**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.

## Ten Core Security Principles



| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Ensure Input Data Integrity focuses on mitigating risks like SQL and command line injection threats. Its aim is to thwart attackers from bypassing authentication loops to gain unauthorized access to sensitive information within an application. This principle applies equally to APIs, which must scrutinize and validate incoming data in request payloads before processing it. All potentially malicious data should undergo thorough validation before any further actions are taken. |
| 1. Heed Compiler Warnings | Compiler Notifications can frequently pinpoint areas in your code where data leakage may occur. Although the notifications provided by Visual Studio or typical Integrated Development Environments (IDEs) are generally effective, it's advisable to leverage static or dynamic analysis tools for deeper insights and to eradicate further security vulnerabilities. |
| 1. Architect and Design for Security Policies | Embedding Security Policies into Architectural Planning entails integrating security measures into software from its conceptualization, rather than adding them as an afterthought. For instance, opting for minimal permissions and considering dividing the application into subsystems when elevated permissions are necessary exemplifies this approach. |
| 1. Keep It Simple | The Principle of Simplicity emphasizes the overall design of the application. Simplifying the codebase and reducing complexity not only streamlines testing and validation during development but also simplifies configuration and usage, reducing the likelihood of errors. In essence, minimizing code results in fewer real-world issues and smoother operation. |
| 1. Default Deny | When crafting the security framework for an application, developers face the decision of structuring the overarching access model. Default Deny proposes that the security model should start by denying access in all instances, only specifying exceptions that grant access. This stands in contrast to designing an open system and subsequently defining exclusions to restrict access. Embracing Default Deny typically results in simpler code within this module and reduces the risk of data leakage. |
| 1. Adhere to the Principle of Least Privilege | As previously stated, it's imperative for an application to consistently follow the principle of least privilege. This principle dictates that code should operate with the minimal necessary permissions and should only elevate permissions when absolutely required. Allowing an application excessive permissions grants access to resources it may not legitimately need, providing attackers with the opportunity to exploit the system. |
| 1. Sanitize Data Sent to Other Systems | Ensuring Data Sanitization for Outgoing Systems addresses the vulnerability that arises when attackers manipulate data sent to a system to launch attacks. SQL injection attacks serve as a prominent example, wherein an unprotected query string can compromise authentication, leading to unauthorized access. This principle also extends to deactivating and eliminating redundant functionality, particularly in commercial products, as they can serve as potential attack vectors for infiltrating the system. |
| 1. Practice Defense in Depth | The concept of Defense in Depth employs numerous defensive layers to mitigate the escalation of a security weakness into a significant vulnerability. Moreover, the presence of multiple defense layers can contain the impact in the event of a successful breach or exploitation. For instance, integrating secure programming practices within a fortified environment fortifies defenses against adversaries attempting to compromise the system and exploit it within the network. |
| 1. Use Effective Quality Assurance Techniques | Before certifying a product or application, it's essential to conduct comprehensive security testing in advance to bolster resilience and pinpoint potential vulnerabilities. This can be achieved through thorough examinations of the source code and penetration testing to fortify the codebase. Moreover, leveraging independent reviewers can enhance the breadth of assessment for the solution and offer impartial perspectives. |
| 1. Adopt a Secure Coding Standard | Employing a robust secure coding standard is paramount when developing a solution. This guarantees that all team members adhere to consistent coding practices, maintaining uniform code quality, and implementing necessary precautions to prevent the introduction of critical 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 | Ensure that float conversions via cast are not accidental. |

| **Noncompliant Code** |
| --- |
| When employing either an implicit or explicit cast, it's imperative to ensure that the operation doesn't lead to the loss or alteration of data integrity. Furthermore, this code is noncompliant due to uncertainty surrounding whether the cast was intended. |
| void function(void) {  float number1 = 2147483647.123432;  int number2 = number1;  } |

| **Compliant Code** |
| --- |
| This code adheres to compliance standards as it explicitly indicates the intent to cast from float to int, acknowledging the potential loss of precision. |
| void function(void) {  float number1 = 2147483647.123432;  int number2 = static\_cast<int>(number1);  } |

|  |
| --- |
| **Principles(s):** 1  This guideline aligns with the principle of "Validate Input Data" as it emphasizes the importance of ensuring that input data is valid before attempting any conversions that might lead to nonsensical outcomes. Failing to validate input data prior to casting could potentially trigger buffer overflows, resulting in program crashes or unintended behaviors. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| Parasoft C/C++test | 2020.2 | CERT\_C-INT31-l | Avoid integer overflows |
| Coverity | 2017.07 | NEGATIVE\_RETURNS | Identify occurrences where an integer expression undergoes implicit conversion to a narrower integer type, where the signedness of an integer value is implicitly altered, or where a complex expression's type is implicitly transformed. |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | For integer types (int, double, float, etc.), ensure division operations do not result in divide-by-zero. |

| **Noncompliant Code** |
| --- |
| This code fails to comply with standards as it encounters a divide-by-zero scenario, leading to an undefined result and potential erratic behavior. |
| void function() {  int numerator = 5;  int denominator = 0;  int result = numerator / denominator;  } |

| **Compliant Code** |
| --- |
| In adherence to best practices, this code ensures that divide-by-zero errors are preemptively handled to prevent undefined behavior. |
| void function() {  int numerator = 5;  int denominator = 0;  try {  int result = numerator / denominator;  }  catch (runtime\_error &e) {  // Handle Error: Division by Zero  }  } |

|  |
| --- |
| **Principles(s):** **10**  The principle of "Establishing a Secure Coding Standard" is relevant here, as it's considered unsound practice to permit divide-by-zero errors. Such errors can yield undefined results and may trigger unpredictable behavior downstream. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | int-division-by-zero  int-modulo-by-zero | Fully checked |
| Cppcheck | 1.66 | zerodiv  zerodivcond | Contextual assessment of division by zero:  Unidentified in cases of division by a struct member, array element, or pointer data equating to zero.  Recognized in instances of potentially hazardous division operations involving variables before or after verifying if said variable equals zero. |
| Parasoft C/C++test | 2020.2 | CERT\_C-INT33-a | Avoid division by zero |
| SonarQube C/C++ Plugin | 3.11 | S3518 |  |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Ensure all strings have a null terminator. |

| **Noncompliant Code** |
| --- |
| This code assumes that the buffer variable is properly terminated but neglects to verify its existence, rendering it noncompliant. |
| #include <iostream>    void process() {  char buffer[12];  std::cin >> buffer;  std::cout << buffer;  } |

| **Compliant Code** |
| --- |
| This code adheres to standards by employing strcpy to copy the buffer contents to the string variable "temp." This method ensures the enforcement of the null terminator, promoting compliance. |
| #include <iostream>  #include <cstring>    void process() {  std::string temp;  char buffer[12];  std::cin >> buffer;  std::strcpy(temp, buffer);  } |

|  |
| --- |
| **Principles(s):** 1, 10  This standard aligns with the principle of "Validate Input Data" as it ensures that strings passed into the program are correctly null-terminated. Failing to null-terminate strings can lead to failures or undefined behavior when using functions that rely on the null terminator. Additionally, "Developing a Coding Standard" is pertinent, as it underscores the importance of maintaining quality string data within the application. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | STRING\_SIZE |  |
| Parasoft C/C++test | 2020.2 | CERT\_C-STR31-e | Avoid using unsafe string functions which may cause buffer overflows |
| PVS-Studio | 7.07 | V518, V645, V727, V755 |  |
| TrustInSoft Analyzer | 1.38 | mem\_access | Exhaustively verified (see one compliant and one non-compliant example). |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | Utilize parameters over raw string queries to access data sources. |

| **Noncompliant Code** |
| --- |
| This code is noncompliant as it injects raw strings directly into the query engine, leaving it vulnerable to additional user input that may cause query failures or enable SQL injection attacks. |
| #include <iostream>  #include <sqlite3.h>  void process() {  sqlite3\* DB;  int exit = 0;  exit = sqlite3\_open("example.db", &DB);  std::string sql("INSERT INTO PERSON VALUES(1, 'STEVE', 'GATES', 30, 'PALO ALTO', 1000.0);");    exit = sqlite3\_exec(DB, sql.c\_str(), NULL, 0, &errorMessage);  if (exit != SQLITE\_OK) {  std::cerr << "Error Insert" << std::endl;  sqlite3\_free(errorMessage);  }  } |

| **Compliant Code** |
| --- |
| This code adheres to the standard by employing parameterized queries instead of raw strings, mitigating the risk of SQL injection attacks. |
| #include <iostream>  #include <sqlite3.h>  void process() {  sqlite3\* DB;  int exit = 0;  exit = sqlite3\_open("example.db", &DB);  std::string sql("INSERT INTO PERSON VALUES(?, ?, ?, ?, ?, ?);");    // Prepare statement with placeholders  sqlite3\_stmt\* stmt;  exit = sqlite3\_prepare\_v2(DB, sql.c\_str(), -1, &stmt, NULL);  // Bind values to placeholders  sqlite3\_bind\_int(stmt, 1, 1);  sqlite3\_bind\_text(stmt, 2, "STEVE", -1, SQLITE\_STATIC);  sqlite3\_bind\_text(stmt, 3, "GATES", -1, SQLITE\_STATIC);  sqlite3\_bind\_int(stmt, 4, 30);  sqlite3\_bind\_text(stmt, 5, "PALO ALTO", -1, SQLITE\_STATIC);  sqlite3\_bind\_double(stmt, 6, 1000.0);  // Execute the prepared statement  exit = sqlite3\_step(stmt);  if (exit != SQLITE\_DONE) {  std::cerr << "Error Insert" << std::endl;  }  // Finalize statement  sqlite3\_finalize(stmt);  } |

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

|  |
| --- |
| **Principles(s):** 3, 7  The principle of "Architect and Design for Security Policies" is relevant here as this standard aims to prevent SQL injection attacks by implementing specific policies. Similarly, "Sanitizing Data Sent to Other Systems" applies as it emphasizes the importance of ensuring that data (in this case, queries) sent to other systems cannot compromise the integrity of the data or system and avoids unexpected responses. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Priority | Medium | 5 | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | SQLI  FB.SQL\_PREPARED\_STATEMENT\_GENERATED\_  FB.SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Findbugs | 1.0 | SQL\_NONCONSTANT\_STRING\_PASSED\_TO\_EXECUTE | Implemented |
| Parasoft Jtest | 2020.2 | BD-SECURITY-TDSQL | Protect against SQL injection |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | When using malloc/alloc, detect and handle memory allocation errors. |

| **Noncompliant Code** |
| --- |
| The provided code is noncompliant since it fails to verify if the memory allocation operation was successful before proceeding with subsequent operations. |
| #include <cstring>    void process(const int \*array, std::size\_t size) noexcept {  int \*copy = (int\*) malloc(5\*sizeof(int));  delete [] copy;  } |

| **Compliant Code** |
| --- |
| To adhere to standards, this code ensures the presence of a valid memory address within the pointer, thereby complying with the established guidelines. |
| #include <cstring>  #include <new>    void process(const int \*array, std::size\_t size) noexcept {  int \*copy = (int\*) malloc(5\*sizeof(int));  if (!copy) {  // Handle error  return;  }  delete [] copy;  } |

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

|  |
| --- |
| **Principles(s):** 5, 10  The principle of "Heeding Compiler Warnings" is significant here because while accessing freed memory often results in application crashes, there are scenarios where it might not occur immediately. This behavior can lead to unauthorized memory access or overwrite values of other variables stored in memory. Additionally, "Developing a Coding Standard" is applicable as this issue needs to be addressed uniformly. Accessing freed memory always poses a risk and should be treated as a standard concern in coding practices. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.66 | leakReturnValNotUsed | Doesn't use return value of memory allocation function |
| Parasoft C/C++test | 2020.2 | CERT\_C-MEM31-a | Ensure resources are freed |
| SonarQube C/C++ Plugin | 3.11 | S3584 |  |
| TrustInSoft Analyzer | 1.38 | malloc | Exhaustively verified. |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Do not leave assertions in production code |

| **Noncompliant Code** |
| --- |
| In this instance, the presence of an assertion within the code flow, rather than within a designated test case, violates compliance standards. Although assertions are typically disabled in production builds, their inclusion within code flow poses a risk of exploitation for injecting malicious code. |
| void process()  {  std::vector<int> myVec\* = std::vector<int>(5);  assert(myVec != nullptr);  myVec.push\_back(1);  } |

| **Compliant Code** |
| --- |
| To ensure compliance, this code removes the assert statement from the code flow. |
| void process()  {  std::vector<int> myVec\* = std::vector<int>(5);  myVec.push\_back(1);  } |

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

|  |
| --- |
| **Principles(s):** 10  This situation primarily relates to "Developing a Coding Standard." By removing the assert statement from production code, we mitigate potential risks and limit the exposure to vulnerabilities that could arise from its inclusion during compilation. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 6.7 | S3346 | Expressions used in "assert" should not produce side effects |

### Coding Standard 7

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

| **Noncompliant Code** |
| --- |
| In this scenario, neither the target method nor the main function includes any exception handling mechanisms for exceptions thrown by raise\_exception(). Consequently, in the absence of a handler, the program terminates abruptly. |
| void raise\_exception() noexcept(false);    void target() {  raise\_exception();  }    int main() {  target();  } |

| **Compliant Code** |
| --- |
| To adhere to standards, the main method in this example implements exception handling by capturing all exceptions using the unexpected error handler. This ensures the program can gracefully terminate. |
| void raise\_exception() noexcept(false);    void target() {  raise\_exception();  }    int main() {  try {  target();  } catch (...) {  // Handle error  }  } |

|  |
| --- |
| **Principles(s):** 10  The principle of "Developing a Coding Standard" is pertinent here. While failing to handle all exceptions might not pose immediate issues in an application under normal circumstances, any unhandled exceptions, particularly those stemming from erroneous data, can lead to application crashes and data corruption. Therefore, establishing a coding standard that mandates proper exception handling helps maintain application stability and integrity. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | main-function-catch-all  early-catch-all | Partially checked |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-008-CPP | Do not leak resources when handling exceptions. |

| **Noncompliant Code** |
| --- |
| This code fails to comply with standards as it lacks proper error handling to clean up the allocated memory in case of an exception thrown during the operation. |
| #include <vector>  void process() {  std::vector<int> \*myVector = new std::vector<int>(5);  try {  myVector->at(1);  } catch (...) {  // Process error with logic  throw;  }  delete myVector;  } |

| **Compliant Code** |
| --- |
| To align with the standard, this code implements a try/catch block with a finally clause to ensure the allocated memory is properly deallocated regardless of whether an error is thrown from at(1). |
| #include <vector>  void process() {  std::vector<int> \*myVector = new std::vector<int>(5);  try {  myVector->at(1);  } catch (...) {  // Process error with logic  throw;  }  finally {  // Handle the deletion of the allocated memory  delete myVector;  }  } |

|  |
| --- |
| **Principles(s):** 2, 9  The principles pertinent to this standard are "Heeding Compiler Warnings" and "Using Effective Quality Assurance Techniques." Paying attention to compiler warnings can help prevent such issues and avoid application crashes. Additionally, employing thorough testing in quality assurance can identify and rectify such issues effectively. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2021.1 |  |  |
| LDRA tool suite | 9.7.1 | 50 D | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-ERR57-a | Ensure resources are freed |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Iterators** | STD-009-CPP | Guarantee that container indices and iterators are within a valid range and are contiguous. |

| **Noncompliant Code** |
| --- |
| This code fails to adhere to the standard as it attempts to access an invalid index, potentially causing undefined behavior. |
| #include <vector>  void process() {  std::vector<int>\* myVector = new std::vector<int>(4);  int a = myVector->at(5);  delete myVector;  } |

| **Compliant Code** |
| --- |
| In alignment with the standard, this code correctly accesses values stored in the vector within valid index ranges. |
| #include <vector>  void process() {  std::vector<int>\* myVector = new std::vector<int>(4);  int a = myVector->at(1);  delete myVector;  } |

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

|  |
| --- |
| **Principles(s):** 10  This standard relates to the principle of "Adopting a Clear Coding Standard." While indexing arrays or vectors within the correct range is technically correct, it's essential to ensure that dynamic index computations stay within valid bounds. It's prudent to check the size before proceeding with operations and to notify users promptly if the index falls outside the acceptable range. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TO  LANG.MEM.TU  LANG.MEM.TBA  LANG.STRUCT.PBB  LANG.STRUCT.PPE | Buffer overrun  Buffer underrun  Type overrun  Type underrun  Tainted buffer access  Pointer before beginning of object  Pointer past end of object |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |
| PVS-Studio | 7.07 | V781 |  |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input/Output** | STD-010-CPP | Do not access closed files. |

| **Noncompliant Code** |
| --- |
| This code fails to comply with the standard as it continues to use the stdout stream after it has been closed, which can lead to undefined behavior. |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    printf("stdout successfully closed.\n");  return 0;  } |

| **Compliant Code** |
| --- |
| To adhere to the standard, this code ensures that the stdout stream is not utilized after it has been closed. |
| #include <stdio.h>    int close\_stdout(void) {  if (fclose(stdout) == EOF) {  return -1;  }    fputs("stdout successfully closed.", stderr);  return 0;  } |

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

|  |
| --- |
| **Principles(s):** 10  This standard is closely related to the principle of "Adopting a Clear Coding Standard." It emphasizes the importance of always checking the status of resources before utilizing them to prevent undefined behaviors resulting from accessing resource pointers. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Implemented |
| Polyspace Bug Finder | R2021a | CERT C: Rule FIO46-C | Checks for use of previously closed resource (rule partially covered) |
| SonarQube C/C++ Plugin | 3.11 | S3588 |  |

## Defense-in-Depth Illustration

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



## Automation

Provide a written explanation using the image provided.



Transitioning from traditional DevOps to DevSecOps involves integrating security practices throughout the software development lifecycle. This entails modifying existing processes across different phases to enforce compliance with security standards.

In the planning phase, it's essential to brainstorm and incorporate security attack patterns and corresponding countermeasures into the product's initial design. This proactive approach ensures that security considerations are addressed from the outset.

During the development and build phases, adherence to secure coding practices and mitigation approaches outlined in coding standards is crucial. This minimizes vulnerabilities and reduces the impact of external manipulations on the application's stability.

In the testing phase, automated unit testing is paramount for assessing individual components of the application. Additionally, implementing integration testing covering the entire application stack helps detect and prevent common security threats such as SQL injection and memory management attacks.

For the release, deployment, operation, and monitoring stages, leveraging secure container systems enhances protection against unauthorized access to the underlying operating system. Automated log collection and analysis aid in identifying and mitigating potential security breaches before they impact the system. Furthermore, monitoring network traffic for abnormal patterns can help detect and thwart distributed denial-of-service (DDoS) attacks aimed at compromising system integrity.

By incorporating these modifications into each phase of the traditional DevOps lifecycle, organizations can transition to a DevSecOps approach, ensuring security is prioritized throughout the software development process.

## Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | Low | Priority | High | Low | 2 |
| STD-002-CPP | Low | Likely | Medium | Low | 2 |
| STD-003-CPP | High | Likely | Medium | High | 1 |
| STD-004-CPP | High | Priority | Medium | 5 | 3 |
| STD-005-CPP | Medium | Priority | Medium | Medium | 2 |
| STD-006-CPP | Low | Unlikely | Low | 1 | 3 |
| STD-007-CPP | Low | Priority | Medium | Low | 3 |
| STD-008-CPP | Low | Priority | High | Low | 3 |
| STD-009-CPP | High | Likely | High | Medium | 2 |
| STD-010-CPP | Medium | Unlikely | Medium | Low | 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 in rest | Encryption-at-rest refers to the practice of encrypting data that is stored but not actively accessed, such as data residing on a hard disk or within a database. The primary objective of this policy is to safeguard data in the event of a security breach where unauthorized access to these files occurs. By encrypting the data, potential attackers would face significant obstacles in decrypting the information, as they would either need to undertake a lengthy and resource-intensive brute force decryption process or acquire access to the encryption keys. This effectively mitigates the impact of data theft, as even if the encrypted files are compromised, the time and effort required to decrypt them would render the stolen data less actionable. Such a robust encryption mechanism significantly enhances the security posture of the organization and provides an additional layer of protection for sensitive data. |
| Encryption at flight | Encryption-in-transit pertains to the practice of securing data as it traverses a network, such as when data is transmitted between a web application and a database. This policy holds significant importance as it aims to safeguard data during its most vulnerable state, exposed to potential interception or eavesdropping. The protection of data in transit is achieved through the implementation of SSL/TLS connections between the web server and the database, ensuring that data exchanged between these endpoints remains encrypted and secure. Additionally, the use of Virtual Private Networks (VPNs) is recommended in scenarios where network segments need to be interconnected, further fortifying the security posture by preventing unauthorized access to transmitted data. By employing these measures, the risk of unauthorized access or interception of data packets by tools like WireShark and TCPDump is greatly reduced, enhancing the overall security of data transmissions across the network. |
| Encryption in use | Encryption-at-Execution refers to the practice of safeguarding data while it is actively being processed or manipulated within a system. This policy is critical as it aims to protect sensitive data during its most vulnerable state, when it is being utilized by applications or services. To achieve encryption-at-execution, various programming techniques and cryptographic methods are employed. For instance, utilizing protected memory mechanisms, such as those offered by the .NET ProtectedMemory class, helps shield sensitive data in memory from unauthorized access or tampering. Additionally, the adoption of advanced cryptographic techniques like Homomorphic encryption enables computations to be performed directly on encrypted data, thereby maintaining data confidentiality even during processing. By implementing these protective measures, the security of data-in-execution is significantly enhanced, mitigating the risk of unauthorized access or exploitation. As a result, potential attackers would face substantial challenges in breaching these safeguards, as decryption of data-at-execution would require significant computational resources and time, rendering the stolen data less actionable and limiting its usefulness. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Verification, also known as authentication, is the method by which a server or application confirms the identity of an individual. Authentication is crucial as it establishes who is authorized to access the application in its entirety. This validation process is managed through a login mechanism, where users typically provide a login identifier linked to their email address or a specific username, along with a password to authenticate their access. To enhance security, additional authentication methods should be implemented, such as two-factor authentication (2FA) or leveraging OAuth technology for single sign-on (SSO) capabilities. This policy should be universally applied to all resources to thwart unauthorized access by unauthorized parties, effectively securing data behind a digital lock where access requires the appropriate authentication credentials. |
| Authorization | Access control, commonly known as authorization, follows authentication and determines the level of access granted to users within a system, its files, and resources. Authorization is pivotal in managing a user's access rights, primarily through role-based permissions that dictate their actions. Aligned with the principle of Default Deny, new users should not inherently possess access to any resources, even upon authentication. Access should only be granted upon assignment of a specific role, defining the user's permitted actions and access privileges to particular resources. This policy is paramount as it prevents authenticated users lacking proper authorization from accessing sensitive data. Moreover, it facilitates seamless elevation of privileges, enabling straightforward promotion of users to higher tiers with enhanced resource access. |
| Accounting | Monitoring, also referred to as accounting, entails the systematic tracking of changes occurring within a given system or resource. An illustrative instance of monitoring involves documenting user access to files and modifications made to database entries. While bespoke systems can be developed to fulfill this function, numerous software solutions offer centralized aggregation of such data, facilitating the generation of comprehensive reports. Notably, platforms like Thycotic Secret Server exemplify this capability, enabling organizations to analyze access patterns and changes to critical resources.  This monitoring framework holds considerable significance both pre- and post-breach. Prior to a breach, it furnishes invaluable insights into user activity, allowing security teams to scrutinize access patterns and potentially thwart unauthorized intrusions. Conversely, in the aftermath of a breach, this information assumes paramount importance for conducting thorough root cause analyses and swiftly remedying identified vulnerabilities.  This policy is highly relevant to our scenario as it serves to fortify defenses against unauthorized intrusions into our secured data, ensuring proactive monitoring and rapid response to security incidents. |

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

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

The only item you must complete beyond this point is the Policy Version History table.

# Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

# Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

# Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

# Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

# Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

# Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 04/17/2021 | Initial Milestone Completion | Michael Mihalik | David Buksbaum |
| 1.2 | 04/24/2021 | Security Policy Completion | Mihcael Mihalik | David Buksbaum |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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