

# Green Pace Secure Development Policy

## Contents

[Overview 2](#_hibw4kc2x4d7)

[Purpose 2](#_kyrhjb9c70mu)

[Scope 2](#_14s6kofzam4w)

[Module Three Milestone 2](#_7fou5plfn792)

[Ten Core Security Principles 2](#_59ag6h90o65r)

[C/C++ Ten Coding Standards 3](#_v8e46o32316j)

[Coding Standard 1 4](#_hleznuxw2qnn)

[Coding Standard 2 5](#_b7ji0ktqxpxn)

[Coding Standard 3 6](#_sp9kfdd3yid0)

[Coding Standard 4 7](#_dackwflswdd4)

[Coding Standard 5 8](#_rn4hgcfahu7k)

[Coding Standard 6 9](#_ruroltxd3481)

[Coding Standard 7 10](#_k1eb4qan09ln)

[Coding Standard 8 11](#_7ribpymdr5uc)

[Coding Standard 9 13](#_2oombf5vficd)

[Coding Standard 10 14](#_od91eks5x8rs)

[Defense-in-Depth Illustration 15](#_t2jdw9kfktos)

[Project One 15](#_w4ihq1kkcccn)

[1.](#_jmf7nrixdkzz) Revise the C/C++ Standards 15

[2.](#_2ram8evsnxuc) Risk Assessment 15

[3.](#_s0wkaedic24z) Automated Detection 15

[4.](#_oxpeaaym2xbv) Automation 15

[5.](#_q0mwhy3vty3k) Summary of Risk Assessments 16

[6.](#_hdxyhc32h68c) Create Policies for Encryption and Triple A 16

[7.](#_us9p89bkeamg) Map the Principles 17

[Audit Controls and Management 18](#_h5t18jzeeggb)

[Enforcement 18](#_wk7xoocsr42)

[Exceptions Process 18](#_yja0h0ov838m)

[Distribution 19](#_fww9imjfv4xs)

[Policy Change Control 19](#_upd4rkdbltvy)

[Policy Version History 19](#_qnvg6lrv7dx)

[Appendix A Lookups 19](#_qzvgeucubkdo)

[Approved C/C++ Language Acronyms 19](#_mbeitjzi6c0x)

## 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 | Input validation must be performed to ensure that only data that is properly formatted is entering the software stack to prevent malicious activity of downstream components. This step should be performed before any input data is used and should be tested at syntactic as well as semantic levels. This prevents injection attacks, buffer overflows and various other malicious inputs. |
| 1. Heed Compiler Warnings | Ensure all warnings are produced by the compiler and fix these warnings through modifying the code base instead of ignoring warnings. Warnings signal a potentially serious problem and some may be fatal; others may or may not be, but you want to detect false positives and fix them before public release. This reduces the risk of undefined behavior and logic errors they could be exploited |
| 1. Architect and Design for Security Policies | Enforce security policies for the software stack. If there are areas in which a portion of the stack requires a different level of privileges, then ensure you split the systems up with a different privilege requirement for each component. This prevents unauthorized access and privilege escalation. |
| 1. Keep It Simple | Do not overcomplicate a system, the more complex things are, the higher the risk is that you will encounter errors. Not only this but debugging warnings/errors/vulnerabilities becomes ever more difficult. Lowers the risk of hidden bugs and vulnerabilities that arise from over complex architecture. |
| 1. Default Deny | Do not exclude specific privilege levels, simply deny all requests by default and allow the privilege system to request permission. If the request is not specifically allowed, it is denied. It prevents unauthorized access. |
| 1. Adhere to the Principle of Least Privilege | Only provide the least set of privileges required for a specific system component. Furthermore, only allow extra privileges for the duration it is needed. If an account is compromised, it limits the damage that may occur by providing the least amount of privilege needed for a task and the amount of time needed. |
| 1. Sanitize Data Sent to Other Systems | Like validating input data, you should ensure all data being passed to another system is cleansed before sending to external systems. Mitigate attacks from possible attacks like SQL Injection, buffer overflow, etc.. |
| 1. Practice Defense in Depth | Do not rely on one layer of defense, if one layer is not strong enough, another layer may prevent the vulnerability of the previous layer from spreading through to the system. |
| 1. Use Effective Quality Assurance Techniques | Implement testing for software stacks and source code audits. Hire security testing to test for zero-day exploits and other vulnerabilities. This prevents flaws before deployment, reducing the risk of exploitation in production. |
| 1. Adopt a Secure Coding Standard | Use a secure coding standard that relates to the code base language and platform(s). This will prevent recurring vulnerabilities and ensure consistent applications. |

### C/C++ Ten Coding Standards

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | DCL60-CPP Obey the One-Definition Rule |

| **Noncompliant Code** |
| --- |
| Defining a class of the same name with different definitions with two different translation units. |
| // a.cpp  struct S {  int a;  };  //b.cpp  class S {  public:  int a;  }; |

| **Compliant Code** |
| --- |
| If you must use the same class definition in multiple translation units, use the header file to introduce the translation unit(s). |
| // S.h  struct S {  int a;  };  // a.cpp  #include “S.h”  // b.cpp  #include “S.h” |

| **Principles(s):**  4. **Keep is simple** - By defining the same name with more than one definition, it will complicate the system which can lead to hidden bugs even if no warnings or errors arise. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | misc-definitions-in-headers | This check will catch non-external, non-inline function and variable definitions which can lead to a One Definition Rule violation.  <https://clang.llvm.org/extra/clang-tidy/checks/misc/definitions-in-headers.html> |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use valid references, pointers, and iterators to reference elements of a basic\_string. This shall cover safe handling of string data, which is critical for validating and managing data values. |

| **Noncompliant Code** |
| --- |
| Copying `input` into a `std::string` replacing (;) with spaces. The iterator `loc` is invalidated after the first call to `insert()` making subsequent calls to `insert()` undefined. |
| #include <string>  void f(const std::string &input) {  std::string email;  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++I; ++loc) {  email.insert(loc, \*i != ‘;’ ? \*i : ‘ ‘);  }  } |

| **Compliant Code** |
| --- |
| Instead, updating the value of `loc` as a result of each call to `insert()` ensures the invalidated iterator will not be accessed. |
| #include <string>  void f(const std::string &input) {  std::string email;  std::string::iterator loc = email.begin();  for (auto i = input.begin(), e = input.end(); i != e; ++I; ++loc) {  loc = email.insert(loc, \*i != ‘;’ ? \*i : ‘ ‘);  }  } |

| **Principles(s):**  1. **Validate Input Data** - Ensures correct handling of strings to prevent data corruption.  9. **Use Effective Quality Assurance Techniques** - Good testing will catch invalid references or iterators early. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| GCC/libstdc++ | 15.2.0 | -D\_GLIBCXX\_DEBUG | This will check containers and asserts to see if you used invalid iterators, pointers or references.  <https://gcc.gnu.org/onlinedocs/libstdc++/manual/debug_mode_using.html> |
| UndefinedBehaviorSanitizer | 22.0.0 | -fsanitize=undefined | UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution  <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html> |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Do not attempt to create a std::string from a null pointer. |

| **Noncompliant Code** |
| --- |
| Calling std::getenv creates a std::string object but because getenv() returns a null pointer on failure, it will lead to undefined behavior. |
| #include <csdtdlib>  #include <string>  void f() {  std::string tmp(std::getenv(“TMP”));  if (!tmp.empty()) {  // …  }  } |

| **Compliant Code** |
| --- |
| In a compliant solution, results are checked for a null before an object gets constructed. |
| #include <csdtdlib>  #include <string>  void f() {  const char \*tmpPtrVal = std::getenv(“TMP”);  std::string tmp(tmpPtrVal ? tmpPtrVal : “”);  if (!tmp.empty()) {  // …  }  } |

| **Principles(s):**  1. **Validate Input Data** - This will prevent null pointer use when creating strings.  10. **Adopt a Secure Coding Practice** - This is to ensure consistent handling of data across the system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | bugprone-stringview-nullptr | Checks for various ways that the const CharT\* constructor of std::basic\_string\_view can be passed a null argument and replaces them with the default constructor in most cases.  <https://clang.llvm.org/extra/clang-tidy/checks/bugprone/stringview-nullptr.html> |
| Polyspace Bug Finder | R2025b | STR51-CPP | Provides a dedicated check for ‘String operations on null pointer.’ |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection |

| **Noncompliant Code** |
| --- |
| Without sanitizing input arguments such as ‘username’ before feeding into a system, you permit SQL injection attacks. Here, the doPrivilegedAction() method uses a PreparedStatement instead of java.sql.Statement. |
| **public** **void** doPrivilegedAction(      String username, **char**[] password    ) **throws** SQLException {      Connection connection = getConnection();  **if** (connection == **null**) {        // Handle error      }  **try** {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString); |

| **Compliant Code** |
| --- |
| Instead, using a parametric query with a ? character as a placeholder. Then validate the username length to ensure the input data is sanitized. |
| **try** {      String pwd = hashPassword(password);        // Validate username length  **if** (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString); |

| **Principles(s):**  1. **Validate Input Data** - Not validating input allows for injection which may run malicious code.  7. **Sanitize Data Sent to Other Systems** - Ensures that only clean data is sent to SQL queries. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klokwork | 2025.3 | CXX.SQL.INJECT | This checker flags SQL injection vulnerabilities.  <https://help.klocwork.com/current/en-us/concepts/whatsnewmain.htm> |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory |

| **Noncompliant Code** |
| --- |
| S is dereferenced after it has been deallocated. This can be exploited to run arbitrary code with the permissions of the process. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| In this solution, dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

| **Principles(s):**  10. **Adopt a Secure Coding Standard** - A good standard will ensure basic issues don’t arise, such as deallocating accessed memory. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AddressSanitizer | 22.0.0 | -fsanitize=address | AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. It can check for Use-after-free  <https://clang.llvm.org/docs/AddressSanitizer.html> |
| Klokwork | 2025.3 | UFM.USE.MIGHT | The UFM checkers find instances of various use-after-free and double-free memory situations in code. The UFM.USE.MIGHT checker flags situations in which already freed memory may be used.  <https://help.klocwork.com/2024/en-us/reference/ufm.use.might.htm> |
| CPPCheck | 2.18 | Enable warning/performance | CPPCheck will flag use after free and double freed memory.  <https://cppcheck.sourceforge.io/> |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Incorporate diagnostic tests using assertions. |

| **Noncompliant Code** |
| --- |
| Using the assert() macro to verify memory allocation succeeds is noncompliant because memory availability depends on the overall state of the system and can become exhausted at any point. Using the assert() macro because it may lead to an abrupt termination of the process, creating a possibility of a DOS attack. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char** \*)**malloc**(len + 1);  **assert**(NULL != dup);    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

| **Compliant Code** |
| --- |
| Instead, detect and handle a possible memory exhaustion by checking if the memory is NULL. |
| **char** \*dupstring(**const** **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char**\*)**malloc**(len + 1);    /\* Detect and handle memory allocation error \*/  **if** (NULL == dup) {  **return** NULL;    }    **memcpy**(dup, c\_str, len + 1);  **return** dup;  } |

| **Principles(s):**  9. **Use Effective Quality Assurance Technique** - Runtime checks should always have proper and appropriately named assertions.  8. **Practice Defense in Depth** - By having quality diagnostic tests it will prevent deeper system failures / manipulation. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CPPCheck | 2.18 | Enable Style and Warnings | This won’t detect the dedicated presence of asserts, but will check for misuse of assert.  https://cppcheck.sourceforge.io/ |
| Clang-Tidy | 20.1.0 | misc-assert-side-effect | Checks for asserts that cause an undefined behavior.  <https://releases.llvm.org/4.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/misc-assert-side-effect.html> |
| CodeQL | 2.23.2 | semmle.code.cpp.commons.Assertions | This would require custom queries catered to finding asserts and creating custom policies.  <https://codeql.github.com/> |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle all exceptions |

| **Noncompliant Code** |
| --- |
| In this example, neither f() or main() catch exceptions thrown by throwing\_func(). Thus std::terminate() is called. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| Instead, the main entry point handles all exceptions which ensures graceful error management. |
| **void** throwing\_func() noexcept(**false**);    **void** f() {    throwing\_func();  }    **int** main() {  **try** {      f();    } **catch** (...) {      // Handle error    }  } |

| **Principles(s):**  3. **Architect and Design for Security Policies** - Appropriately designs security policies will catch all exceptions and have a default exception for those that are undefined.  9. **Use Effective Quality Assurance Techniques** - Quality assurance techniques ensure reliable error handling and expected behavior. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | bugprone-exception-escape | Finds functions which may throw an exception directly or indirectly, but they should not.  <https://clang.llvm.org/extra/clang-tidy/checks/bugprone/exception-escape.html> |
| CPPcheck | 2.18 | Enable Warnings | Includes dedicated logic for exception safety. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Free Resources | [STD-008-CPP] | Properly deallocate dynamically allocated resources |

| **Noncompliant Code** |
| --- |
| The ‘space’ variable is passed as the expression to the ‘new’ operator. The pointer to that call is then passed to operator\_delete() resulting in attempting to free memory that is not returned by operator\_new(). |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...    **delete** s1;  } |

| **Compliant Code** |
| --- |
| Instead, remove the call to operator\_delete() and explicitly call the destructor. In most cases, invoking the destructor is not warranted. |
| #include <iostream>    **struct** S {    S() { std::cout << "S::S()" << std::endl; }    ~S() { std::cout << "S::~S()" << std::endl; }  };    **void** f() {    alignas(**struct** S) **char** space[**sizeof**(**struct** S)];    S \*s1 = **new** (&space) S;      // ...      s1->~S();  } |

| **Principles(s): 10. Adopt a Secure Coding Standard -** Similarly to deallocating accessed memory, a good standard would prevent memory from not being deallocated when no longer in use. If memory is never deallocated, you put it at risk of manipulation and memory leaks. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 1.1.2.4 | clang-analyzer-cplusplus.NewDelete | Check for double-free and use-after-free problems. Traces memory managed by new/delete.  https://clang.llvm.org/docs/analyzer/checkers.html#cplusplus-newdelete |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Handle Proper Returns | [STD-009-CPP] | Do not return from a function declared [[noreturn]] |

| **Noncompliant Code** |
| --- |
| In this if a value of 0 is passed, control will flow off the end of the function resulting in an implicit return. |
| #include <cstdlib>    [[**noreturn**]] **void** f(**int** i) {  **if** (i > 0)  **throw** "Received positive input";  **else** **if** (i < 0)      std::**exit**(0);  } |

| **Compliant Code** |
| --- |
| Instead, do not return on any code path. |
| #include <cstdlib>    [[**noreturn**]] **void** f(**int** i) {  **if** (i > 0)  **throw** "Received positive input";    std::**exit**(0);  } |

| **Principles(s):** 10. **Adopt a Secure Coding Standard** - It is standard to not return when declared noreturn, similarly to [[noexecpt]] should not throw. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 20.1.0 | -Wunreachable-code-return | Detects if you are attempting to return in a function that is labeled as noreturn. |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Avoid Race Conditions | [STD-010-CPP] | Avoid race conditions while checking for the existence of a symbolic link |

| **Noncompliant Code** |
| --- |
| The lstat() function collects information about a symbolic link instead of its target. This means that there is a time-of-check and time-of-use race condition between the call to lstat() and next calls to open() because both functions operate on a file name that can be manipulated asynchronously. |
| **char** \*filename = /\* file name \*/;  **char** \*userbuf = /\* user data \*/;  unsigned **int** userlen = /\* length of userbuf string \*/;    **struct** stat lstat\_info;  **int** fd;  /\* ... \*/  **if** (lstat(filename, &lstat\_info) == -1) {    /\* Handle error \*/  }    **if** (!S\_ISLNK(lstat\_info.st\_mode)) {     fd = open(filename, O\_RDWR);  **if** (fd == -1) {         /\* Handle error \*/     }  }  **if** (write(fd, userbuf, userlen) < userlen) {    /\* Handle error \*/  } |

| **Compliant Code** |
| --- |
| By using O\_NOFOLLOW to cause open() to fail if passed a symbolic link to avoid TOCTOU by not having a check and use call to avoid a race condition. |
| **char** \*filename = /\* file name \*/;  **char** \*userbuf = /\* user data \*/;  unsigned **int** userlen = /\* length of userbuf string \*/;    **int** fd = open(filename, O\_RDWR|O\_NOFOLLOW);  **if** (fd == -1) {    /\* Handle error \*/  }  **if** (write(fd, userbuf, userlen) < userlen) {    /\* Handle error \*/  } |

| **Principles(s):** 3. **Architect and Design for Security Policies** - This promotes secure file access design and prevents race conditions that could be exploited. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang-Tidy | 1.1.7.15 | security.insecureAPI.DeprecatedOrUnsafeBufferHandling | Warn on occurrences of unsafe or deprecated buffer handling functions.  <https://clang.llvm.org/docs/analyzer/checkers.html#security-insecureapi-deprecatedorunsafebufferhandling> |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



### Automation



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Throughout the entire DevSecOps we should implement dynamic and live automated testing during each stage to ensure proper security. During the initial stage of assessment and planning, we should integrate policy standards into threat modeling. We can use “ThreatModeler” to incorporate automated checks for attack vectors. When moving to the design state we should integrate OWASP and an internal policy checklist into design reviews and other static analysis tools (mentioned previously). During the build stage we should be dynamically testing using dependency scanners to prevent easily preventable attacks. Furthermore, when verifying and testing we should implement vulnerability scanners such as Clang-Tidy (others mentioned above) and further extend testing to include security and compliance validation. When moving onto the transition stage, we should implement a validation before deployment. While we have done verification and testing, we can use an infrastructure-as-code scanner to ensure secure deployment settings. During the monitor and detection stage, implementing custom policy-based alerting will be helpful to detect any suspicious concerns. These additions/modifications will ensure a properly strong developing operation security cycle; it should be treated as a continuous process and use the aforementioned automation tools to ensure compliance at each stage.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Medium | Unlikely | Low | Medium | 2 |
| STD-002-CPP | High | Likely | Medium | High | 4 |
| STD-003-CPP | Medium | Possible | Low | Medium | 3 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | Medium | Possible | Low | High | 2 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Likely | Medium | High | 4 |
| STD-008-CPP | High | Likely | Medium | High | 4 |
| STD-009-CPP | Low | Unlikely | Low | Low | 1 |
| STD-010-CPP | High | Possible | High | High | 5 |

### 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** | **Encryption refers to the act of ensuring any data in any form is safe from external individuals in-case of unauthorized access. This means, if the data is stored on a hard-drive, it is encrypted; if data is being transferred through cloud, it is encrypted; lastly if data is being used, it is only being used by a user that has decryption for the data.** |
| --- | --- |
| Encryption at rest | Protecting data that is physically stored on hard drives / solid-state drives, including backups and databases.  Transparent Data Encryption (TDE) may be used to encrypt SQL and Azure SQL database data files at rest. Only authorized users should have access to the keystore that can perform the I/O decryption of the data. All user logins shall be logged and all encryption and decryption actions shall be logged, this also includes changes to the database. When protecting a physical drive, BitLocker Drive Encryption may be used to protect the file volume; when online the drive is never decrypted, only unlocked.  These steps will prevent gaining access to data if a physical drive is stolen. |
| Encryption in flight | All data that is transferred from one machine to another, being sent across networks or moved through cloud infrastructure.  If doing file transfers you shall use ‘scp’ or ‘sftp’ so that it is encrypted. Furthermore, ‘rsync’ may be used for synchronous file transfers over SSH. When on Windows, Remote Desktop is a good encryption tool for file transfers to hosts. Finally, self-signed certificates shall be avoided and only Certificate Authorities shall be used so that public keys may be verified by a trusted third party with automatic renewal. When sending to external networks, a VPN shall be used and all network communication shall be protected by TLS, specifically v1.2 or higher. When using cipher suites, only strong ciphers shall be used (such as AES).  These steps will prevent interceptions, tampering and man-in-the middle attacks. |
| Encryption in use | In use data refers to data that is being processed in memory, this should still be encrypted in-case an attacker is able to obtain this information. This may also be referred to as in-memory encryption or runtime encryption.  We may use partially homomorphic encryption (PHE) which involves only the use of one type of mathematical operation on encrypted data or fully homomorphic encryption (FHE) which allows both mathematical operations on encrypted data. Both cases addressed in-use data vulnerabilities by allowing computations to run directly on encrypted data without the need of decrypting the data. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication enables the need for login credentials to ensure the party is who they claim they are. This would involve the requirement of a username and password and for best security a multi-factor authentication system (biometrics, 2FA key). When a user logins, the password is encrypted through hashing, then protected in transit by an encrypted channel and authentication server. The database changes as a new user is added, and this data shall be encrypted as well. Each user is provided a default level of access to files with the lowest possible permissions as needed and defined by their level of access. |
| Authorization | The authorization step is where privileges are granted to the party if the authentication server accepts the provided data. This is where the user level of access is defined based on the database permission in reference to the user’s identity. An admin can access and make changes to this portion of the database to ensure only the correct files and level of security are provided to the authorized user. |
| Accounting | In the step of accounting, all user login, file transfer, user activity and attempts (even when denied) are logged. This information includes IP addresses, what data was transmitted or received (this shall be encrypted and only available to the correct level of access) and their URI. |

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

## 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 | 09/17/2025 | Module 3 Milestone Start | Ronny Valtonen | Ronny Valtonen |
| 1.2 | 10/06/2025 | Project Start | Ronny Valtonen | Ronny Valtonen |
| 1.3 | 10/09/2025 | Policies for Encrypt and Triple-A | Ronny Valtonen | Ronny Valtonen |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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