policy we need to uphold. |

|  |  |
| --- | --- |
| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| Authentication | The Authentication part of Triple-A is the part that pertains to identifying users. This will be done typically through a User Login like all of us should be familiar with as we use them nearly daily. Each user has their own unique combination of credentials to access the network. The credentials provided are compared with the credentials stored in the database. This policy applies because user authentication is crucial to protect your network from attackers, it is a great first layer of defense. |
| Authorization | After a user is authenticated, they must then obtain Authorization to perform tasks on the network. Their level of access to the system depends on the permission level set on their user account. Consumer type users may be able to navigate the network and manage their own account and access files they have permission to, but an administrative user could make modifications to the database, add new users to the system, and change permissions for other users. This policy applies because we need to practice defense-in-depth, after a user gains access, we should only provide them with enough permissions to complete tasks required by their user account, and only escalate permissions when absolutely necessary. That is supported by multiple secure coding principles. |
| Accounting | The final part is Accounting. This monitors the resources that each user consumes while on the network. This can be divided up into different pieces of data like the amount of time on the system, the data sent and received, and the files they accessed on the network. This is a great final defense-in-depth measure to help secure our network. If somehow an attacker bypasses all of our other security measures, at least now we will be able to see what they are doing/accessing on our network. It is the last resort, but it is also necessary if we wish to have the most secure system that we can build. |

**\***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 | 04/08/2021 | Completed Policy Document | Devon Darling |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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