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



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc68896745)

[Purpose 2](#_Toc68896746)

[Scope 2](#_Toc68896747)

[Ten Core Security Principles (Seacord, 2018) 2](#_Toc68896748)

[C/C++ Ten Coding Standards 3](#_Toc68896749)

[STD-001-CPP. Avoid unsigned integer overflow 4](#_Toc68896750)

[STD-002-CPP. Prevent divide by zero errors 6](#_Toc68896751)

[STD-003-CPP. Prevent off-by-one error with strncpy() 8](#_Toc68896752)

[STD-004-CPP. Prevent SQL Injection 10](#_Toc68896753)

[STD-005-CPP. Use new and delete for safe pointer allocation 12](#_Toc68896754)

[STD-006-CPP. Do not use assertions in releases 14](#_Toc68896755)

[STD-007-CPP. Handle all exceptions 16](#_Toc68896756)

[STD-008-CPP. Do not use magic numbers 18](#_Toc68896757)

[STD-009-CPP. Pointers should place the \* close to the pointer type 20](#_Toc68896758)

[STD-010-CPP. Close open file handlers 22](#_Toc68896759)

[Defense-in-Depth Illustration 24](#_Toc68896760)

[Project One 24](#_Toc68896761)

[1. Revise the C/C++ Standards 24](#_Toc68896762)

[2. Risk Assessment 24](#_Toc68896763)

[3. Automated Detection 24](#_Toc68896764)

[4. Automation 24](#_Toc68896765)

[5. Summary of Risk Assessments 25](#_Toc68896766)

[6. Create Policies for Encryption and Triple A 26](#_Toc68896767)

[7. Map the Principles 26](#_Toc68896768)

[Audit Controls and Management 28](#_Toc68896769)

[Enforcement 28](#_Toc68896770)

[Exceptions Process 28](#_Toc68896771)

[Distribution 29](#_Toc68896772)

[Policy Change Control 29](#_Toc68896773)

[Policy Version History 29](#_Toc68896774)

[Appendix A Lookups 29](#_Toc68896775)

[Approved C/C++ Language Acronyms 29](#_Toc68896776)

[References 30](#_Toc68896777)

# 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 (Seacord, 2018)

| **Principles** | **Description** |
| --- | --- |
| 1. ValidateInput Data | Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set. |
| 1. Keep It Simple | Keep the design as simple and small as possible. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex. |
| 1. Default Deny | Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted. |
| 1. Adhere to the Principle of Least Privilege | Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem. |
| 1. Practice Defense in Depth | Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment. |
| 1. Use Effective Quality Assurance Techniques | Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions. |
| 1. Adopt a Secure Coding Standard | Develop and/or apply a secure coding standard for your target development language and platform. |

## C/C++ Ten Coding Standards

This list of coding standards are a collection of rules and guidelines for software development at Green Pace. These standards help to determine secure and proper development procedures and methods for development in C/C++.

### STD-001-CPP. Avoid unsigned integer overflow

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Avoid using unsigned integers, use signed integers instead to avoid overflow  Unsigned integers are error prone and produce little advantage over the use of signed integers unless severely memory constrained. They can never be negative and nothing prevents computations of unsigned integers that mathematically would produce a negative value but instead being represented as a large positive value instead. |

| **Noncompliant Code** |
| --- |
| This noncompliant code subtracts a large unsigned integer value from a smaller unsigned integer value producing a large numerical value. |
| void func(unsigned int ui\_x, unsigned int ui\_y) {  unsigned int ui\_result;  ui\_result = ui\_x - ui\_y;  }  int main()  {  unsigned int val1 = 1;  unsigned int val2 = 2;  // subtract larger value val2 from smaller value val1 1-2  func(val1, val2) // resulting value 4294967295  } |

| **Compliant Code** |
| --- |
| This compliant solution uses signed integers instead of unsigned integers. This will produce a negative value when subtracting a larger value from a smaller value. |
| void func(signed int si\_x, signed int si\_y) {  signed int si\_result;  si\_result = si\_x - si\_y;  }  int main()  {  signed int val1 = 1;  signed int val2 = 2;  // subtract larger value val2 from smaller value val1 1-2  func(val1, val2) // resulting value -1  } |

|  |
| --- |
| **Principles(s):** This standard follows the keep it simple policy as this will provide a simpler approach across all projects. Additionally, this standard also follows the Defense in depth policy as well. This is because if the standard is not followed an attacker could expose a vulnerability. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **integer-overflow** | Fully checked |
| CodeSonar | 6.0p0 | **ALLOC.SIZE.ADDOFLOW ALLOC.SIZE.IOFLOW ALLOC.SIZE.MULOFLOW ALLOC.SIZE.SUBUFLOW MISC.MEM.SIZE.ADDOFLOW MISC.MEM.SIZE.BAD MISC.MEM.SIZE.MULOFLOW MISC.MEM.SIZE.SUBUFLOW** | Addition overflow of allocation size Integer overflow of allocation size Multiplication overflow of allocation size Subtraction underflow of allocation size Addition overflow of size Unreasonable size argument Multiplication overflow of size Subtraction underflow of size |
| Compass/ROSE |  |  | Can detect violations of this rule by ensuring that operations are checked for overflow before being performed (Be mindful of exception INT30-EX2 because it excuses many operations from requiring validation, including all the operations that would validate a potentially dangerous operation. For instance, adding two unsigned ints together requires validation involving subtracting one of the numbers from UINT\_MAX, which itself requires no validation because it cannot wrap.) |
| Coverity | 2017.07 | **INTEGER\_OVERFLOW** | Implemented |

### STD-002-CPP. Prevent divide by zero errors

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Prevent Divide By Zero errors  Dividing by zero results in a divide by zero error. Using a safe divide capability should return a zero value when this happens. |

| **Noncompliant Code** |
| --- |
| This noncompliant code divides a numerator by zero without any checks divide by zero checks. |
| int func(int numerator, int denominator) {  return numerator / denominator;  }  int main()  {  int numerator = 1;  int denominator = 0;    func(numerator, denominator); // exception Integer division by zero.  } |

| **Compliant Code** |
| --- |
| This compliant code checks for the denominator value of 0 and returns a 0 value if true. This prevents an division by zero exception. |
| int func(int numerator, int denominator) {  if (denominator == 0) {  return 0;  }  return numerator / denominator;  }  int main()  {  int numerator = 1;  int denominator = 0;    func(numerator, denominator); // results in 0  } |

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

|  |
| --- |
| **Principles(s):** This standard follows the validated input policy as this will prevent unwanted and program halting behaviors that attackers could expose and exploit. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | P6 | L2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **int-division-by-zero**  **int-modulo-by-zero** | Fully checked |
| CodeSonar | 6.0p0 | **LANG.ARITH.DIVZERO LANG.ARITH.FDIVZERO** | Division by zero Float Division By Zero |
| Compass/ROSE |  |  | Can detect some violations of this rule (In particular, it ensures that all operations involving division or modulo are preceded by a check ensuring that the second operand is nonzero.) |
| Coverity | 2017.07 | **DIVIDE\_BY\_ZERO** | Fully implemented |

### STD-003-CPP. Prevent off-by-one error with strncpy()

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Prevent off-by-one error when copying string data with strncpy() |

| **Noncompliant Code** |
| --- |
| This noncompliant code demonstrates copying string data but not properly accounting for the null terminator. The function strncpy() is not guaranteed to null-terminate the destination string. The caller must ensure the string is properly null terminated. |
| int main()  {  const int dest\_size = 7;  char src[] = "abcdef";  char dst[dest\_size];    strncpy(dst, src, dest\_size-1); // not copy null terminator to dst  // dst is not null terminated now    } |

| **Compliant Code** |
| --- |
| This compliant code adds a null terminator to the end of the destination string ensuring it is properly terminated. |
| int main()  {  const int dest\_size = 7;  char src[] = "abcdef";  char dst[dest\_size];    strncpy(dst, src, dest\_size-1); // not copy null terminator to dst  dst[dest\_size - 1] = '\0'; // ensures null terminator is added    } |

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

|  |
| --- |
| **Principles(s):** This standard follows the secure coding standard to prevent improperly truncated string that can cause a non-null terminated string. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Probable | Medium | **P8** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **MISC.MEM.NTERM** | No Space For Null Terminator |
| Compass/ROSE |  |  | Could detect violations in the following manner: all calls to strncpy() and the other functions should be followed by an assignment of a terminating character to null-terminate the string |
| GCC | 8.1 | -Wstringop-truncation | Detects string truncation by strncat and strncpy. |

### STD-004-CPP. Prevent SQL Injection

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Prevent SQL Injection  SQL Injection vulnerabilities occur in applications that process SQL queries that originate from untrusted or external sources. Use prepared statements or escape external input to prevent SQL Injections |

| **Noncompliant Code** |
| --- |
| This noncompliant code does not validate or escape user supplied input to the SQL query. |
| int main()  {  sqlConn = sql.connection(); // simulated connection for demo purposes  std::string ext\_input\_usr, ext\_input\_pwd; // simulated username and password  std::string sqlStatement = "SELECT \* FROM db\_user WHERE username=" + ext\_input\_usr + " and password=" + ext\_input\_pwd;  sqlConn.execute(sqlStatement);  } |

| **Compliant Code** |
| --- |
| This compliant code uses prepared statements with parameterized query statement. This makes the entire query one long string and prevents a sql injection attack. |
| int main()  {  sqlConn = sql.connection(); // simulated connection for demo purposes  std::string ext\_input\_usr, ext\_input\_pwd; // simulated username and password  std::string query = "SELECT \* FROM db\_user WHERE username=? and password=?"; // uses parameter placeholders ?  std::string sqlStatement = sqlConn.PrepareStatement(query);  sqlStatement.setParameter(1, ext\_input\_usr); // replaces 1st placeholder with external value  sqlStatement.setParameter(2, ext\_input\_pwd); // replaces 2nd placeholder with external value  sqlConn.execute(sqlStatement); // executes sql query safely  } |

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

|  |
| --- |
| **Principles(s):** This coding standard adheres to the Sanitize Data sent to other systems principle as well as the validated input principle. Attackers may be able to invoke unused functionality in these components through the use of SQL. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported by stubbing/taint analysis |
| CodeSonar | 6.0p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| Coverity | 6.5 | **TAINTED\_STRING** | Fully implemented |
| Klocwork | 2018 | **NNTS.TAINTED** **SV.TAINTED.INJECTION** |  |

### STD-005-CPP. Use new and delete for safe pointer allocation

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Use new and delete expressions for safe memory allocation and deallocation  The new expressions allocates memory to hold an object being requested and may initialize the object in memory. The delete expression will deallocate the memory used by the object being deleted. C has std::free() function and this should never be used with objects initialized with new. |

| **Noncompliant Code** |
| --- |
| This noncompliant code incorrectly pairs allocation and deallocation methods for an object. |
| int main()  {  int\* myInt = new int(12); // C++ memory allocation  std::free(myInt); // WRONG: C memory deallocation  } |

| **Compliant Code** |
| --- |
| This compliant code correctly pairs allocation and deallocation using new and delete. |
| int main()  {  int\* myInt = new int(12); // C++ memory allocation  delete myInt; // Correct: C++ memory deallocation  } |

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

|  |
| --- |
| **Principles(s):**This coding standard follows the secure code principle in that properly using pointers prevents misuse and accessing memory that doesn’t exist. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **invalid\_dynamic\_memory\_allocation dangling\_pointer\_use** |  |
| Axivion Bauhaus Suite | 6.9.0 | **CertC++-MEM51** |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDeleteLeaks -Wmismatched-new-delete clang-analyzer-unix.MismatchedDeallocator | Checked by clang-tidy, but does not catch all violations of this rule |
| CodeSonar | 6.0p0 | **ALLOC.FNH ALLOC.DF ALLOC.TM** | Free non-heap variable Double free Type mismatch |

### STD-006-CPP. Do not use assertions in releases

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Do not use assertions except for testing and debugging  Assertions can be disabled and therefore should not be used in asserting certain conditions outside of debugging and unit testing. Using logic conditions will be the more appropriate solution. |

| **Noncompliant Code** |
| --- |
| This noncompliant code incorrectly uses an assertion statement in the release code to test if a user is an admin before returning an admin value. If assertion is disabled then the assertion statement is ignored and the admin value is always returned. |
| #define NDEBUG // disables assertion  #include <assert.h>  int main()  {  int usr\_role = 0;  // Statement below fails if usr\_role != 1 (i.e. user is admin),  // but is not executed at all when compiling with -NDEBUG!  assert(usr\_role == 1); // This assertion is ignored  return\_admin\_value(); // will return admin value  } |

| **Compliant Code** |
| --- |
| This compliant code is a proper use of checking |
| #define NDEBUG  int main()  {  int usr\_role = 0;  // Statement below does not return admin data if usr\_role != 1  if (usr\_role == 1) {  return\_admin\_value(); // will return admin value  }  } |

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

|  |
| --- |
| **Principles(s):** This standard follows the principle of Input validation. Proper input validation can eliminate the vast majority of software vulnerabilities such as relying on assert statements in release code. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Probable | Low | P3 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| AssertChecker | 1.0,.0 | ASSERT\_CHECKER | Checks for usage of assert() and displays error. |

### STD-007-CPP. Handle all exceptions

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Handle All Exceptions  When an exception is thrown, there must be a handler to match the exception thrown. All exceptions must be caught and controlled to allow the stack to be properly unwound prior to termination. |

| **Noncompliant Code** |
| --- |
| This noncompliant code shows that there are no exception controls in func() or in main(). By calling throw\_excep() then the program terminates abnormally. |
| // exception call  void throw\_excep() noexcept(false);    void func() {  // function throws an exception  throw\_excep();  }    int main() {  func(); // exception is thrown, no handler  } |

| **Compliant Code** |
| --- |
| This compliant code shows that when an exception is thrown by func() that the catch() statement will catch the exception. This exception can now be properly handled and graceful termination can occur. |
| // exception call  void throw\_excep() noexcept(false);    void func() {  // function throws an exception  throw\_excep();  }    int main() {  try{  func(); // exception is thrown  catch (...) {  // exception handling  }  } |

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

|  |
| --- |
| **Principles(s):** [This standard follows the adopt a secure coding standard principle. Handling exceptions prevents misuse of the program and allows for better logging and auditing of misues. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | **main-function-catch-all early-catch-all** | Partially checked |
| Axivion Bauhaus Suite | 6.9.0 | **CertC++-ERR51** |  |
| 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 |

### STD-008-CPP. Do not use magic numbers

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | [STD-008-CPP] | Do not use magic numbers  A magic number is a bare number used in code. It is magic because the number has no apparent meaning and other developers are unable to make sense of what the number means. |

| **Noncompliant Code** |
| --- |
| This noncompliant example shows that the number 22 is a magic number. It is unclear what 22 means in this context and why it is causing the car to stop. |
| if (car == 22) { stopTheCar(); } |

| **Compliant Code** |
| --- |
| This compliant example shows that the number 22 is not a magic number. It is clear what 22 means in this context since it is assigned to a variable named OUT\_OF\_GAS and known why it is causing the car to stop. |
| #define OUT\_OF\_GAS (22)  if (car == OUT\_OF\_GAS) { stopTheCar(); } |

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

|  |
| --- |
| **Principles(s):** This standard follows the principle of adopting a secure coding standard. Using magic numbers can causes confusion and can be complicit in introducing security concerns without valid variable naming. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 6.9.0 | **CertC-DCL06** |  |
| Compass/ROSE |  |  | Could detect violations of this recommendation merely by searching for the use of "magic numbers" and magic strings in the code itself. That is, any number (except a few canonical numbers: −1, 0, 1, 2) that appears in the code anywhere besides where assigned to a variable is a magic number and should instead be assigned to a const integer, enum, or macro. Likewise, any string literal (except "" and individual characters) that appears in the code anywhere besides where assigned to a char\* or char[] is a magic string |
| ECLAIR | 1.2 | **CC2.DCL06** | Fully implemented |
| LDRA tool suite | 9.7.1 | **201 S** | Fully implemented |

### STD-009-CPP. Pointers should place the \* close to the pointer type

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Variables | [STD-009-CPP] | Pointers should place the \* close to the pointer type  Pointers should place the \* close to the pointer type and not close to the variable name. This makes it clear that the variable is a pointer to a specific type. Placing \* close the variable name should be reserved for dereferencing. |

| **Noncompliant Code** |
| --- |
| This noncompliant example shows the pointer variable declaration with the \* close to the variable name. It is less clear that this is a pointer to a string. |
| int main()  {  std::string \*pStr;  } |

| **Compliant Code** |
| --- |
| This compliant example shows the pointer variable declaration with the \* close to the variable type. It is clear that this is a pointer to a string named pStr. |
| int main()  {  std::string\* pStr;  } |

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

|  |
| --- |
| **Principles(s):** This standard follows the adopt a secure coding standard principle. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | **P2** | **L3** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | **296 S** | Partially implemented |
| Parasoft C/C++test | 2020.2 | **CERT\_CPP-DCL53-a** **CERT\_CPP-DCL53-b** | Always declare functions at file scope Identifier declared in a local or function prototype scope shall not hide an identifier declared in a global or namespace scope |
| Polyspace Bug Finder | R2020a | CERT C++: DCL53-CPP | Checks for declarations that can be confused between:   * Function and object declaration * Unnamed object or function parameter declaration   Rule fully covered. |
| LDRA tool suite | 9.7.1 | **296 S** | Partially implemented |

### STD-010-CPP. Close open file handlers

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| File Input/Output | [STD-010-CPP] | Close File handlers when no longer required.  A file open() call must be matched with a file close() call when file IO use has completed. This may be at the end of a function call or at the end of the program prior to termination. All open file handlers need to be closed when exceptions are thrown and caught as well. |

| **Noncompliant Code** |
| --- |
| This noncompliant code shows the file being opened but never closed. It is never properly closed when the function returns. |
| void file\_func(const std::string& fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    /\* do something with the file/file contents\*/  return; // file handle is left open  } |

| **Compliant Code** |
| --- |
| This compliant code shows once work has completed with the files that the file is closed prior to returning out of the function. |
| void file\_func(const std::string& fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }    /\* do something with the file/file contents\*/  // close the file handler  file.close();  return; // file handle is now closed  } |

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

|  |
| --- |
| **Principles(s):** This follows the principle of adopting a secure coding standard as leaving file handles open and not properly closing then can cause corruptions and additional memory related issue. |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | **ALLOC.LEAK** | Leak |
| Klocwork | 2018 | **RH.LEAK** |  |
| Parasoft C/C++test | 2020.2 | **CERT\_CPP-FIO51-a** | Ensure resources are freed |
| Parasoft Insure++ |  |  | Runtime detection |

## 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.

DevSecOps is the approach to applying security to product development in the DevOps process. This allows for a more secure approach via various automation techniques to incorporate security within the DevOps model. This is the idea of implementing security as code into the development lifecycle and to maintain monitoring and analytics of the lifecycle. This is a simple concept but has important aspects incorporated into it. The idea that at all levels and integrations points security should be implemented to ensure a safe product to market.

Automated testing within a development lifecycle is the goal and target of DevSecOps. Implementing security CI/CD pipelines that scans code upon merging or unit testing when a merge is performed. Writing effective tests to broadly cover the code base to ensure continuous security compliance.

## 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 | Likely | High | P9 | L2 |
| STD-002-CPP | Medium | Likely | Medium | P6 | L2 |
| STD-003-CPP | Medium | Probable | Medium | **P8** | **L2** |
| STD-004-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-005-CPP | High | Likely | Medium | **P18** | **L1** |
| STD-006-CPP | Low | Probable | Low | P3 | L3 |
| STD-007-CPP | Low | Probable | Medium | **P4** | **L3** |
| STD-008-CPP | Low | Unlikely | Medium | **P2** | **L3** |
| STD-009-CPP | Low | Unlikely | Medium | **P2** | **L3** |
| STD-010-CPP | Medium | Unlikely | Medium | **P4** | **L3** |

## 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 is the policy to encrypt sensitive data stored on persistent storage such as credit card numbers or social security numbers stored in a database. |
| Encryption at flight | Encryption in flight is the process of encrypting data between systems. This would be HTTPS/TLS encryption between a client browser and the backend web server. |
| Encryption in use | Encryption in use is securing data that is being handled in memory during transit. This would be something like hashing a password and validating the hash against a stored hash in a database over HTTPS. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the process of validating a user is who they say they are. This is typically done with username and password coupled with multi-factor authentication such as an RSA token. |
| Authorization | Authorization is the process of validating a user’s permission or privilege to access or process some data. |
| Accounting | Accounting is the process in logging and auditing all actions of a user so that a trail can be rebuilt on what, when, and how a user accessed or modified data. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

## Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

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

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

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

# Audit Controls and Management

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

Evidence will include the following:

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

# Enforcement

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

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

# Exceptions Process

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

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

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

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

# Distribution

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

# Policy Change Control

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

# Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 2.0 | 04/11/2021 | Green Pace Secure Development | Sammy Shuck |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

# References

Seacord, R. C. (2018, May). *Top 10 Secure Coding Practices*. Retrieved from CERT Secure Coding: https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices