**Green Pace Developer: Security Policy Guide**



# 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.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Eliminate a vast majority of software vulnerabilities by validating input from all untrusted data sources. This applies to most external data sources, such as network interfaces, command line arguments, user-controlled files, and user input. |
| 1. Heed Compiler Warnings | Use the highest warning level available when compiling code and eliminate warnings by modifying the code. Utilize static and dynamic testing methods to detect and eliminate security flaws. |
| 1. Architect and Design for Security Policies | Design software to implement and enforce security policies. Creating an architecture with security in mind will prevent vulnerabilities further down the road. |
| 1. Keep It Simple | Keep the design as simple as possible. This promotes readability and maintainability that aids implementation, configuration, and use. This also helps to maintain security and to detect and correct vulnerabilities. |
| 1. Default Deny | Access is based on permission rather than exclusion. By default, access is denied, and only under certain conditions is access permitted. |
| 1. Adhere to the Principle of Least Privilege | Processes should be carried out with the least set of privileges necessary to complete the job. An example of this is that users are only given the level of access appropriate to complete their job functions. This reduces exploitable, unauthorized access to higher levels of the code. |
| 1. Sanitize Data Sent to Other Systems | All data passed to complex subsystems such as relational databases, command shells, and commercial off-the-shelf components must be sanitized. This prevents unauthorized access that would be missed by input validation. |

|  |  |
| --- | --- |
| 1. Practice Defense in Depth | Utilize multiple defense strategies to mitigate risk. This is done so that in the event one security measure fails, another layer is in place to protect against the attack. This prevents security flaws from becoming an exploitable vulnerability. |
| 1. Use Effective Quality Assurance Techniques | Utilizing quality assurance techniques, such as fuzz testing, penetration testing, and source code audits, can be effective in identifying and eliminating vulnerabilities. Independent security reviews performed by external reviewers can also be beneficial. |
| 1. Adopt a Secure Coding Standard | Developing and applying a secure coding standard for the specific development language and platform can assist in mitigating vulnerabilities and promote consistent, secure code throughout all aspects of development. |

### 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** | **Include the appropriate type information in function declarators** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Function declarators must be declared with the appropriate type information, including a return type and parameter list. If type information is not properly specified in a function declarator, the compiler cannot properly check function type information. When using standard library calls, the easiest (and preferred) way to obtain function declarators with appropriate type information is to include the appropriate header file. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the identifier-list form for parameter declarations |
| int max(a, b)  int a, b;  {  return a > b ? a : b;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, int is the type specifier, max(int a, int b) is the function declarator, and the block within the curly braces is the function body |
| int max(int a, int b) {  return a > b ? a : b;  } |

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

| **Principles(s):** Keep It Simple – Declaring the appropriate data types when declaring a function helps to ensure functionality and promotes readability and maintainability. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Function-prototype  Implicit-function-declaration | Partially checked |
| ÉCLAIR | 1.2 | CC2.DCL07 | Fully implemented |
| LDRA tool suite | 9.7.1 | 21 S  135 S  170 S | Fully implemented |
| PC-lint Plus | 1.4 | 718, 746, 936, 9074 | Fully supported |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Use only explicitly signed or unsigned char type for numeric values** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Use only signed char and unsigned char types for the storage and use of numeric values because it is the only portable way to guarantee the signedness of the character types. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the char-type variable c may be signed or unsigned. Assuming 8-bit, two's complement character types, this code may print out either i/c = 5 (unsigned) or i/c = -17 (signed). It is much more difficult to reason about the correctness of a program without knowing if these integers are signed or unsigned. |
| char c = 200;  int i = 1000;  printf("i/c = %d\n", i/c); |

| **Compliant Code** |
| --- |
| In this compliant solution, the variable c is declared as unsigned char. The subsequent division operation is now independent of the signedness of char and consequently has a predictable result. |
| unsigned char c = 200;  int i = 1000;  printf("i/c = %d\n", i/c); |

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

| **Principles(s):** Validate Input Data / Effective quality assurance techniques. Properly validating input can prevent this error from occurring, while effective quality assurance can catch the error and allow for it to be corrected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.TYPE.IOT | Inappropriate operand type |
| ECLAIR | 1.2 | CC2.INT07 | Fully implemented |
| Polyspace Bug Finder | R2023a | CERT C: Rec. INT07-C | Checks for use of plain char type for numeric value (rec. fully covered) |
| PC-lint Plus | 1.4 | 9112 | Fully supported |

#### 

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not inadvertently truncate a string** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Alternative functions that limit the number of bytes copied are often recommended to mitigate buffer overflow vulnerabilities. These functions truncate strings that exceed the specified limits. Additionally, some functions, such as strncpy(), do not guarantee that the resulting character sequence is null-terminated. Unintentional truncation results in a loss of data and in some cases leads to software vulnerabilities. |

| **Noncompliant Code** |
| --- |
| The standard functions strncpy() and strncat() copy a specified number of characters n from a source string to a destination array. In the case of strncpy(), if there is no null character in the first n characters of the source array, the result will not be null-terminated and any remaining characters are truncated. |
| char \*string\_data;  char a[16];  /\* ... \*/  strncpy(a, string\_data, sizeof(a)); |

| **Compliant Code** |
| --- |
| Either the strcpy() or strncpy() function can be used to copy a string and a null character to a destination buffer, provided there is enough space. The programmer must be careful to ensure that the destination buffer is large enough to hold the string to be copied and the null byte to prevent errors, such as data truncation and buffer overflow. |
| char \*string\_data = NULL;  char a[16];    /\* ... \*/    if (string\_data == NULL) {  /\* Handle null pointer error \*/  }  else if (strlen(string\_data) >= sizeof(a)) {  /\* Handle overlong string error \*/  }  else {  strcpy(a, string\_data);  } |

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

| **Principles(s):** Validate input / QA techniques. Validating input ensures that strings aren’t inadvertently truncated. QA techniques ensure that inadvertent string truncation doesn’t take place and that proper protections are in place. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | MISC.MEM.NTERM | No space for null terminator |
| GCC | 8.1 | -Wstringop-truncation | Detects string truncation by strncat and strncpy |
| LDRA tool suite | 9.7.1 | 115 S, 44 S | Partially implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-STR03-a | Avoid overflow due to reading a not zero terminated string |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software vulnerability. As a result, it is necessary to sanitize all string data passed to complex subsystems so that the resulting string is innocuous in the context in which it will be interpreted. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example is taken from [VU#881872], a vulnerability in the Sun Solaris TELNET daemon (in.telnetd) that allows a remote attacker to log on to the system with elevated privileges. The vulnerability in in.telnetd invokes the login program by calling execl(). This call passes unsanitized data from an untrusted source (the USER environment variable) as an argument to the login program. An attacker, in this case, can gain unauthenticated access to a system by setting the USER environment variable to a string, which is interpreted as an additional command-line option by the login program. This kind of attack is called argument injection. |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")),  0); |

| **Compliant Code** |
| --- |
| This compliant solution inserts the "--" (double dash) argument before the call to getenv("USER") in the call to execl().Because the login program uses the POSIX getopt() function to parse command-line arguments, and because the "--" option causes getopt() to stop interpreting options in the argument list, the USER variable cannot be used by an attacker to inject an additional command-line option. This is a valid means of sanitizing the untrusted user data in this context because the behavior of the interpretation of the resulting string is rendered innocuous. The call to execl() is not susceptible to command injection because the shell command interpreter is not invoked. |
| (void) execl(LOGIN\_PROGRAM, "login",  "-p",  "-d", slavename,  "-h", host,  "-s", pam\_svc\_name,  "--",  (AuthenticatingUser != NULL ? AuthenticatingUser :  getenv("USER")), 0); |

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

| **Principles(s):** Sanitize data sent to other systems / Practice defense in depth. Sanitizing data can eliminate vulnerabilities and prevent adverse effects. Defense in depth implements safe guards against vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-STR02-a  CERT\_C-STR02-b  CERT\_C-STR02-c | Protect against command injection  Protect against file name injection  Protect against SQL injection |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Polyspace Bug Finder | R2023a | CERT C: Rec. STR02-C | Checks for: Execution of externally controlled command. Command executed from externally controlled path. Library loaded from externally controlled path.  Rec. partially covered |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Do not access freed memory** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Evaluating a pointer—including dereferencing the pointer, using it as an operand of an arithmetic operation, type casting it, and using it as the right-hand side of an assignment—into memory that has been deallocated by a memory management function is undefined behavior. Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable vulnerabilities. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, std::string::c\_str() is being called on a temporary std::string object. The resulting pointer will point to released memory once the std::string object is destroyed at the end of the assignment expression, resulting in undefined behavior when accessing elements of that pointer. |
| #include <string>    std::string str\_func();  void display\_string(const char \*);    void f() {  const char \*str = str\_func().c\_str();  display\_string(str); /\* Undefined behavior \*/  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a local copy of the string returned by str\_func() is made to ensure that string str will be valid when the call to display\_string() is made. |
| #include <string>    std::string str\_func();  void display\_string(const char \*s);    void f() {  std::string str = str\_func();  const char \*cstr = str.c\_str();  display\_string(cstr); /\* ok \*/  } |

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

| **Principles(s):** Validate input data / Architect and design for security policies / Effective QA techniques. All three of these principles can help to ensure that freed memory isn’t access. Input validation ensures input doesn’t attempt to access freed memory, while the design of the program does the same. Effective QA helps to identify potential access to freed memory so that it can be corrected. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 23.04 | Dangling\_pointer\_use | Supported  Astree reports all accesses to freed allocated memory. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-MEM30 | Detects memory accesses after its deallocation and double memory deallocations. |
| CodeSonar | 7.3p0 | 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** | **Use a static assertion to test the value of a constant expression** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities. The runtime assert() macro has some limitations, however, in that it incurs a runtime overhead and because it calls abort(). Consequently, the runtime assert() macro is useful only for identifying incorrect assumptions and not for runtime error checking. As a result, runtime assertions are generally unsuitable for server programs or embedded systems. |

| **Noncompliant Code** |
| --- |
| This noncompliant code uses the assert() macro to assert a property concerning a memory-mapped structure that is essential for the code to behave correctly. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    int func(void) {  assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int));  } |

| **Compliant Code** |
| --- |
| This portable compliant solution uses static\_assert. Static assertions allow incorrect assumptions to be diagnosed at compile time instead of resulting in a silent malfunction or runtime error. Because the assertion is performed at compile time, no runtime cost in space or time is incurred. An assertion can be used at file or block scope, and failure results in a meaningful and informative diagnostic error message. |
| #include <assert.h>    struct timer {  unsigned char MODE;  unsigned int DATA;  unsigned int COUNT;  };    static\_assert(sizeof(struct timer) == sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int),  "Structure must not have any padding"); |

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

| **Principles(s):** Effective QA techniques / Secure coding standard. Using assertations is a valuable tool for testing and QA. Using assertations also checks for vulnerabilities and promotes secure coding. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Misc-static-assert | Check by clang-tidy |
| CodeSonar | 7.3p0 | (customization) | Users can implement a custom check that reports uses of the assert() macro |
| ÉCLAIR | 1.2 | CC2.DCL03 | Fully implemented |
| LDRA tool suite | 9.7.1 | 44 S | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | All exceptions thrown by an application must be caught by a matching exception handler. Even if the exception cannot be gracefully recovered from, using the matching exception handler ensures that the stack will be properly unwound and provides an opportunity to gracefully manage external resources before terminating the process. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, neither f() nor main() catch exceptions thrown by throwing\_func(). Because no matching handler can be found for the exception thrown, std::terminate() is called. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  f();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the main entry point handles all exceptions, which ensures that the stack is unwound up to the main() function and allows for graceful management of external resources. |
| void throwing\_func() noexcept(false);    void f() {  throwing\_func();  }    int main() {  try {  f();  } catch (...) {  // Handle error  }  } |

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

| **Principles(s):** Architect and design for security policies / Effective QA techniques / Adopt a secure coding standard. Software should be designed to handle all exceptions. Effective QA helps to ensure all exceptions are handled, and if one is not handled it can be corrected. Ensuring that all exceptions are handled when code is initially written can be achieved is a secure coding standard is implement that requires such. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| CodeSonar | 7.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Parasoft C/C++ test | 2022.2 | 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 |
| Polyspace Bug Finder | R2023a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Never hard code sensitive information** |
| --- | --- | --- |
| **Sensitive Information** | [STD-008-CPP] | Hard coding sensitive information, such as passwords or encryption keys can expose the information to attackers. Anyone who has access to the executable or dynamic library files can examine them for strings or other critical data, revealing the sensitive information. Leaking data protected by International Traffic in Arms Regulations (ITAR) or the Health Insurance Portability and Accountability Act (HIPAA) can also have legal consequences. Consequently, programs must not hard code sensitive information. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example must authenticate to a remote service with a code, using the authenticate() function declared below. It passes the authentication code to this function as a string literal. The authentication code exists in the program's binary executable and can be easily discovered. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  if (!authenticate("correct code")) {  printf("Authentication error\n");  return -1;  }    printf("Authentication successful\n");  // ...Work with system...  return 0;  } |

| **Compliant Code** |
| --- |
| This compliant solution requires the user to supply the authentication code, and securely erases it when done, using memset\_s(), an optional function provided by C11's Annex K. Alternatively, the program could read the authentication code from a file, letting file system security protect the file and the code from untrusted users. |
| /\* Returns nonzero if authenticated \*/  int authenticate(const char\* code);    int main() {  #define CODE\_LEN 50  char code[CODE\_LEN];  printf("Please enter your authentication code:\n");  fgets(code, sizeof(code), stdin);  int flag = authenticate(code);  memset\_s(code, sizeof(code), 0, sizeof(code));  if (!flag) {  printf("Access denied\n");  return -1;  }  printf("Access granted\n");  // ...Work with system...  return 0;  } |

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

| **Principles(s):** Practice defense in depth / Effective QA techniques / Adopt a secure coding standard. Not hard coding information protects the important information from malicious attacks. Effective QA techniques can catch and correct the mistake, while adopting a secure coding standard can discourage programmers from hard coding sensitive information. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | HARDCODED.AUTH  HARDCODED.DNS  HARDCODED.KEY  HARDCODED.SALT  HARDCODED.SEED | Hardcoded Authentication  Hardcoded DNS Name  Hardcoded Crypto Key  Hardcoded Crypto Salt  Hardcoded Seed in PRNG |
| Parasoft C/C++ test | 2022.2 | CERT\_C-MSC41-a | Do not hard code string literals |
| PC-lint Plus | 1.4 | 2460 | Assistance provided: reports when a literal is provided as an argument to a function parameter with the ‘noliteral’ argument Semantic; several Windows API functions are marked as such and the ‘-sem’ option can apply it to other functions as appropriate |
| Polyspace Bug Finder | R2023a | CERT C: Rule MSC41-C | Checks for hard coded sensitive data (rule partially covered) |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Close files when they are no longer needed** |
| --- | --- | --- |
| **File Management** | [STD-009-CPP] | A call to the std::basic\_filebuf<T>::open() function must be matched with a call to std::basic\_filebuf<T>::close() before the lifetime of the last pointer that stores the return value of the call has ended or before normal program termination, whichever occurs first. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a std::fstream object file is constructed. The constructor for std::fstream calls std::basic\_filebuf<T>::open(), and the default std::terminate\_handler called by std::terminate() is std::abort(), which does not call destructors. Consequently, the underlying std::basic\_filebuf<T> object maintained by the object is not properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  std::terminate();  } |

| **Compliant Code** |
| --- |
| In this compliant solution, std::fstream::close() is called before std::terminate() is called, ensuring that the file resources are properly closed. |
| #include <exception>  #include <fstream>  #include <string>    void f(const std::string &fileName) {  std::fstream file(fileName);  if (!file.is\_open()) {  // Handle error  return;  }  // ...  file.close();  if (file.fail()) {  // Handle error  }  std::terminate();  } |

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

| **Principles(s):** Architect and design for security policies / Effective QA techniques. Failing to close a file after its use can have adverse effects. Designing the code to ensure files are closed after use can prevent these, and effective QA techniques can identify these errors and correct them accordingly. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2022.2 | CERT\_CPP-FIO51-a | Ensure resources are freed |
| Klocwork | 2023.1 | RH.LEAK |  |
| Polyspace Bug Finder | R2023a | CERT C++: FIO51-CPP | Checks for resource leak (rule partially covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not declare more that one variable per declaration** |
| --- | --- | --- |
| **Variable Declaration** | [STD-010-CPP] | Every declaration should be for a single variable, on its own line, with an explanatory comment about the role of the variable. Declaring multiple variables in a single declaration can cause confusion regarding the types of the variables and their initial values. If more than one variable is declared in a declaration, care must be taken that the type and initialized value of the variable are handled correctly. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a programmer or code reviewer might mistakenly believe that both i and j have been initialized to 1. In fact, only j has been initialized, and i remains uninitialized. |
| int i, j = 1; |

| **Compliant Code** |
| --- |
| In this compliant solution, it is readily apparent that both i and j have been initialized to 1. |
| int i = 1;  int j = 1; |

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

| **Principles(s):** Architect and design for security policies / Keep it simple / Effective QA techniques / Adopt a secure coding standard. Designing software that only declares one variable per declaration prevents confusion and promotes readability and maintainability, which also keeps the code simple. Effective QA techniques can identify this and correct it before it becomes a more serious issue. Adopting a secure coding standard can discourage this type of coding practice. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.3p0 | LANG.STRUCT.DECL.ML | Multiple Declarations on Line |
| ÉCLAIR | 1.2 | CC2.DCL04 | Fully implemented |
| Parasoft C/C++ test | 2022.2 | CERT\_C-DCL04-a | Each variable should be declared in a separate declaration statement |
| PC-lint Plus | 1.4 | 9146 | Partially supported: exceptions not supported |

### 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.

Automation can be implemented in Verify and Test by automating security and vulnerability testing and QA. It can also be implemented in Monitor and detect. Automation can be used to conduct static testing as well as dynamic testing, frequently checking for vulnerabilities and security risks.

### 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 | Unlikely | Low | Low | 3 |
| STD-002-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-003-CPP | Medium | Probable | Medium | Medium | 2 |
| STD-004-CPP | High | Likely | Medium | High | 1 |
| STD-005-CPP | High | Likely | Medium | High | 1 |
| STD-006-CPP | Low | Unlikely | High | Low | 3 |
| STD-007-CPP | Low | Probable | Medium | Medium | 3 |
| STD-008-CPP | High | Probable | Medium | High | 1 |
| STD-009-CPP | Medium | Unlikely | Medium | Low | 3 |
| STD-010-CPP | Low | Unlikely | Low | 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 in rest aims to prevent unauthorized access to data on the disk. Encrypting this data adds an additional layer of security, as even if the data is accessed, it will be unusable without the encryption key. |
| Encryption at flight | Encryption in flight is the practice of encrypting data while it is being transmitted. This ensures that if the data is intercepted during transmission, it is unusable by the attacker. |
| Encryption in use | Encryption in use is the practice of encrypting data while it is being used. If an attacker were to gain access to unencrypted data in use, that data could be used to enable access to both data at rest and data in flight. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the practice of confirming a user’s access to a network or system, usually via a user login. Newer, more advanced systems require two factor authentication, such as a user login combined with biometrics or a single use access code. |
| Authorization | Authorization refers to the specific level of access granted to a user. This includes privileges to make changes to the database, such as write or delete data, as well as add users. Most users will simply be able to read the information on the database, while the ability to make changes will be limited to a select few people. |
| Accounting | Accounting is the process of monitoring user activity. This aids in identifying unauthorized access to the system. User activity is monitored to ensure that they are only access information that they are authorized to access. |

**\***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 |  |
| 1.0.1 | 05/20/2023 | Milestone: Coding Standards | Andrew Riley | Andrew Riley |
| 1.1 | 06/11/2023 | Project One: Security Policy | Andrew Riley | Andrew Riley |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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