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



Green Pace Secure Development Policy

Robert Worker

4/11/2021

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 6](#_Toc52464060)

[Coding Standard 3 8](#_Toc52464061)

[Coding Standard 4 10](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 21](#_Toc52464067)

[Coding Standard 10 23](#_Toc52464068)

Defense-in-Depth Illustration 26

Project One 27

1. Revise the C/C++ Standards 27

2. Risk Assessment 28

3. Automated Detection 28

4. Automation 29

5. Summary of Risk Assessments 29

6. Create Policies for Encryption and Triple A 29

7. Map the Principles 30

Audit Controls and Management 30

Enforcement 30

Exceptions Process 31

Distribution 31

Policy Change Control 32

Policy Version History 32

Appendix A Lookups 32

Approved C/C++ Language Acronyms 32

References 33

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | “Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].” (Seacord, & Schiela, 2018). Additionally, methods and techniques related to data sanitization and exclusion from user input formatted strings shall be applied to validate input data. |
| 1. Heed Compiler Warnings | “Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [[C MSC00-A](https://wiki.sei.cmu.edu/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels), [C++ MSC00-A](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046361)]. Use static and dynamic analysis tools to detect and eliminate additional security flaws” (Seacord, & Schiela, 2018). Reference compiler diagnostic messages and understand implication of suppressing compiler warnings before doing so. |
| 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” (Seacord, & Schiela, 2018). Consider implementing DevSecOps that integrates security throughout the SDLC. |
| 1. Keep It Simple | Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. 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” (Seacord, & Schiela, 2018). Adhering to language-specific naming conventions for classes, methods, variables, etc. allow for easier reference for code audits and tracking. |
| 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 [Saltzer 74, Saltzer 75]” (Seacord, & Schiela, 2018). Implement protected and private classes where applicable to negate members of external public classes from accessing members within the private classes. |
| 1. Adhere to the Principle of Least Privilege | “Every process should execute with the 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 [Saltzer 74, Saltzer 75]” (Seacord, & Schiela, 2018). Isolating functions or methods by accessing privileges circumvents unauthorized access when used in conjunction with user input validation methods. |
| 1. Sanitize Data Sent to Other Systems | “Sanitize all data passed to complex subsystems [[C STR02-A](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+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” (Seacord, & Schiela, 2018). Before exporting data, verify through conditional statements or data verification methods that the data being sent external if formatted correctly, is of the accepted data type, and is in range of the data type being sent. |
| 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](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-securityflaw) from becoming an exploitable vulnerability and/or limit the consequences of a successful [exploit](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-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 [Seacord 05]” (Seacord, & Schiela, 2018). Adding multiple and redundant verification and authentication checks so the act of circumventing these checks is more difficult. |
| 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 [Seacord 05]” (Seacord, & Schiela, 2018). SAST and DAST whitebox and blackbox testing methods with defined test cases that adhere to product specifications maintains product quality and security. |
| 1. Adopt a Secure Coding Standard | “Develop and/or apply a secure coding standard for your target development language and platform” (Seacord, & Schiela, 2018). Referencing authentic and trusted “Best Practices” for the language, framework, programming environment, or additional tools assist in adhering to quality and trusted standards leading to more robust code and securer product. |

### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | DCL12-C | “Abstract data types are not restricted to object-oriented languages such as C++ and Java. They should be created and used in C language programs as well. Abstract data types are most effective when used with private (opaque) data types and information hiding” (Seacord & Gangopadhyay, 2019). |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example is based on the managed string library developed by CERT [[Burch 2006](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Burch06)]. In this example, the managed string type and the functions that operate on this type are defined in the string\_m.h header file as follows:” (Seacord & Gangopadhyay, 2019). |
| struct string\_mx {  **size\_t** size;  **size\_t** maxsize;    unsigned **char** strtype;  **char** \*cstr;  };    typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ |

| **Compliant Code** |
| --- |
| “This compliant solution reimplements the string\_mx type as a private type, hiding the implementation of the data type from the user of the managed string library. To accomplish this, the developer of the private data type creates two header files: an external string\_m.h header file that is included by the user of the data type and an internal file that is included only in files that implement the managed string abstract data type” (Seacord & Gangopadhyay, 2019). |
| struct string\_mx;  typedef struct string\_mx string\_mx;    /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ |

|  |
| --- |
| **Principles(s):** “The implementation of the string\_mx type is fully visible to the user of the data type after including the string\_m.h file. Programmers are consequently more likely to directly manipulate the fields within the structure, violating the software engineering principles of information hiding and data encapsulation and increasing the probability of developing incorrect or nonportable code” (Seacord & Gangopadhyay, 2019). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axicion Bauhaus Suite | 6.9.0 | CertC-DCL12 | N/A |
| LDRA tool suite | 9.7.1 | 104 D | Partially implemented |
| Polyspace Bug Finder | R2020a | Cert C;  Rec. DCL-12C | Checks for structure or union object implementation visible in file where pointer to this object is not dereferenced (rule partially covered) |
| Parasoft C/C++ test | 2020.2 | Cert\_C-DCL12-a | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | NUM00-Java | “Programs must not allow mathematical operations to exceed the integer ranges provided by their primitive integer data types” (Mertikas & Rozenau, 2021). |

| **Noncompliant Code** |
| --- |
| “Either operation in this noncompliant code example could result in an overflow. When overflow occurs, the result will be incorrect” (Mertikas & Rozenau, 2021). |
| public static int multAccum(int oldAcc, int newVal, int scale) {    // May result in overflow    return oldAcc + (newVal \* scale);  } |

| **Compliant Code** |
| --- |
| “This compliant solution uses the safeAdd() and safeMultiply() methods defined in the "Precondition Testing" section to perform secure integral operations or throw ArithmeticException on overflow” (Mertikas & Rozenau, 2021). |
| public static int multAccum(int oldAcc, int newVal, int scale) {    return safeAdd(oldAcc, safeMultiply(newVal, scale));  } |

|  |
| --- |
| **Principles(s): “**Failure to perform appropriate range checking can lead to integer overflows, which can cause unexpected program control flow or unanticipated program behavior.**”** (Mertikas & Rozenau, 2021). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | BAD\_SHIFT  OVERFLOW\_BEFORE\_WIDEN | Implemented |
| Parasoft Jtest | 2020.2 | PB.NUM.ICO  PB.NUM.BSA  PB.NUM.CACO | Avoid calculations which result in overflow or NaN Do not use an integer outside the range of [0, 31] as the amount of a shift Avoid using compound assignment operators in cases which may cause overflow |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | IDS01-Java | “Many applications that accept untrusted input strings employ input filtering and validation mechanisms based on the strings' character data. For example, an application's strategy for avoiding cross-site scripting (XSS) vulnerabilities may include forbidding <script> tags in inputs. Such blacklisting mechanisms are a useful part of a security strategy, even though they are insufficient for complete input validation and sanitization” (Mohindra & Snavely, 2017). |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example attempts to validate the String before performing normalization” (Mohindra & Snavely, 2017). |
| // String s may be user controllable  // \uFE64 is normalized to < and \uFE65 is normalized to > using the NFKC normalization form  String s = "\uFE64" + "script" + "\uFE65";    // Validate  Pattern pattern = Pattern.compile("[<>]"); // Check for angle brackets  Matcher matcher = pattern.matcher(s);  if (matcher.find()) {    // Found black listed tag    throw new IllegalStateException();  } else {    // ...  }    // Normalize  s = Normalizer.normalize(s, Form.NFKC); |

| **Compliant Code** |
| --- |
| “This compliant solution normalizes the string before validating it. Alternative representations of the string are normalized to the canonical angle brackets. Consequently, input validation correctly detects the malicious input and throws an IllegalStateException” (Mohindra & Snavely, 2017). |
| String s = "\uFE64" + "script" + "\uFE65";    // Normalize  s = Normalizer.normalize(s, Form.NFKC);    // Validate  Pattern pattern = Pattern.compile("[<>]");  Matcher matcher = pattern.matcher(s);  if (matcher.find()) {    // Found blacklisted tag    throw new IllegalStateException();  } else {    // ...  } |

|  |
| --- |
| **Principles(s):** “Validating input before normalization affords attackers the opportunity to bypass filters and other security mechanisms. It can result in the execution of arbitrary code.” (Mohindra & Snavely, 2017). |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| Fortify | 1.0 | Process\_Control | Implemented |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | IDS00-Java | SQL queries from untrusted sources leading to exploitation of un-sanitized query inputs |

| **Noncompliant Code** |
| --- |
| “This code example modifies the doPrivilegedAction() method to use a PreparedStatement instead of java.sql.Statement. However, the prepared statement still permits a SQL injection attack by incorporating the unsanitized input argument username into the prepared statement” (Mohindra & Rozenau, 2021). |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;    class Login {    public Connection getConnection() throws SQLException {      DriverManager.registerDriver(new              com.microsoft.sqlserver.jdbc.SQLServerDriver());      String dbConnection =        PropertyManager.getProperty("db.connection");      // Can hold some value like      // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"      return DriverManager.getConnection(dbConnection);    }      String hashPassword(char[] password) {      // Create hash of password    }      public void doPrivilegedAction(      String username, char[] password    ) throws SQLException {      Connection connection = getConnection();      if (connection == null) {        // Handle error      }      try {        String pwd = hashPassword(password);        String sqlString = "select \* from db\_user where username=" +          username + " and password =" + pwd;        PreparedStatement stmt = connection.prepareStatement(sqlString);          ResultSet rs = stmt.executeQuery();        if (!rs.next()) {          throw new SecurityException("User name or password incorrect");        }          // Authenticated; proceed      } finally {        try {          connection.close();        } catch (SQLException x) {          // Forward to handler        }      }    }  } |

| **Compliant Code** |
| --- |
| “Uses a parametric query with a *?* character as a placeholder for the argument” (Mohindra & Rozenau, 2021). |
| public void doPrivilegedAction(    String username, char[] password  ) throws SQLException {    Connection connection = getConnection();    if (connection == null) {      // Handle error    }    try {      String pwd = hashPassword(password);        // Validate username length      if (username.length() > 8) {        // Handle error      }        String sqlString =        "select \* from db\_user where username=? and password=?";      PreparedStatement stmt = connection.prepareStatement(sqlString);      stmt.setString(1, username);      stmt.setString(2, pwd);      ResultSet rs = stmt.executeQuery();      if (!rs.next()) {        throw new SecurityException("User name or password incorrect");      }        // Authenticated; proceed    } finally {      try {        connection.close();      } catch (SQLException x) {        // Forward to handler      }    }  } |

|  |
| --- |
| **Principles(s):** “Failure to sanitize user input before processing or storing it can result in injection attacks” (Mohindra & Rozenau, 2021). |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | P12 | L1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and security errors |
| 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 |
| Fortify | 1.0 | HTTP\_Response\_Splitting  SQL\_Injection\_Persistence  SQL\_Injection | Implemented |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM30-C | “It is at the memory manager's discretion when to reallocate or recycle the freed memory. When memory is freed, all pointers into it become invalid, and its contents might either be returned to the operating system, making the freed space inaccessible, or remain intact and accessible. As a result, the data at the freed location can appear to be valid but change unexpectedly. Consequently, memory must not be written to or read from once it is freed” (Seacord & Kafka, 2021). |

| **Noncompliant Code** |
| --- |
| “This example from Brian Kernighan and Dennis Ritchie [[Kernighan 1988](https://wiki.sei.cmu.edu/confluence/display/c/AA.+Bibliography#AA.Bibliography-Kernighan88)] shows both the incorrect and correct techniques for freeing the memory associated with a linked list. In their (intentionally) incorrect example, p is freed before p->next is executed, so that p->next reads memory that has already been freed” (Seacord & Kafka, 2021). |
| #include <stdlib.h>    struct node {  **int** value;    struct node \*next;  };    void free\_list(struct node \*head) {    for (struct node \*p = head; p != NULL; p = p->next) {  **free**(p);    }  } |

| **Compliant Code** |
| --- |
| “Kernighan and Ritchie correct this error by storing a reference to p->next  in q before freeing p:” (Seacord & Kafka, 2021). |
| #include <stdlib.h>    struct node {  **int** value;    struct node \*next;  };    void free\_list(struct node \*head) {    struct node \*q;    for (struct node \*p = head; p != NULL; p = q) {      q = p->next;  **free**(p);    }  } |

|  |
| --- |
| **Principles(s):** “Reading memory that has already been freed can lead to abnormal program termination and denial-of-service attacks. Writing memory that has already been freed can additionally lead to the execution of arbitrary code with the permissions of the vulnerable process. ”(Seacord & Kafka, 2021). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 21.10 | dangling\_pointer\_use | Supported  Astree reports all accesses to freed allocated memory |
| Axivion Bauhaus Suite | 6.9.0 | CertC-MEM30 | Detects memory accesses after it deallocation and double memory deallocations |
| CodeSonar | 6.0p0 | ALLOC.UAF | Use after free |
| Coverity | 2017.07 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MET01-Java | The Java Language Specification, [§14.10, "The assert Statement"](http://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.10) [[JLS 2015](https://wiki.sei.cmu.edu/confluence/display/java/Rule+AA.+References" \l "RuleAA.References-JLS15)], states that  assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.  A secondary problem with using assertions for argument checking is that erroneous arguments should result in an appropriate run-time exception (such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException). An assertion failure will not throw an appropriate exception” (Seacord & Lallier, 2015). |

| **Noncompliant Code** |
| --- |
| “The method getAbsAdd() computes and returns the sum of the absolute value of parameters x and y. It lacks argument validation, in violation of [MET00-J. Validate method arguments](https://wiki.sei.cmu.edu/confluence/display/java/MET00-J.+Validate+method+arguments). Consequently, it can produce incorrect results because of integer overflow or when either or both of its arguments are Integer.MIN\_VALUE” (Seacord & Lallier, 2015). |
| public static int getAbsAdd(int x, int y) {    return Math.abs(x) + Math.abs(y);  }  getAbsAdd(Integer.MIN\_VALUE, 1); |

| **Compliant Code** |
| --- |
| “This compliant solution validates the method arguments by ensuring that values passed to Math.abs() exclude Integer.MIN\_VALUE and also by checking for integer overflow:” (Seacord & Lallier, 2015). |
| public static int getAbsAdd(int x, int y) {    if (x == Integer.MIN\_VALUE || y == Integer.MIN\_VALUE) {      throw new IllegalArgumentException();    }    int absX = Math.abs(x);    int absY = Math.abs(y);    if (absX > Integer.MAX\_VALUE - absY) {      throw new IllegalArgumentException();    }    return absX + absY;  } |

|  |
| --- |
| **Principles(s):** “Using assertions to validate method arguments can result in inconsistent computations, runtime exceptions, and control flow vulnerabilities” (Seacord & Lallier, 2015). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| N/A | N/A | N/A | N/A |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | ERR55-CPP | “The C++ Standard, [except.spec], paragraph 8 [[ISO/IEC 14882-2014](https://wiki.sei.cmu.edu/confluence/display/cplusplus/AA.+Bibliography#AA.Bibliography-ISO/IEC14882-2014)], states the following:  A function is said to allow an exception of type E if the constant-expression in its noexcept-specification evaluates to false or its dynamic-exception-specification contains a type T for which a handler of type T would be a match (15.3) for an exception of type E” (Svoboda & Herter, 2020). |

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, a function is declared as nonthrowing, but it is possible for std::vector::resize() to throw an exception when the requested memory cannot be allocated” (Svoboda & Herter, 2020). |
| #include <cstddef>  #include <vector>    void f(std::vector<**int**> &v, **size\_t** s) noexcept(true) {    v.resize(s); // May throw  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the function's noexcept-specification is removed, signifying that the function allows all exceptions” (Svoboda & Herter, 2020). |
| #include <cstddef>  #include <vector>    void f(std::vector<**int**> &v, **size\_t** s) {    v.resize(s); // May throw, but that is okay  } |

|  |
| --- |
| **Principles(s):** “Throwing unexpected exceptions disrupts control flow and can cause premature termination and denial of service” (Svoboda & Herter, 2020). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Unhandled-throw-noexcept | Partially checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-ERR55 |  |
| LDRA tool suite | 9.7.1 | 56 D | Partially Implemented |
| Parasoft C/C++Test | 2020.2 | CERT\_CPP-ERR55-a | Where a function’s declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s) |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Input Output (FIO)** | FIO50-CPP | “Calling std::basic\_ostream<T>::seekp() or std::basic\_istream<T>::seekg() eventually results in a call to std::basic\_filebuf<T>::seekoff() for file stream positioning. Given that std::basic\_iostream<T> inherits from both std::basic\_ostream<T> and std::basic\_istream<T>, and std::fstream inherits from std::basic\_iostream, either function is acceptable to call to ensure the file buffer is in a valid state before the subsequent I/O operation” (Pincar & Gangopadhyay, 2019). |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example appends data to the end of a file and then reads from the same file. However, because there is no intervening positioning call between the formatted output and input calls, the behavior is [undefined](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior)” (Pincar & Gangopadhyay, 2019). |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }      file << "Output some data";    std::string str;    file >> str;  } |

| **Compliant Code** |
| --- |
| “In this compliant solution, the std::basic\_istream<T>::seekg() function is called between the output and input, eliminating the [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior)” (Pincar & Gangopadhyay, 2019). |
| #include <fstream>  #include <string>    void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }      file << "Output some data";      std::string str;    file.seekg(0, std::ios::beg);    file >> str;  } |

|  |
| --- |
| **Principles(s):** “Alternately inputting and outputting from a stream without an intervening flush or positioning call is undefined behavior.”(Pincar & Gangopadhyay, 2019). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-FIO50-a | Do not alternately input and output from a stream without an intervening flush or positioning call |
| Polyspace Bug Finder | R2020a | CERT C++:FIO50-CPP | Checks for alternating input and output from a stream without flush or positioning call (rule fully covered) |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Concurrency | CON54-CPP | “The wait(), wait\_for(), and wait\_until() member functions of the std::condition\_variable class temporarily cede possession of a mutex so that other threads that may be requesting the mutex can proceed. These functions must always be called from code that is protected by locking a mutex. The waiting thread resumes execution only after it has been notified, generally as the result of the invocation of the notify\_one() or notify\_all() member functions invoked by another thread” (Keaton & Gangopadhyay, 2019). |

| **Noncompliant Code** |
| --- |
| “This noncompliant code example nests the call to wait() inside an if block and consequently fails to check the condition predicate after the notification is received. If the notification was spurious or malicious, the thread would wake up prematurely” (Keaton & Gangopadhyay, 2019). |
| #include <condition\_variable>  #include <mutex>    struct Node {    void \*node;    struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element(std::condition\_variable &condition) {    std::unique\_lock<std::mutex> lk(m);      if (list.next == nullptr) {      condition.wait(lk);    }      // Proceed when condition holds.  } |

| **Compliant Code** |
| --- |
| “This compliant solution calls the wait() member function from within a while loop to check the condition both before and after the call to wait()”(Keaton & Gangopadhyay, 2019). |
| #include <condition\_variable>  #include <mutex>    struct Node {    void \*node;    struct Node \*next;  };    static Node list;  static std::mutex m;  static std::condition\_variable condition;    void consume\_list\_element() {    std::unique\_lock<std::mutex> lk(m);      while (list.next == nullptr) {      condition.wait(lk);    }      // Proceed when condition holds.  } |

|  |
| --- |
| **Principles(s): “**Failure to enclose calls to the wait(), wait\_for(), or wait\_until() member functions inside a while loop can lead to indefinite blocking and denial of service (DoS).**”** (Keaton & Gangopadhyay, 2019). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-CON54-a | Wrap functions that can spuriously wake up in a loop |
| Polyspace Bug Finder | R2020a | CERT C++:CON54-CPP | Checks for situations where functions that can spuriously wake up are not wrapped in loop |
| PRQA QA-C++ | 4.4 | 5019 |  |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming (OOP) | [STD-nnn-LLL] | “Do not directly or indirectly invoke a virtual function from a constructor or destructor that attempts to call into the object under construction or destruction. Because the order of construction starts with base classes and moves to more derived classes, attempting to call a derived class function from a base class under construction is dangerous. The derived class has not had the opportunity to initialize its resources, which is why calling a virtual function from a constructor does not result in a call to a function in a more derived class. Similarly, an object is destroyed in reverse order from construction, so attempting to call a function in a more derived class from a destructor may access resources that have already been released” (Dewhurst & Herter, 2020). |

| **Noncompliant Code** |
| --- |
| “In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release()” (Dewhurst & Herter, 2020). |
| struct B {    B() { seize(); }    virtual ~B() { release(); }    protected:    virtual void seize();    virtual void release();  };    struct D : B {    virtual ~D() = default;    protected:    void seize() override {      B::seize();      // Get derived resources...    }      void release() override {      // Release derived resources...      B::release();    }  }; |

| **Compliant Code** |
| --- |
| “In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources” (Dewhurst & Herter, 2020). |
| class B {    void seize\_mine();    void release\_mine();    public:    B() { seize\_mine(); }    virtual ~B() { release\_mine(); }    protected:    virtual void seize() { seize\_mine(); }    virtual void release() { release\_mine(); }  };    class D : public B {    void seize\_mine();    void release\_mine();    public:    D() { seize\_mine(); }    virtual ~D() { release\_mine(); }    protected:    void seize() override {      B::seize();      seize\_mine();    }      void release() override {      release\_mine();      B::release();    }  }; |

|  |
| --- |
| **Principles(s):** ”When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class’s non-static data members, and the object to which the call applies is the object (call it x) under construction or destruction, the function called is the final overrider in the constructor’s or destructor’s class and not one overriding it in a more-derived class.” (Dewhurst & Herter, 2020). |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 | Virtual-call-in-constructor  Invalid\_function\_pointer | Fully Checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-OOP50 |  |
| Clang | 9.7.1 | Clang-analyzer-  Alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| RuleChecker | 20.10 | Virtual-call-in-constructor | Fully checked |

## 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.

Guidance shall be applied to the DevOps process utilizing development environment tools such as Azure DevOps server, pipelines, repos, and issue tracking tools such as Jira. Enforcement can be automated by modifying permissions of the aforementioned tools that impose guidelines on individuals defined by role-based or attribute-based access. Scanning and remediation tools such as SASTs, SCAs, DASTs, IASTs, and RASPs shall also apply granular permissions based on individual roles. On top of tool permissions and settings, internal guidelines that reference best practices from organizations such as OWASP, CWE, Nist, and Sans shall be readily available to all development and relevant teams. Each iteration of the above DevSecOps model shall verify that enforcement is being applied to each stage of the development cycle with the appropriate risk-assessment evaluation.

## 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 |
| --- | --- | --- | --- | --- | --- |
| CON54-CPP | Low | Unlikely | Medium | Low | 3 |
| DCL12-C | Low | Unlikely | High | Low | 3 |
| ERR55-CPP | Low | Likely | Low | Medium | 2 |
| FIO50-CPP | Low | Likely | Medium | Medium | 2 |
| IDS00-J | High | Probable | Medium | High | 1 |
| IDS01-J | High | Probable | Medium | High | 1 |
| MEM30-C | High | Likely | Medium | High | 1 |
| MET01-J | Medium | Probable | Medium | Medium | 2 |
| NUM00-J | Medium | Unlikely | Medium | Low | 3 |
| OOP50-CPP | Low | Unlikely | Medium | Low | 3 |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |

## 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 | Hard Drives, servers, cloud data storage, and personal devices require encryption. This process is in place since attackers who have access to data in rest require an additional step in decrypting data, giving time for owner to respond to attack. |
| Encryption at flight | LAN file transfers, P2P sharing, cloud-sync data, Use transport layer protocols such as HTTPS/TLC, SSH, SFTP, PGP key for encryption. |
| Encryption in use | Build-In Application, No clear text |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Requires verification of the individual’s identity. This process is in place since the identity of an individual identifies their role. |
| Authorization | Requires verification of individual’s access rights and trust. This process is in place since specific individuals have access to specific material and access rights. |
| Accounting | Requires logging of all access and movement throughout the system. This process is in place for the completion of audit logs that ensure that all user access is monitored and stored for future reference. |

**\***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 log

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.

False positives stemming from SAST, DAST, IAST, and other testing tools will be reviewed by a cybersecurity team before result is deemed safe to ignore. The developer will not have access to solely verify that an exception found from scanning tools is deemed safe. The developer can make a request to ignore a flagged threat as a false positive before committing to shared repository. Only the cybersecurity team has the ability to set a flagged threat as a false positive after manual analysis and verification that found threat is in fact a false positive. The cybersecurity team should reference best practices from organizations such as OWASP, CWE, Nist, and Sans in the process of vetting found threat. The cybersecurity team will also write a report and inform the developer of their findings and the justification of their findings. This exception policy does not apply to “Code Smells,” only to “Security Hotspots” and “Vulnerabilities” found within developer source code.

# 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 | 04/09/2021 | Revised Tables to reflect best practices, principles, and testing tools | Robert Worker |  |
| 2.0 | 4/11/2021 | Finalized Risk Assessment | Robert Worker |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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

References

Seacord, R., & Schiela, R. (2018, May 05). Top 10 Secure Coding Practices. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

Data Type – Seacord & Gangopadhyay. (2019, August 30). DCL12-C. Implement abstract data types using opaque types. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/c/DCL12-C.+Implement+abstract+data+types+using+opaque+types>

Data Value – Mertikas, E & Rozenau, M. (2021, Feb 26). NUM00-J. Detect or prevent integer overflow. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow>

String Correctness – Mohindra, D & Snavely, W. (2017, Nov 16). IDS01-J. Normalize strings before validating them. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/java/IDS01-J.+Normalize+strings+before+validating+them>

SQL Injection – Mohindra, D. & Rozenau, R. (2021, Feb 26). IDS00-J. Prevent SQL injection. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection>

Memory Protection – Seacord, R. & Kafka, K. (2021, Feb 28). MEM30-C. Do not access freed memory. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory>

Assertions – Seacord, R. & Lallier, J. (2015, Jun 18). MET01-J. Never use assertions to validate method arguments. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/java/MET01-J.+Never+use+assertions+to+validate+method+arguments>

Exceptions – Svoboda, D & Herter, J. (2020, Sep 25). ERR55-CPP. Honor exception specifications. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR55-CPP.+Honor+exception+specifications>

Input Output (FIO) – Pincar, J & Gangopadhyay, A. (2019, Oct 21). FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/FIO50-CPP.+Do+not+alternately+input+and+output+from+a+file+stream+without+an+intervening+positioning+call>

Concurrency – Keaton, D & Gangopadhyay, A. (2019, Oct 21). CON54-CPP. Wrap functions that can spuriously wake up in a loop. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON54-CPP.+Wrap+functions+that+can+spuriously+wake+up+in+a+loop>

Object Oriented Programming – Dewhurst, S & Herter, J. (2020, Oct 05). OOP50-CPP. Do not invoke virtual functions from constructors or destructors. Retrieved March 21, 2021, from <https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.+Do+not+invoke+virtual+functions+from+constructors+or+destructors>