**Green Pace Developer: Security Policy Guide**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

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

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

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Seven Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | All incoming data must undergo rigorous validation to ensure it adheres to expected formats, ranges, and types. Failing to validate input opens the door to various security vulnerabilities, including injection attacks like SQL injection and cross-site scripting (XSS), as well as buffer overflows. |
| 1. Heed Compiler Warnings | Compiler warnings serve as early indicators of potential weaknesses within the codebase. Ignoring these warnings may result in exploitable vulnerabilities. It's imperative to address and resolve all compiler warnings promptly to maintain a secure software environment. |
| 1. Architect and Design for Security Policies | Security considerations should be an integral part of software design and architecture. This involves identifying potential risks, establishing security policies, and implementing appropriate security measures at every level of the application's architecture. |
| 1. Keep It Simple | Striving for simplicity in design and implementation helps reduce the attack surface and facilitates the identification and mitigation of security vulnerabilities. Complex systems are inherently more difficult to secure and often harbor hidden flaws exploitable by attackers. |
| 1. Default Deny | Adopting a default deny stance involves restricting access to resources by default and only granting access to explicitly authorized entities. This approach minimizes the risk of unauthorized access and mitigates the impact of security breaches. |
| 1. Adhere to the Principle of Least Privilege | Users, processes, and systems should be granted only the minimum level of access necessary to fulfill their intended functions. By limiting privileges, organizations can mitigate the potential damage caused by compromised or malicious entities. |
| 1. Sanitize Data Sent to Other Systems | Any data transmitted to external systems, such as databases or APIs, must undergo rigorous sanitization to prevent injection attacks and uphold data integrity. Sanitization involves validating, cleaning, and escaping user input to eliminate potentially harmful characters or code. |
| 1. Practice Defense in Depth | Implementing multiple layers of security controls, including network security, host-based security, application security, and user authentication mechanisms, helps safeguard against a broad spectrum of threats. The defense-in-depth approach provides comprehensive protection for systems and data. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance processes, such as code reviews, static analysis, and penetration testing, play a crucial role in identifying and rectifying security vulnerabilities. Regular QA activities throughout the software development lifecycle are essential to ensure a secure codebase. |
| 1. Adopt a Secure Coding Standard | Following a secure coding standard establishes consistency and adherence to best practices in software development. These standards define rules and guidelines for writing secure code, helping to prevent common programming errors that lead to security vulnerabilities. |

### 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] | It's essential to properly manage the range of integer data types to prevent wrapping (integer overflow) during conversions between signed and unsigned integers at their maximum and minimum values. For instance, INT30-C mandates that integer values should not wrap when used in pointer arithmetic, such as array indexing, as a length or size of an object, as the bound of array access, in function arguments of type size\_t or rsize\_t, or in security-critical code. Truncation errors occur when converting a value from an unsigned integer type to a narrower width, resulting in data loss if the value cannot be represented in the new type. |

| **Noncompliant Code** |
| --- |
| Truncation may occur when a value is too small to adequately represent the result, and conversions can lead to values exceeding the range of the resulting type. |
| unsigned long int ul = ULONG\_MAX;  signed char sc;  sc = (signed char)ul; |

| **Compliant Code** |
| --- |
| Ensure that the range is valid when converting from an unsigned type to a signed type. |
| unsigned long int ul = ULONG\_MAX;  signed char sc;  if (ul <= SCHAR\_MAX) {  sc = (signed char)ul;  } else {  // Handle error condition appropriately  } |

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

| **Principles(s): 1 –** Validate Input Data: Casting to an unspecified result does not allow the input data to  validated to give or revoke access to a system. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2 | CertC++-INT50 |  |
| Parasfot C/C++test | 2021.2 | CERT\_CPP-INT50-a | Ensure that an expression with an underlying enum type exclusively contains values that align with the enumerators defined within the enumeration. |
| PVS-Studio | 7.15 | V1016 |  |
| PRQA QA-C++ | 4.4 | 3013 |  |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | When choosing a data type, remember that unsigned integers should be used for values that can't be negative, while signed integers should be used for values that can be negative. It's a good idea to pick the smallest data type that can fully represent the range of possible values for a variable, as this saves memory. For example, using 'short' for a variable may allow negative sizes and could have limited range, whereas using 'size\_t' provides more precision and ensures the variable can hold larger values. This advice aligns with the rule 'Ensure that operations on signed integers do not result in overflow'. |

| **Noncompliant Code** |
| --- |
| This code overlooks the fact that the unsigned integer value will wrap around, leading to an infinite loop. |
| char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  size\_t cnt = /\* initialize \*/;  for (unsigned int i = cnt-2; i >= 0; i--) {  a[i] += a[i+1];  } |

| **Compliant Code** |
| --- |
| Since size\_t is an unsigned type, the standard defines its behavior to be modulo arithmetic. |
| char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  size\_t cnt = /\* initialize \*/;  for (size\_t i = cnt-2; i != SIZE\_MAX; i--) {  a[i] += a[i+1];  } |

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

| **Principles(s): 5 –** Default Deny: Firstly, denies access to data inside a container unless valid references or  pointers are related to the data inside the container. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++4746, C++4747, C++4748, C++4749 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-CTR51-a | Doesn’t modify the container while iterating |
| PVS-Studio | 7.12 | V783 |  |
| Astree | 20.10 | overflow-upon-dereference |  |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Incorrectly determining string sizes and neglecting buffer boundaries can result in buffer overflows and runtime errors. It's crucial to avoid copying data from an unbounded source like stdin into a fixed-length array. For string literals, adhere to the following rules: STR30-C, which prohibits attempting to modify string literals, and STR31-C, which mandates ensuring that storage for strings has sufficient space for character data and the null terminator.  Additionally, be cautious with UTF decoders, as they can pose a security risk if an attacker sends an octet sequence not permitted by UTF-8 syntax. Follow the guidance outlined in MSC10-C regarding character encoding and UTF-8-related issues.  A robust and secure approach to string handling is the "callee allocates, callee frees" model from the C++ standard class template std::basic\_string. In this model, the callee is responsible for both allocating and freeing storage, and null terminators are not required, reducing the likelihood of errors. This model is considered the most secure and should be followed when working with C++. Avoid using the deprecated gets() function. |

| **Noncompliant Code** |
| --- |
| C++: If a user inputs more than 11 characters, it will result in an out-of-bounds write |
| #include <iostream>  int main(void) {  char buf[12];  std::cin >> buf;  std::cout << "echo: " << buf << '\n';  } |

| **Compliant Code** |
| --- |
| The overflow in the previous example is prevented by setting the field width member to the size of the character array buf. |
| #include <iostream>  int main(void) {  char buf[12];  std::cin.width(12);  std::cin >> buf;  std::cout << "echo: " << buf << '\n';  }  string str;  string::iterator i;  for (i = str.begin(); i != str.end(); ++i) {  cout << \*i  } |

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

| **Principles(s): 1 –** Architect and Design for Security Policies: Prevents any database architect issues by making  sure sufficient storage is available. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++2835, C++2836, C++2839, C++5216 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-STR50-b  CERT\_CPP-STR50-c  CERT\_CPP-STR50-e  CERT\_CPP-STR50-f  CERT\_CPP-STR50-g | Prevent overflow risks by avoiding the processing of non-null-terminated strings. Mitigate overflow possibilities during buffer writes. Safeguard against buffer overflows stemming from tainted data. Exercise caution to prevent buffer write overflows resulting from tainted data. Refrain from utilizing the 'c' character. |
| Klocwork | 2021.3 | NNTS.MIGHT  NNTS.TAINTED |  |
| SonarQube C/C++ Plugin | 4.1 | S3519 |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Attack vectors for strings encompass command-line arguments, environmental variables, console input, text files, and network connections. These vectors provide opportunities for attackers to execute strategic SQL injection-based attacks, potentially causing overflow. As per Secure Coding in C and C++, "String concatenation is the primary point of entry for script injection." To mitigate risks, it's essential to validate the size and data type of input and impose appropriate limits. Reject entries containing binary data, escape sequences, and comment characters. Review all code calling EXECUTE, EXEC, or sp\_executesql, and consistently utilize Parameterized Queries in SQL.  In C++, the operator >> extracts characters and stores them in successive elements of the array pointed to by str, halting extraction upon encountering valid whitespace, a null character, or EOF. To prevent buffer overflow, restrict the extraction operation to a specified number of characters by setting the field width (ios\_base::width or setw()) to a value greater than 0.  Related SEI Rule: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| Unfiltered code is vulnerable to SQL injection through user input |
| SqlDataAdapter myCommand =  new SqlDataAdapter("LoginStoredProcedure '" +  Login.Text + "'", conn); |

| **Compliant Code** |
| --- |
| Employing the Parameters collection with Dynamic SQL |
| SqlDataAdapter myCommand = new SqlDataAdapter(  "SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id", conn);  SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",  SqlDbType.VarChar, 11);  Parm.Value = Login.Text; |

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

| **Principles(s):** 3 – Architect and Design for Security Policies: Prevents issues from any stored variables being replaced and destroying later on in the code. This issue also relinquishes ownership of the managed pointer value. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++4721, C++4722, C++4723 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MEM56-a | Doesn’t modify the container while iterating |
| PVS-Studio | 7.12 | V1006 |  |
| Astree | 20.10 | dangling\_pointer\_use |  |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | A stack buffer overflow occurs due to stack smashing, where a security vulnerability is exploited, and the stack buffer is flooded with data supplied by an untrusted user. This manipulation corrupts the stack, allowing code injection into the program at runtime, enabling the user to seize control. Moreover, dereferencing null or invalid pointers can result in undefined behavior and segmentation faults. Pointer subterfuge exploits may also overwrite function pointers to redirect control to attacker-supplied shellcode. Mitigation strategies include eliminating buffer overflow scenarios and encrypting pointers. Adherence to CERT Standard for C++ security guidelines is crucial, encompassing rules such as MEM50-CPP (avoiding access to freed memory), EXP53-CPP (preventing reading uninitialized memory), MEM52-CPP (detecting and handling memory allocation errors), and MEM31-C (freeing dynamically allocated memory when no longer needed). The latter rule suggests catching a std::bad\_alloc exception if adequate memory allocation fails. Furthermore, compliance with the CERT C++ Coding Standard (SEI 2012b), specifically MEM08-CPP (using new and delete instead of raw memory allocation and deallocation), helps prevent subtle memory errors from emerging. |

| **Noncompliant Code** |
| --- |
| The vulnerable code demonstrates dereferencing the pointer 's' after it has been deallocated. If this leads to a "write after free" scenario, the vulnerability can be exploited to execute arbitrary code with the permissions of the vulnerable process. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The solution is to deallocate the memory only when it is no longer needed. |
| #include <new>    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):** 8 – Practice Defense in Depth: This can cause a null-pointer or pointer behavior to be undefined and give results previous array new-expression to a command operator (new). |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++2110, C++2111, C++2112, C++2113, C+ +2118, C++3337, C++3339, C++4262, C+ +4263, C++4264 |  |
| Parasoft C/C++test | 2021.2 | CERT\_CPP-MEM51-a  CERT\_CPP-MEM51-b CERT\_CPP-MEM51-c  CERT\_CPP-MEM51-d | Maintain consistency by employing matching forms in calls to new/malloc and delete/free. Always include empty brackets ([]) when deallocating arrays with delete. Ensure both copy constructor and copy assignment operator are declared for classes featuring a nontrivial destructor. Appropriately deallocate dynamically allocated memory. |
| PVS-Studio | 7.12 | V1006, V515  V554, V611  V701, V748  V773 |  |
| Astree | 20.10 | invalid\_dynamic\_memory\_allocation dangling\_pointer\_use |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | As a general guideline, employ assert statements to handle errors that verify conditions that should never be true if all code is functioning correctly. Since these scenarios do not necessitate recovery at runtime, exceptions should be avoided. Such errors typically indicate code that requires correction, making exception handling unnecessary.  Assert statements serve as visible indicators for programmers and are particularly valuable in detecting deprecated code errors. When assertions are utilized, encountering invalid input results in an immediate halt.  Of particular usefulness is testing for error conditions at points in the code where they should have already been addressed. However, assertions should not serve as replacements for proper error-handling mechanisms. For further insights, refer to MSC11-C for guidance on incorporating diagnostic tests using assertions. |

| **Noncompliant Code** |
| --- |
| The vulnerable code reveals that 's' is dereferenced after deallocation. If this leads to a "write after free" scenario, the vulnerability can be exploited to execute arbitrary code using the permissions of the vulnerable process. |
| myErr = myGraphRoutine(a, b);  ASSERT(!myErr);  \_ASSERT(!myErr); |

| **Compliant Code** |
| --- |
| In this scenario, a graphic routine yields an error code or zero to denote success. Assuming the error-handling mechanism functions correctly, any encountered errors should be appropriately handled, and the variable 'myErr' should be reset to zero before reaching the assertion. If 'myErr' retains a value other than zero, the assertion will fail, causing the program to halt, and prompting the appearance of the Assertion Failed Dialog Box. |
| myErr = myGraphRoutine(a, b);  /\* Code to handle errors and  reset myErr if successful \*/  ASSERT(!myErr); -- MFC version  \_ASSERT(!myErr); -- CRT version |

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

| **Principles(s):** 9 – Use Effective Quality Assurance Techniques: Allows good programs and techniques to be used to test sections of code for any issues. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL03 |  |
| CodeSonar | 6.1p0 | (customization) | Custom check uses assert() |
| ECLAIR | 1.2 | CC2.DCL03 | 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] | In determining when to throw exceptions, it is essential to consider situations where a function encounters an obstacle preventing it from fulfilling its intended task; in such cases, throwing an exception is appropriate. However, if the function can still recover and deliver expected services to the user, handling the issue internally without throwing an exception is preferable. For APIs facing unrecoverable errors, it is advisable to throw exceptions back to the client code, empowering the client to decide on suitable actions, such as retrying connections or exploring alternative endpoints. Exceptions are also useful for validating input parameters of public functions for potential errors. It is recommended to throw exceptions by value and catch them by reference or const reference, ensuring proper memory management by the compiler for the exception object. Additionally, only catch exceptions that can be effectively handled to maintain code robustness. Adherence to SEI Cert C++ Standards, such as ERR61-CPP, ERR51-CPP, ERR57-CPP, and ERR56-CPP, helps ensure proper exception handling practices are followed, guaranteeing program integrity and reliability. |

| **Noncompliant Code** |
| --- |
| According to the SEI Cert C++ Coding Standard, this example demonstrates a flawed copy assignment operator. The implicit invariants of the class are as follows: 1) the 'array' member is a valid (possibly null) pointer, and 2) the 'nElems' member stores the number of elements in the array pointed to by 'array'.  The function deallocates 'array' and assigns the element counter, 'nElems', before allocating a new block of memory for the copy. Consequently, if the new expression throws an exception, the function will have altered the state of both member variables in a manner that violates the implicit invariants of the class. As a result, such an object is left in an indeterminate state, and any operation performed on it, including its destruction, leads to undefined behavior. |
| #include <cstring>    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** |
| --- |
| The copy assignment operator offers the strong exception safety guarantee. The function first allocates new storage for the copy before altering the state of the object. It proceeds to modify the object's state only after the allocation succeeds. Moreover, by copying the array to the newly allocated storage before deallocating the existing array, the function sidesteps the need to test for self-assignment, thus enhancing the code's performance in typical scenarios [Sutter 2004]. |
| #include <cstring>    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):**10 – Adopt a Secure Coding Standard: Allows developers to be held to a high standard regarding safety and security protocols during the project development to ensure any exception data is properly allowed for the allocation of new data storage. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | High | P9 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 527 S, 56 D, 71 D |  |
| PVS-studio | 7.15 | V565, V1023, V5002 |  |
| Parasoft C/C++test | 2021.2 |  | Always catch exceptions |
| Helix QAC | 2021.2 | C++4075, C++4076 |  |

### 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.

An entity, whether it's a company, an individual, or a government entity, should incorporate automated testing into their code creation process. Automated testing ensures that programmers' work undergoes consistent testing procedures, safeguarding the project against errors and warnings. Through DevOps process automation, specific sections or aspects of a project can undergo automated testing with minimal human intervention. This automation acts as a shield for the project's main branch, scrutinizing and testing code before it merges into the main branch. Such testing protocols protect the project from inadequate code segments that may pose security risks or jeopardize the project's integrity. Additionally, automated testing ensures that each team member's code adheres to the same standards and undergoes uniform testing procedures.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | Hgih | Probable | High | Medium | 2 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | Medium | High | 5 |
| STD-005-CPP | High | Likely | Medium | High | 5 |
| STD-006-CPP | Low | Unlikely | High | Low | 1 |
| STD-007-CPP | High | Likely | High | Medium | 3 |

### Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest refers to the process of encrypting data while it's stored. This means that the data remains accessible, but it's rendered unreadable without the correct key. This security measure provides vital protection for companies and governments, offering an extra layer of defense against physical crimes like device theft or unauthorized access to sensitive information stored on thumb drives or other storage media. |
| Encryption in flight | Encryption in transit, also known as encryption in flight, occurs when data is encrypted during transmission. Although the data may not be encrypted while stored or in use, it is encrypted as it's transferred to another location. This security measure safeguards sensitive information against interception by unauthorized parties, as the data remains encrypted without the correct key. Encryption in transit is particularly beneficial for entities whose employees or users engage in telecommuting or mobile work, ensuring data security even when accessed outside the office environment. |
| Encryption in use | Encryption in use occurs when data is encrypted while being accessed or utilized, granting specific users access based on their security clearance. This encryption strategy enhances the security of business or government databases by implementing layers of protection to segregate user activities from employees. By doing so, it prevents employees with lower security clearances or new hires from obtaining unrestricted access to sensitive data within the system. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication employs various security measures such as user logins, passcodes, secure networks, and additional features like fingerprint scanning and two-factor identification. The specific authentication methods utilized depend on the security clearance level assigned to individual users. |
| Authorization | Authorization leverages authentication features to determine the extent of security access granted to a user. Based on their security clearance level, users are authorized to access databases, files, and employee records. Admin-level credentials provide users with the capability to make modifications to databases and files. Additionally, admin users are empowered to add new users and assign them access privileges to specific files within the system. |
| Accounting | Accounting integrates authentication and authorization features to maintain records of data modifications within a system, including details about the individuals responsible for these changes. Some systems mandate users to provide comments elucidating the rationale behind specific alterations. These comments serve dual purposes: enhancing security measures and assisting with future reference. By intertwining these defenses, the system establishes a multi-layered defense mechanism where each feature reinforces the others. |

**\***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
  + 3. Architect and Design for Security Policies
    - Develop a system or program alongside security features at the project's outset, ensuring they are integral components rather than add-ons at the project's end.
  + 4. Keep It Simple
    - Simplify OS development logs to facilitate easy tracking of changes and discrepancies.
  + 5. Default Deny
    - Restrict user account access until authorization and authentication processes are completed.
  + 6. Adhere to the Least Privilege
    - Employ the minimum security access necessary for data retrieval to mitigate accidental data breaches, rather than utilizing maximum-security privileges.
  + 7. Sanitize Data Sent to Other Systems
    - Continuously review files for security risks even after data storage, particularly when suspicious activities are detected.
  + 8. Practice Defense in Depth
    - Foster collaboration among multi-layer defense features, enabling them to collectively respond and alert each other to potential attacks or defense layer failures.
  + 10. Adopt a Secure Coding Standard
    - Implement a consistent security standard within the development team, ensuring adherence to established security protocols for system integrity.
* Firewall logs
  + 1. Validate Input Data
    - Ensure thorough validation of any external data entering the system to mitigate potential security risks.
  + 5. Default Deny
    - Implement a policy where user accounts are automatically denied access until both authorization and authentication processes are completed.
  + 6. Adhere to the Least Privilege
    - Utilize the minimum necessary security access for data retrieval, rather than relying on maximum-security privileges, to prevent inadvertent data breaches.
  + 7. Sanitize Data Sent to Other Systems
    - Continuously inspect files for security risks, even after data storage, particularly when suspicious activities are detected.
  + 8. Practice Defense in Depth
    - Facilitate collaboration among multi-layer defense mechanisms, enabling them to cooperate and alert each other in the event of an attack or defense layer failure.
  + 10. Adopt a Secure Coding Standard
    - Implement a uniform security standard within the development team to ensure consistency and adherence to established security protocols for system integrity.
* Anti-malware logs
  + 1. Validate Input Data
    - Ensure all incoming external data is thoroughly validated to mitigate potential security risks.
  + 5. Default Deny
    - Implement a policy where user account access is automatically denied until successful verification of authorization and authentication.
  + 6. Adhere to the Least Privilege
    - Utilize the minimum necessary security access for data retrieval, rather than relying on maximum-security privileges, to prevent inadvertent data breaches.
  + 7. Sanitize Data Sent to Other Systems
    - Continuously review files for security risks, even post-storage, especially when suspicious activities are detected.
  + 8. Practice Defense in Depth
    - Foster collaboration among multi-layer defense mechanisms, enabling them to cooperate and alert each other in response to attacks or defense layer failures.

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 |  |
| 2.0 | 04/17/2024 | Project 1 Commit | Chris McLernon |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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