**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 | The proper testing of any input supplied by a user or an application, often just referred to as Input or Data Validation, and it prevents improperly formed data from entering an information system. Most external data sources should be considered suspicious. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. |
| 1. Architect and Design for Security Policies | Create software architecture and design said software to implement and enforce security policies as much as possible. |
| 1. Keep It Simple | Stay simple in design and keep it as small as possible, in order to make adding and working on the code easy, while not having as much room for hidden errors. Scale it based on the work as things get more complex. |
| 1. Default Deny | Access decisions based on permission instead of exclusion. Access Denied by default. |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the least privileges necessary to complete the job, elevated permission should only be accessed for as little time as possible to complete the task. |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems needs to be sanitized to avoid attackers invoking unused functionality through SQL and other injection based attacks. |
| 1. Practice Defense in Depth | Risk management with multiple defense strategies that are layered, so that each layer of defense has a backup if it was inadequate and can further prevent security flaws from being exploitable by attackers. |
| 1. Use Effective Quality Assurance Techniques | Effective Quality Assurance must be good in identifying and then eliminating vulnerabilities, various testing like Fuzz or penetration testing, and source code audits, should all be implemented into programs, and independent security reviews can augment this for more secure systems. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language and platform. |

## C/C++ Ten Coding Standards

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

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Data can be signed or unsigned when being sent into the system, signed information should be converted to unsigned so they can be converted to larger integer sizes when manipulating code. |

| **Noncompliant Code** |
| --- |
| In this code the unsigned int would be able to result in a value in excess of the UCHAR\_MAX because of the way the integer would be |
| Static const char table[UCHAR\_MAX + 1] = { ‘a’ /\*...\*/ };  ptrdiff\_t first\_not\_in\_table(const char \*c\_str) {    for (const char \*s = c\_str; \*s; ++s) {      if (table[(unsigned int)\*s] != \*s) {        return s - c\_str;      }    }    return -1;  } |

| **Compliant Code** |
| --- |
| Compliant code would just be code that uses the right type of input information for each thing the code is trying to accomplish. |
| static const char table[UCHAR\_MAX + 1] = { 'a' /\* ... \*/ };    ptrdiff\_t first\_not\_in\_table(const char \*c\_str) {    for (const char \*s = c\_str; \*s; ++s) {      if (table[(unsigned char)\*s] != \*s) {        return s - c\_str;      }    }    return -1;  } |

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

|  |
| --- |
| **Principles(s):** Keep it Simple can be effective here because while the compliant code may not look simple to someone new to coding, it is consistent and consistency in what in input and the type of data is key. Validate Input Data is also effective here because of similar reasoning, as improper data could cause errors or make things seem suspicious. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | likely | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | -Wvarargs | Does not catch the violation in the third noncompliant code example (it is conditionally supported by [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions" \l "BB.Definitions-Clang)) |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2020a | [CERT C++: EXP58-CPP](https://www.mathworks.com/help/bugfinder/ref/certcexp58cpp.html) | Checks for incorrect data types for second argument of va\_start (rule fully covered) |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Values, or variables, go hand in hand with the data types, while the types are the different things that can be entered, the values are what gets entered. |

| **Noncompliant Code** |
| --- |
| The value entered would need to match up what is wanted for it to be compliant, you don’t want a character string like the name Charlie when your asking for a number! |
| Int mynum = “Charlie”;  cout << mynum; |

| **Compliant Code** |
| --- |
| Compliance here is just a case of values matching the inputs that they require, it’s hard to determine compliance and noncompliance for the basis of the values and inputs. |
| Int mynum;  mynum = 15;  cout << mynum; |

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

|  |
| --- |
| **Principles(s):** As this goes hand in hand with the previous standard, the principles that apply also match. (Copied from above: Keep it Simple can be effective here because while the compliant code may not look simple to someone new to coding, it is consistent and consistency in what in input and the type of data is key. Validate Input Data is also effective here because of similar reasoning, as improper data could cause errors or make things seem suspicious.) |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | likely | Low | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | return-implicit | Fully checked |
| CodeSonar | 6.0p0 | LAND.STRUCT.MRS | Missing a return statement |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Also seemed to be referred to const correctness, it serves as a way to keep strings from getting modified by things that should not change the process. The way strings are allocated, lack of const can allow strings to be changed freely and that could cause problems. |

| **Noncompliant Code** |
| --- |
| Noncompliant code here would have a tendency to lead to undefined behaviors where variables would not exist or have some other error occur, because the getenv string would return a null pointer. |
| Std::string tmp(std::getenv(“TMP”));  if (!tmp.empty()) {  //…  } |

| **Compliant Code** |
| --- |
| In this example the call is double checked for the null before the string object is actually constructed, making sure that the code will not have undefined behaviors because of a null pointer issue. |
| Const char \*tmpPtrVal = std::getenv(“TMP”;  std::string tmp(tmpPtrVal ? TmpPtrVal : “”);  if (!tmp.empty()) {  //…  } |

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

|  |
| --- |
| **Principles(s):** Architect&Design for Security Policies is viable here because string correctness creates less openings for attacking code strings, heeding compiler warnings works in tandem with this idea and this also means you are using effecting quality assurance by keeping string correctness and minimizing string vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | String-literal-modification  write-to-string-literal | Fully checked |
| Parasoft C/C++ test | 2021.1 | CERT\_CPP\_STR52-a | Use valid references pointers and iterators. |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Injections allow for data to be spoofed into input fields and compliance for this standard would be to combat these injection attacks. |

| **Noncompliant Code** |
| --- |
| This code does nothing to combat the attack where value=value, which is usually what is used because they will add “or value=value” to the code. |
| sql = "SELECT ID, NAME, PASSWORD FROM USERS WHERE NAME='Fred'"; |

| **Compliant Code** |
| --- |
| You would want, at the very least, regex searches in order to stop the or value=value attack to be usable in these code situations, this example is taken from my own code for the previous assignment. |
| int find\_match\_position(std::string s) {  std::smatch m;  std::regex e1("or");  std::regex\_search(s, m, e1);  std::cout << "Match for concerning phrase found, position information hidden.";  return m.position(0);  } |

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

|  |
| --- |
| **Principles(s):** Heeding compiler warnings here helps us to make sure that sql injections will fail, and this in turn helps us practice defense in depth by making the base sql more secure. Making sure sql injections will fail also applies to secure coding standards and security policies because it stops potential attacks. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | -- | SV.DATA.BOUND SV.DATA.DB SV.HTTP\_SPLIT SV.PATH SV.PATH.INJ SV.SQL | -- |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Free memory is what gets allocated dynamically when you are doing memory allocation, we are not supposed to directly access the freed memory and dynamically allocated memory should be freed when not in use, no other memory should be freed. |

| **Noncompliant Code** |
| --- |
| Two memory allocations are performed within the same expression here, and these allocations are passed as arguments to a function call, so an exception is thrown as a result of one calls to new, as it could result in memory leaks. |
| Struct A {/\* ... \*/};  struct B {/\* ... \*/};  voidg (A \*, B \*);  void f() {  g(new A, new B);  } |

| **Compliant Code** |
| --- |
| In order to take the above example, and make it compliant with how memory should be handled, you would either use references or make use of std::unique\_ptr in many cases, here we used object references. |
| Struct A { /\* ... \*/ };  struct B { /\* ... \*/ };    void g(A &a, B &b);  void f() {  A a;  B b;  g(a, b);  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings that so the code is properly handling memory usage and protection, which is also a secure coding standard that applies to standard policies because this means we are protecting memory in and out of use from attacks or vulnerabilities. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | leak |
| PC-lint Plus | 1.4 | 429 | Fully supported |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions handle errors by testing assumptions that are made by a programmer, and because of this are an extremely valuable diagnostic tool. |

| **Noncompliant Code** |
| --- |
| This code uses the macro assert() in order to assert a property concerning a memory mapped structure which is essential for the code to behave as intended. |
| 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 |
| --- |
| Static assertions allow for incorrect assumptions to be diagnosed at compile time instead of causing malfunctions or runtime errors, so using static\_assert is key for compliant code. |
| 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):** Default Deny is used here because assertions decide certain factors but access should be denied by default if assertions don’t state something otherwise, this is also good for secure coding standards and security policies because it allows for testing things more properly. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | (customization) | Custom checks can be implemented with assert macro |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Functions throw exceptions based on the exception specifications, and if an exception doesn’t match these specifications, it will cause std:unexpected to be called, which throws bad\_exception after and eventually run std::terminate. |

| **Noncompliant Code** |
| --- |
| This code is declared as nonthrowing but it is possible for vector::resize to throw exceptions when memory cannot be allocated. |
| void f(std::vector<int> &v, size\_t s) noexcept(true) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| This version of the code the noexcept chunk of the code is removed so the function will now allow all exceptions, which is okay for it to do. |
| void f(std::vector<int> &v, size\_t s) {    v.resize(s); // May throw, but that is okay  } |

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

|  |
| --- |
| **Principles(s):** Design for security policies and compiler warnings are both big here, as well as defense in depth, as they are all covering the idea of errors thrown during coding whether intentional or not intentional and what the errors could do to the code or even force the code to abort and require proper testing. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.1 | C++ 4075 or 4076 | -- |
| Parasoft C/C++ test | 2021.1 | CERT\_CPP\_ERR56-a | Ensures resources get freed |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Guarantee Exception Safety | STD-008-CPP | Proper error handling and exceptional situations is an essential part of software operation, and exceptions are better than error codes for reporting errors in C++, the strong exception safety guarantee is a property of an operation such that, in addition to satisfying the basic exception safety guarantee, if the operation terminates by raising an exception, it has no observable effects on program state. |

| **Noncompliant Code** |
| --- |
| This example shows a flawed copy assignment operator, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the impliciant invariants of the class. In the end it results in undefined behavior. |
| class IntArray {    int \*array;    std::size\_t nElems;  public:    // ...      ~IntArray() {      delete[] array;    }        IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {      if (this != &rhs) {        delete[] array;        array = nullptr;        nElems = rhs.nElems;        if (nElems) {          array = new int[nElems];          std::memcpy(array, rhs.array, nElems \* sizeof(\*array));        }      }      return \*this;    }      // ...  }; |

| **Compliant Code** |
| --- |
| In this code the same copy assignment operator has the exception safety guarantee, and it allocated new storage for the copy before changing the states of the object, after all this the function will be able to change the state and in addition, the function avoids the test for self-assignment which is an improvement to the performance. |
| class IntArray {    int \*array;    std::size\_t nElems;  public:    // ...      ~IntArray() {      delete[] array;    }      IntArray(const IntArray& that); // nontrivial copy constructor      IntArray& operator=(const IntArray &rhs) {      int \*tmp = nullptr;      if (rhs.nElems) {        tmp = new int[rhs.nElems];        std::memcpy(tmp, rhs.array, rhs.nElems \* sizeof(\*array));      }      delete[] array;      array = tmp;      nElems = rhs.nElems;      return \*this;    }      // ...  }; |

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

|  |
| --- |
| **Principles(s):** This also uses validation of input data and quality assurance as it is confirming exceptions and making sure that things are operating safely, so things that apply to exceptions also apply here. (Copied from Exceptions: Design for security policies and compiler warnings are both big here, as well as defense in depth, as they are all covering the idea of errors thrown during coding whether intentional or not intentional and what the errors could do to the code or even force the code to abort and require proper testing. ) |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.1 | C++ 4075 or 4076 | -- |
| Parasoft C/C++ test | 2021.1 | CERT\_CPP\_ERR56-a | Ensures resources get freed |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Freed Memory | STD-009-CPP | Previously touched on in memory protection, we are not supposed to directly access Freed memory, when memory is freed all pointers to it are invalid, but the data at the freed location can appear to be valid, and it must not be written to or read from while freed. |

| **Noncompliant Code** |
| --- |
| Here, s is dereferenced after it has been deallocated which can result in a write-after-free, which can be exploited to run arbitrary code with permissions of the now vulnerable process, these allocations and deallocations tend to be far removed which makes them difficult to recognize and diagnose. |
| struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| **Compliant Code** |
| --- |
| Here the dynamically allocated memory is not deallocated until it is no longer required in the code. (new and delete compliance example, automatic storage duration also would have a compliant result) |
| 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):** Heed Compiler warnings Adopt A secure Coding Standard, and Keep it Simple, and Design for Security Policies These all apply because using freed memory is something that should not be done as all memory needs to be initialized and used properly and then freed after use, freed memory is memory that was used and should no longer be accessed. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.UAF | Use after free |
| TrustInSoft Analyzer | 1.38 | Dangling\_pointer | Exhaustively verified |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Un-Initialized  Memory | STD-010-CPP | Just like freed memory, uninitialized memory, while not exactly freed memory is memory that you should not be reading from, typically this memory has a value of some sort attached to it, essentially this just means you want to make sure everything is initialized prior to use. |

| **Noncompliant Code** |
| --- |
| Here we have undefined behavior because an uninitialized local variable is evaluated as part of an expression to print its value. |
| void f() {    int i;    std::cout << i;  } |

| **Compliant Code** |
| --- |
| To fix the above example all we needed to do was initialize the variable by setting it to 0 |
| 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):** Heed Compiler warnings Adopt A secure Coding Standard, and Keep it Simple, and Design for Security Policies  Memory that has not been properly initialized is like using freed memory, which we previously discussed is something that should not be done as this creates potential gaps and problems in codes that could be exploited. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Low-Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | leak |
| PC-lint Plus | 1.4 | 429 | Fully supported |

## 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.

[When verifying and testing, many of these automation actions I have listed in the standards could be used to confirm everything is working intentionally, but ideally these are things you would want to test in the designing and building stages, as you build sections of code, you should be testing and confirming that they are following the standards you have set out for yourself and your team, or that is set by what is needed of everyone. In the production side of the phase, you should always be regularly testing and re-testing your code to make sure no new flaws have been detected by updated versions of automation testing, and use new knowledge based on successful or failed attacks to further improve the code.]

## 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 | Medium | likely | Low | High | 2 |
| STD-002-CPP | Medium | likely | Low | High | 2 |
| STD-003-CPP | High | likely | Medium | High | 3 |
| STD-004-CPP | High | likely | Medium | High | 3 |
| STD-005-CPP | High | likely | Low | High | 3 |
| STD-006-CPP | Medium | likely | Low | Medium | 1 |
| STD-007-CPP | Medium | likely | Low | Medium | 1 |
| STD-008-CPP | Medium | likely | Low | Medium | 2 |
| STD-009-CPP | High | likely | Medium | High | 5 |
| STD-010-CPP | Medium | likely | Low/Medium | High | 4 |

## 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 | This is designed to prevent attackers from accessing unecrypted data by ensuring that the data is encrypted when on disk. An attacker may obtain the drive that has the ecrypted data, but they wont have the keys and will then have to defeat the encryption in order to get into the data. This encryption is high priority and highly recommended for this reason because this makes the attack required a much more complex required. |
| Encryption at flight | Encrypting data while the data is being transmitted is a key factor as well because when you are transmitting data between applications or other software, it may not be initially encrypted and this means that it would be vulnerable to attacks during transit, and this is to stop that from happening by providing protection to the transmitted information at both ends of the transmission. |
| Encryption in use | To ensure that data is never left unsecured, in use encryption is applied to the process for all lifecycle stages, that way the data is always protected in some way when potential attacks are in progress, this reduces the amount of vulnerabilities, especially if different stages encrypt differently or there are multiple layers of encryption. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Get what the user knows by getting things like their username and password, and their security questions. Or have other means of identification when possible like fingerprinting or other biometric tests to authenticate people, or use what they have as an authentication, access cards are a good example of this, and 2 factor auth via phones is also a good example, these each come with their own flaws. In our case we are using user logins and the security measures like security questions. |
| Authorization | Based on the user level of access, we can say that Role-Based Access Control is highly used, which is where access is determined by the role within an organization, and every user login has this information attached to it in some respect, which allows of the staff to access the appropriate parts of the system they require for their job, without being able to change or interfere with other parts of the system, only allowing them to access and modify the appropriate files, while locking them out of other files based on their access level. MAC (Mandatory Access Control) works in tandem with this to make sure only trusted individuals within these user levels can access more delicate information. |
| Accounting | These systems are monitored closely, and it includes monitoring when accounts are accessed and what the account is used for, most of the time they are monitored by some sort of security information and event management tool that the servers use. Knowing what is being access and by who at all times can inform whether more or less authorizations are required and it can help discover suspicious activity. |

**\***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 | 05/23/2021 | Coding Standard Submit | Jacob Mousseau |  |
| 1.1 | 05/23/2021 | Project Submit | Jacob Mousseau | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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