**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. |
| --- | --- |
| 1. ValidateInput Data | Input data validation is used to ensure data entering the software or system is correctly formed. Validation prevents noncompliant and malformed data from entering the software, system, or database. All untrusted and potentially untrusted data should be processed using input validation. |
| 1. Heed Compiler Warnings | When developing software, compilers will alert the developer of warnings and errors in the code. It is important to understand and fix these warnings before release. These errors and warnings are designed to alert the developer of errors and defects that may be difficult to identify on runtime. |
| 1. Architect and Design for Security Policies | Architect and design for security policies refers to software architecture designed to implement security countermeasures. The security policy statement was created to describe how entities in a system or software interact with one another. The operations of these entities are also described in this document. The level of protection is also outlined in this policy along with action to be taken when security requirements are not met. |
| 1. Keep It Simple | It is important to not overcomplicate software, systems, and their respective designs. Avoiding complexity creates an understandable and usable system or software. |
| 1. Default Deny | Default denying is a principle that denies actions, abilities, or permissions that are not specifically stated as allowed. Denying actions not outlined as allowed prevents unforeseen cases that could cause unintended or malicious outcomes. |
| 1. Adhere to the Principle of Least Privilege | It is important to give users the least permission necessary to complete their designated tasks. Following this principle keeps users within their use cases and helps prevent unintended or malicious outcomes. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data refers to cleaning data before it is sent to other systems. For example, data should remove unwanted characters from the data before it is processed. Whenever a trust boundary is crossed, the data crossing this boundary should be sanitized prior to crossing the boundary. This principle can prevent malicious exploitations such as buffer overflows and SQL injections. |
| 1. Practice Defense in Depth | The principles of Defense in Depth require a system to implement multiple, unrelated layers of security for specific attacks. The depth of each layer can be determined by the organization for their specific needs. The justification for multiple layers is that if one layer fails to prevent an attack, another layer should be able to defend it. |
| 1. Use Effective Quality Assurance Techniques | It is important to implement and utilize effective QA techniques. Software and systems should be tested early and often by developers and testers. A process to test the system and software should be created and communicated throughout the development team. Rules and requirements for the software or system should be communicated as well. |
| 1. Adopt a Secure Coding Standard | Secure coding standards are guidelines created by several organizations to prevent vulnerabilities in software. Adopting one or more of these standards will help understand and prevent vulnerabilities in the development phase. Some organizations that have created secure coding standards are CERT, CVE, NVD, OWASP, and DISA STIG. |

### 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** |
| --- | --- | --- |
| **Data Type** | [DCL-012-C] | Abstract data types are more effective when used with private data types. This can yield poor portability of the software. |

| **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. This violates the software engineering principle of information hiding and data encapsulation leading to less portable 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);  /\* ...etc. \*/ |

| **Compliant Code** |
| --- |
| In the external header file, the string\_mx type is defined to the instance of struct string\_mx. In the internal header file the struct string\_mx is fully defined but not visible to the user. |
| /\* External instance \*/  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);  /\* ...etc. \*/  /\* Internal instance \*/  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):** Default Deny: We should deny access to abstract data types by making them private. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | High | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CERTC-DCL12 | Checks for redundant and unreachable code regardless of depth |
| LDRA Tool Suite | 9.7.1 | 104 D | Includes capabilities for requirements traceability, test management, coding standards compliance, code quality review, code coverage analysis, data-flow and control-flow analysis, unit/integration/target testing, and certification and regulatory support |
| Parasoft C/C++test | 2022.2 | CERTC-DCL12-A | Finds runtime defects, stability issues, memory leaks, null pointers, uninitialized memory, and buffer overflows |
| Polyspace Bug Finder | R2023.b | CERTC: Rec. DCL12-C | Checks structure or union object implementation visibility in file where a pointer to this object is not dereferenced |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| **Data Value** | [INT-030-C] | When the limit size of storage for a data type is reached, the value will wrap to the other end of the limit. This should be addressed and avoided especially when using pointer arithmetic. |

| **Noncompliant Code** |
| --- |
| The noncompliant code uses two integers, wrapping can occur during the operation usum function. The value could then be used to allocate insufficient memory for another operation. |
| void func(unsigned int ui\_a, unsigned int ui\_b)  {  unsigned int usum = ui\_a + ui\_b;  /\* ...etc. \*/  } |

| **Compliant Code** |
| --- |
| A precondition test can be added to ensure the operands of the addition does not result in wrapping. |
| void func(unsigned int ui\_a, unsigned int ui\_b)  {  if (UINT\_MAX – ui\_a < ui\_b)  {  /\* Handle the overflow \*/  }  else  {  unsigned int usum = ui\_a + ui\_b;  }  /\* ...etc. \*/  } |

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

| **Principles(s):** Validate Input Data: Integer wrapping commonly occurs with the use of input data, data should always be validated to not exceed the limits of the data type, this ensures no unsigned integer wrapping occurs |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | High | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrèe | 23.10 | Integer overflow | Fully checks for integer overflow |
| Axivion Bauhaus Suite | 7.2.0 | CERTC-INT30 | Checks for redundant and unreachable code |
| Parasoft C/C++test | 2022.2 | CERTC-INT30-A  CERTC-INT30-B  CERTC-INT30-C | Checks for integer overflow and integer underflows arising from operators “+”, “-”, “\*”, and “<<” |
| Polyspace Bug Finder | R2023.b | CERTC-: Rule NT30-C | Checks for unsigned integer overflow and unsigned integer constant overflow |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Do not modify string literals** |
| --- | --- | --- |
| **String Correctness** | [STR-030-C] | String literals are stored in read-only memory, editing this can lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| The char pointer str is initialized to the address of a string literal, “string literal”. Modifying the string literal leads to undefined behavior. |
| char \*str = “string literal”;  str[0] = ‘S’; |

| **Compliant Code** |
| --- |
| Initializing the string literal to define the size of a char array allows the char array, and therefore the string literal, to 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):** Adopt a Secure Coding Standard: By implementing a secure coding standard the modification of string literals will be avoided, chosen standards will include rules such as STR-030 |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | High | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.7.1 | 157 S | Partial implementation |
| Parasoft C/C++test | 2022.2 | CERTC-STR30-A  CERTC-STR30-B | Checks for modified string literals |
| PC-Lint Plus | 2.0 | 489, 1776 | Partial support |
| Polyspace Bug Finder | R2023.b | CERTC: Rule STR30-B | Checks for writing to constant qualified objects |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize data passed to complex subsystems** |
| --- | --- | --- |
| **SQL Injection** | [STR-002-C] | String data passed into another system can lead to unintended actions and vulnerabilities. All data should be sanitized before being passed to subsystems such as a SQL database. |

| **Noncompliant Code** |
| --- |
| This code takes an email as an input string. This input string is used as an argument in the system() function call. If a SQL injection is passed by the user, sensitive data can be accessed by the user. |
| sprintf(buffer, “/bin/mail %s < /tmp/email”, addr);  system(buffer); |

| **Compliant Code** |
| --- |
| Creating a list of acceptable characters will create a reference to check the input string against. This prevents characters that may be used for SQL injections to be removed prior to the data being passed. |
| static char ok\_chars[] = “abcdefghijklmnopqrstuvwxyz”  “ABCDEFGHIJKLMNOPQRSTUVWXYZ”  “0123456789\_-.@”;  char user\_data[] = “Illegal char 1:} Illegal 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):** Sanitize Data Sent to Other Systems: sanitizing data prior to use prevents SQL injections, the area that sanitizes input data will use different system which are not exploitable whereas the area where the data is uses may be exploitable |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Klocwork | 2023.3 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| LDRA Tool Suite | 9.7.1 | 108 D, 109 D | Partial implementation |
| Parasoft C/C++test | 2022.2 | CERTC-STR02-A  CERTC-STR02-B  CERTC-STRO2-C | Checks for command injection, file name injection, and SQL injection |
| Polyspace Bug Finder | R2023.b | CERTC: Rec. STR02-C | Checks for externally controlled command execution, execution of command from externally controlled path, and libraries loaded from externally controlled paths |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Explicitly construct and destruct objects when manually managing objects** |
| --- | --- | --- |
| **Memory Protection** | [MEM-053-CPP] | Objects not using the new operator to allocate sufficient memory are managed manually and should be manually deallocated and destroyed. An object that is used outside of its lifespan can lead to undefined behavior and errors. |

| **Noncompliant Code** |
| --- |
| A manual management of memory is utilized due to the user provided construction with a call to malloc(). The constructor for the object is never called. This results in undefined behavior when the class is later accessed by 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 in the compliant code block. |
| #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);  } |

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

| **Principles(s):** Keep It Simple: Explicit statements to construct or destruct objects simplifies the process and prevents errors |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Medium | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.3 | C++4761, C++4762, C++4766, C++4767 |  |
| Parasoft C/C++test | 2022.2 | CERTCPP-MEM53-A | Checks for invocation, malloc/realloc, for objects that have constructors |
| PVS-Studio | 7.27 | V749 |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Understand the termination behavior of assert() and abort()** |
| --- | --- | --- |
| **Assertions** | [ERR-006-C] | Since assert() calls abort(), cleanup functions registered with atexit() are not called. This can lead to errors in the correct termination of a program. |

| **Noncompliant Code** |
| --- |
| A cleanup function is called before the program exits. If the assert function fails, the program will exit before cleanup occurs. |
| void cleanup(void)  {  /\* Delete temp files, restore states, etc. \*/  }  int main(void)  {  if (atexit(cleanup) != 0)  {  /\* Handle error \*/  }  /\* ...rest of program... \*/  assert(/\*Something unexpected didn’t happen\*/);  /\* ...rest of program... \*/  } |

| **Compliant Code** |
| --- |
| If statements are utilized instead of assert() to allow cleanup processes to occur with proper termination routines. |
| void cleanup(void)  {  /\* Delete temp files, restore states, etc. \*/  }  int main(void)  {  if (atexit(cleanup) != 0)  {  /\* Handle error \*/  }  /\* ...rest of program... \*/  if (/\*Something unexpected happened\*/)  {  exit(EXIT\_FAILURE);  }  /\* ...rest of program... \*/  } |

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

| **Principles(s):** Use Effective Quality Assurance Techniques: Understanding the assert() and abort() functions are critical to creating effective quality assurance techniques, assert() should be called to test software in early iterations, proper implementation of assertions are key |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Low | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Compass/ROSE |  |  | Detects violations of this rule involving abort() |
| LDRA Tool Suite | 9.7.1 | 44 S | Enhanced enforcement of this rule |
| Parasoft C/C++test | 2022.2 | CERTC-ERR06-A | Checks for improper use of assertions |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | [ERR-051-CPP] | All exceptions thrown must be caught and handled by a matching exception handler. If not, the stack may not unwind correctly due to abort() being called. This could result in destructors not being called. |

| **Noncompliant Code** |
| --- |
| Neither f() or main() catch and handle exceptions thrown by throwing\_func(). |
| void throwing\_func() noexcept(false);  void f()  {  throwing\_func();  }  int main()  {  f();  } |

| **Compliant Code** |
| --- |
| All exceptions are caught and handled. This allows the stack to properly unwind. |
| void throwing\_func() noexcept(false);  void f()  {  throwing\_func();  }  int main()  {  try  {  f();  }  catch (/\*error\*/)  {  /\* Handle error \*/  }  } |

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

| **Principles(s):** Heed Compiler Warnings: Exceptions often relate to the compiler being used, catching excpetions is important in allowing the compiler to safely and properly unwind the stack |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Medium | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.7.1 | 527 S | Partial implementation |
| Parasoft C/C++test | 2022.2 | CERTCPP-ERR51-A  CERTCPP-ERR51-B | Checks for exceptions being caught and that each exception thrown has a handler of a compatible type |
| Polyspace Bug Finder | R2023.b | CERTC++: ERR51-CPP | Checks for unhandled exceptions |
| PRQA QA-C++ | 4.4 | 4035, 4036, 4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Do not delete a polymorphic object without a virtual destructor** |
| --- | --- | --- |
| Polymorphic Object | [OOP-052-CPP] | Deleting an object using a pointer to a type that does not have a virtual destructor can result in undefined behavior. |

| **Noncompliant Code** |
| --- |
| The implicitly declared destructor is not declared as virtual with other virtual functions. |
| struct Base  {  virtual void f();  };  struct Derived : Base {};  void f()  {  Base \*b = new Derived();  /\* ...rest of function... \*/  delete b;  } |

| **Compliant Code** |
| --- |
| The destructor for the Base structure is explicitly declared as a virtual destructor. This ensures that undefined behavior will not occur when using the polymorphic delete operation. |
| struct Base  {  virtual ~Base() = default;  virtual void f();  };  struct Derived : Base {};  void f()  {  Base \*b = new Derived();  /\* ...rest of function... \*/  delete b;  } |

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

| **Principles(s):** Adopt a Secure Coding Standard: implementing a coding standard will help avoid erros such as not calling a virtual destructor when dealing with polymorphic objects |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | High | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA Tool Suite | 9.7.1 | 303 S | Partial Implementation |
| Parasoft C/C++test | 2022.2 | CERTCPP-OOP52-A | Ensure definition of virtual destructor in classes used as base classes which include virtual functions |
| PRQA QA-C++ | 4.4 | 3402, 3403, 3404 |  |
| Polyspace Bug Finder | R2023.b | CERTC++: OOP52-CPP | Checks for situations where a class includes virtual functions but does not include a virtual destructor |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Use correct integer precisions** |
| --- | --- | --- |
| Integer Precision | [INT-035-C] | Integer sizes are contributed to by padding bits, this does not inherently carry over to integer precision. This can cause incorrect assumptions about the numeric range of these types. |

| **Noncompliant Code** |
| --- |
| If this code block is run on a platform where unsigned int has one or more padding bits the value of exp can be too large. |
| #include <limits.h>  unsigned int pow2(unsigned int exp)  {  if (exp >= sizeof(unsigned int) \* CHAR\_BIT)  {  /\*Handle error\*/  }  return 1 << exp;  } |

| **Compliant Code** |
| --- |
| Using the popcount() function allows the precision of any integer type to be determined. This includes signed and unsigned integer types. |
| #include <stddef.h>  #include <stdint.h>  /\* Returns the number of set bits \*/  size\_t popcount(uintmax\_t num)  {  size\_t precision = 0;  while (num != 0)  {  if (num % 2 == 1)  {  precision ++;  }  num >>= 1;  }  return precision;  }  #define PRECISION(umax\_value) popcount(umax\_value) |

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

| **Principles(s):** Validate Input Data: Assumptions in input data can lead to errors, focus on using less assumption when data handling such as integer precision |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Low | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrèe | 23.10 |  | Fully supported, reports overflows due to insufficient precision |
| Helix QAC | 2023.3 | C0582  C++3115 |  |
| Parasoft C/C++test | 2022.2 | CERTC-INT35-A | Correct integer precision is necessary when checking the right hand operand of the shift operator |
| Polyspace Bug Finder | R2023.b | CERTC: Rule INT35-C | Checks for instances where integer precisions are exceeded |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Do not confuse narrow and wide character strings and functions** |
| --- | --- | --- |
| Buffer Overflow | [STR-038-C] | Passing narrow to wide or wide to narrow strings and string functions could lead to undefined behavior. |

| **Noncompliant Code** |
| --- |
| This code block uses strncpy() function to attempt to copy up to 10 wide characters. Since the wide characters may contain null bytes the copy operation could end earlier than anticipated. This can result in truncation of the wide string. |
| #include <stddef.h>  #include <string.h>  void func(void)  {  wchar\_t wide\_str1[] = L”0123456789”;  wchar\_t wide\_str2[] = L”0000000000”;  strncpy(side\_str1, wide\_str1, 10);  } |

| **Compliant Code** |
| --- |
| Using wcsncpy() for wide character strings and strncpy() for narrow character strings eliminates the chance of data truncation and buffer overflow. |
| #include <string.h>  #include <wchar.h>  void func(void)  {  wchar\_t wide\_str1[] = L”0123456789”;  wchar\_t wide\_str2[] = L”0000000000”;  /\* Use the proper width function \*/  wcsncpy(wide\_str2, wide\_str1, 10);  char narrow\_str1[] = “0123456789”;  char narrow\_str2[] = “0000000000”;  /\* Use the proper width function \*/  strncpy(narrow\_str2, narrow\_str1, 10);  } |

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

| **Principles(s):** Keep It Simple: Overly complex programs can create confusion regarding wide and narrow character strings, this confusion can be avoided through simplification |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | High | Low | High | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2023.3 | C0432  C++0403 |  |
| Parasoft C/C++test | 2022.2 | CERTC-STR38-A | Mitigate confusion of wide and narrow strings and string functions |
| PC-Lint Plus | 2.0 | 2452, 2480, 2481 | Partial support, reports illegal conversions involving pointers to char or wchar\_t and byte/wide-oriented stream inconsistencies |
| Polyspace Bug Finder | R2023.b | CERTC: Rule STR38-C | Checks for misuse of narrow or wide character strings |

### 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 will help enforce standards in both phases of DevOps. In the pre-production phase automation can be utilized in the verify and test step. Unit testing tools such as QUnit can be implemented to create testable sections of code. Pre-production testing can be used to detect and patch vulnerabilities such as overflow, injections, and ensure best coding practices are followed. In the production phase, penetration tests can be utilized to ensure product quality and security consistency. In the monitor and detect step of the production phase, accounting can be implemented creating records and notifications of unexpected events and unauthorized access. This can ensure depth in the system’s security coverage and allow missed vulnerabilities to be identified and addressed swiftly.

### 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 | Low | Medium | High | 2 |
| DCL-012-C | Low | Low | High | Low | 3 |
| INT-030-C | High | High | High | High | 2 |
| STR-030-C | Low | High | Low | Medium | 2 |
| STR-002-C | High | High | Medium | High | 1 |
| MEM-053-CPP | High | High | Medium | High | 1 |
| ERR-006-C | Medium | Low | Medium | Low | 3 |
| ERR-051-CPP | Low | Medium | Medium | Low | 3 |
| OOP-052-CPP | Low | High | Low | Medium | 2 |
| INT-035-C | Low | Low | Medium | Low | 3 |
| STR-038-C | High | High | Low | High | 1 |

### 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 | This ensures that data is encrypted while being stored on-disk. This policy applies because it is utilized to secure sensitive data that is stored. For example, if a hacker obtains access to a hard drive containing sensitive data, if the hard drive is encrypted using this policy the hacker will still need the encryption key to access the data. |
| Encryption at flight | This ensures that data is encrypted while being transferred between locations. Most, if not all data will be transferred to various locations at some instance. To ensure the security data in motion it is important to utilize this policy. Attackers will often attempt to utilize cloud connection or WiFi vulnerabilities to intercept this data during transmission. If this data is encrypted the data obtained will be inaccessible without the encryption key. |
| Encryption in use | This ensures that data being used or data in memory such as RAM is encrypted. This policy applied because most data will be used in some form. It is important to ensure that sensitive data is encrypted when in use in computations or in memory. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves confirming that the user trying to access the system is who they claim to be. Utilizing authentication allows the system to reject or accept a user based on the credentials provided by the users. This usually takes the form of a username and password combination. This policy should be applied to any system that allows users to access sensitive data. Authentication typically applies to one of the first layers of defense in a system. |
| Authorization | Authorization refers to the privileges, roles, and rights assigned to authenticated users. This policy applies since the principle of least privileges should be followed. This means that users should be given the least privileges and access needed to complete their tasks. This principle ensures that users have less access to sensitive data deterring harmful use of sensitive data by unauthorized users. |
| Accounting | Accounting refers to the process of monitoring user actions and activities and recording incidents for compliance and security reasons. Accounting allows unauthorized or risky actions to be addressed and acknowledged before damage to the system is done. Accounting can be utilized on authorized users of the system to record proof of actions taken. Additionally, unauthorized users typically leave a trail indicating the vulnerability in the system. Through accounting, if an attack occurs, the vulnerability can be identified and addressed promptly. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.0.1 | 11/10/2023 | Added Ten Security Standards | Cody Faircloth |  |
| 1.1 | 12/1/2023 | Document Completed | Cody Faircloth |  |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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