**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 7](#_Toc52464060)

[Coding Standard 3 9](#_Toc52464061)

[Coding Standard 4 11](#_Toc52464062)

[Coding Standard 5 13](#_Toc52464063)

[Coding Standard 6 15](#_Toc52464064)

[Coding Standard 7 17](#_Toc52464065)

[Coding Standard 8 19](#_Toc52464066)

[Coding Standard 9 22](#_Toc52464067)

[Coding Standard 10 24](#_Toc52464068)

[Defense-in-Depth Illustration 26](#_Toc52464069)

[Project One 26](#_Toc52464070)

[1. Revise the C/C++ Standards 26](#_Toc52464071)

[2. Risk Assessment 26](#_Toc52464072)

[3. Automated Detection 26](#_Toc52464073)

[4. Automation 26](#_Toc52464074)

[5. Summary of Risk Assessments 27](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 27](#_Toc52464076)

[7. Map the Principles 28](#_Toc52464077)

[Audit Controls and Management 31](#_Toc52464078)

[Enforcement 31](#_Toc52464079)

[Exceptions Process 31](#_Toc52464080)

[Distribution 32](#_Toc52464081)

[Policy Change Control 32](#_Toc52464082)

[Policy Version History 32](#_Toc52464083)

[Appendix A Lookups 32](#_Toc52464084)

[Approved C/C++ Language Acronyms 32](#_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 | Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software [vulnerabilities](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-vulnerability). Be suspicious of most external data sources, including command-line arguments, network interfaces, environmental variables, and user-controlled files [Seacord 05]. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [[C MSC00-A](https://wiki.sei.cmu.edu/confluence/display/c/MSC00-C.+Compile+cleanly+at+high+warning+levels), [C++ MSC00-A](https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046361)]. Use static and dynamic analysis tools to detect and eliminate additional security flaws. |
| 1. Architect and Design for Security Policies | Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set. |
| 1. Keep It Simple | Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. Additionally, the effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex. |
| 1. Default Deny | Base access decisions on permission rather than exclusion. This means that, by default, access is denied, and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75]. |
| 1. Adhere to the Principle of Least Privilege | Every process should execute 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 [Saltzer 74, Saltzer 75]. |
| 1. Sanitize Data Sent to Other Systems | Sanitize all data passed to complex subsystems [[C STR02-A](https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems)] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem. |
| 1. Practice Defense in Depth | Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a [security flaw](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-securityflaw) from becoming an exploitable vulnerability and/or limit the consequences of a successful [exploit](https://wiki.sei.cmu.edu/confluence/display/c/BB.+Definitions#BB.Definitions-exploit). For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05]. |
| 1. Use Effective Quality Assurance Techniques | Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Fuzz testing, penetration testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05]. |
| 1. Adopt a Secure Coding Standard | 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** | **Ensure that integer conversions do not result in lost or misinterpreted data** |
| --- | --- | --- |
| **Data Type** | STD-001-C | Integer conversions, both implicit and explicit (using 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  -Function arguments of type size\_t or resize\_t  The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. |

| **Noncompliant Code (Unsigned and signed)** |
| --- |
| Type range errors, including loss of data (truncation) and loss of sign (sign error) can occur when converting from a value of an unsigned integer type to a value of a signed integer type. This noncompliant code 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):** Adopt a secure coding standard  **Ensure that integer conversions do not result in lost or misinterpreted data** maps with the principle of adopting a secure coding standard because integer conversion errors can lead to buffer overflows and the execution of arbitrary code by an attacker. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 1.6 | memsetValueOutOfRange | The second argument to memset() cannot be represented as unsigned char |
| Coverity | 2017.07 | NEGATIVE\_RETURNS  REVERSE\_NEGATIVE  MISRA\_CAST | Can find any address, loop bounds, and other expressions that may contain dangerous implied integer conversions that would result in unexpected behavior.  Can find instances where a negativity check occurs after the negative value has been used for something else  Can find instance where an integer expression is implicitly converted to a narrower integer type, where the signedness of an integer value is implicitly converted, or where the type of complex expression is implicitly converted |
| CodeSonar | 602p0 | 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 |
| Astree | 20.10 |  | Supported via MISRA C : 2012 rules 10.1, 10.3, 10.4, 10.6 and 10.7 |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Ensure that operations on signed integer do not result in overflow** |
| --- | --- | --- |
| **Data Value** | STD-002-C | Signed integer overflow is undefined behavior. Consequently, implementations have considerable latitude in how they deal with signed integer overflow. Of particular importance are operations on signed integer values that originate from a tainted source and are used as integer operands of any pointer arithmetic, assignment expression, postfix expression and function arguments. |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in a signed integer overflow during the subtraction of the signed operands si\_a and si\_b: |
| void func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** diff = si\_a - si\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution tests the operands of the subtraction to guarantee there is no possibility of signed overflow, regardless of representation: |
| #include <limits.h>    void func(**signed** **int** si\_a, **signed** **int** si\_b) {  **signed** **int** diff;    if ((si\_b > 0 && si\_a < INT\_MIN + si\_b) ||        (si\_b < 0 && si\_a > INT\_MAX + si\_b)) {      /\* Handle error \*/    } else {      diff = si\_a - si\_b;    }      /\* ... \*/  } |

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

| **Principles(s):** Adopt a secure coding standard  **Ensure that operations on signed integer do not result in overflow** maps with the principle of adopting a secure coding standard because integer overflow can lead to buffer overflows and the execution of arbitrary code by an attacker. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Parasoft c/c++ test | 2021.2 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow expression in “+”, “- “, “\*” operator  Integer overflow or underflow in constant expressions in “<<” operator |
| CodeSonar | 602p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| PRQA QA-C | 9.7 | 2800, 2801, 2802, 2803, 2860, 2861, 2862, 2863 | Fully implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Guarantee that storage for strings has sufficient space for character data and the null terminator** |
| --- | --- | --- |
| **String Correctness** | STD-003-C | Copying data to a buffer that is not large enough to hold that data results in a buffer overflow. Buffer overflow occur frequently when manipulating strings. To prevent such errors, either limit copies through truncation or, preferably, ensure that the destination is of sufficient size to hold the data to be copied. |

| **Noncompliant Code** |
| --- |
| Because the input is unbounded, the following code could lead to a buffer overflow. |
| #include <iostream>    void f() {  **char** buf[12];    std::cin >> buf;  } |

| **Compliant Code** |
| --- |
| The best solution for ensuring that data is not truncated and for guarding against buffer overflows is to use std::string instead of a bounded array, as in this compliant solution. |
| #include <iostream>  #include <string>    void f() {    std::string input;    std::string stringOne, stringTwo;    std::cin >> stringOne >> stringTwo;  } |

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

| **Principles(s):** Adopt a secure coding standard  **Guarantee that storage for strings has sufficient space for character data, and the null terminator** maps with the principle of adopting a secure coding standard because copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported  Astree reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Axivion Bahaus Suite | 7.20 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow.  Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| CodeSonar | 602p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warnings classes that report uses of library functions prone to internal buffer overflows. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Sanitize 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. |

| **Noncompliant Code** |
| --- |
| This noncompliant code is taken from a vulnerability in Sun Solaris Telnet daemon,that allow a remote attacker to log on to the system with elevated privileges. |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")),    0); |

| **Compliant Code** |
| --- |
| The compliant solution insert the “- - “ (double dash) arguments before the call to getenv(“USER”) in the call execl() |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")), 0); |

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

| **Principles(s):** Sanitize data sent to others system/ adopt a secure coding standard  **Sanitize data passed to complex subsystems** maps with both principles because it can lead to injection attacks, data integrity issues, and loss of sensitive data. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported by stubbing/taint analysis |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| CodeSonar | 6.2p0 | 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 |
| Parasoft c/c++ test | [Insert text.] | CERT\_C.STR02-a  CERT\_C.STR02-b  CERT\_C.STR02-c | Protect against command injection  Protect again file name injection  Protect again SQL injection |

#### Coding Standard 5

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

| **Noncompliant Code (std::unique\_ptr)** |
| --- |
| In this noncompliant example the dynamically allocated memory managed by the buffer object is accessed after it has been implicitly deallocated by the object’s destructor |
| #include <iostream>  #include <memory>  #include <cstring>    **int** main(**int** argc, const **char** \*argv[]) {    const **char** \*s = "";    if (argc > 1) {      enum { BufferSize = 32 };      try {        std::unique\_ptr<**char**[]> buff(new **char**[BufferSize]);        std::**memset**(buff.get(), 0, BufferSize);        // ...        s = std::**strncpy**(buff.get(), argv[1], BufferSize - 1);      } catch (std::bad\_alloc &) {        // Handle error      }    }      std::cout << s << std::endl;  } |

| **Compliant Code** |
| --- |
| In this compliant solution, a variable with automatic storage duration with type std::string is used in place of the std::unique\_ptr<char[]>, which reduces the complexity and improves the security of the solution. |
| #include <iostream>  #include <string>    **int** main(**int** argc, const **char** \*argv[]) {    std::string str;      if (argc > 1) {      str = argv[1];    }      std::cout << str << std::endl;  } |

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

| **Principles(s):** adopt a secure coding standard  **Do not access freed memory** maps with the principle of adopting a secure coding standard because dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial of service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permission of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++. MEM50 |  |
| CodeSonar | 6.2p0 | 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 memory. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Incorporate diagnostic tests using assertions** |
| --- | --- | --- |
| **Assertions** | STD-006-C | Incorporate diagnostic test into your program using for example, the assert() macro. Assertions are primarily intended for use during debugging and are generally turned off before code is deployed by defining the NDEBUG macro. A failed assertion can lead to a denial of service attack if triggered by a malicious user, such as size being derived, in some way from client input. Assertion should never be used to verify the absence of runtime errors. |

| **Noncompliant Code ( malloc())** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded which is inappropriate because doing so might lead to abrupt termination of the process, opening the possibility of a denial of service attack. |
| **char** \*dupstring(const **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char** \*)**malloc**(len + 1);  **assert**(NULL != dup);    **memcpy**(dup, c\_str, len + 1);    return dup;  } |

| **Compliant Code** |
| --- |
| This compliant solution demonstrates how to detect and handle possible memory exhaustion |
| **char** \*dupstring(const **char** \*c\_str) {  **size\_t** len;  **char** \*dup;      len = **strlen**(c\_str);    dup = (**char**\*)**malloc**(len + 1);    /\* Detect and handle memory allocation error \*/    if (NULL == dup) {        return NULL;    }    **memcpy**(dup, c\_str, len + 1);    return dup;  } |

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

| **Principles(s):** Adopt a secure coding standard/ Use effective quality assurance techniques  **Incorporate diagnostic tests using assertions maps with both principles** because assertions are valuable diagnostic tools for finding and eliminating software defects that may result in vulnerabilities. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.2p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECT | Can detect the specific instance where assertion contains an operation/function call that may have side effect |
| Parasoft c/c++ test | 2021.2 | CERT\_C.MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Handle all exceptions** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | When an exception is thrown, control is transferred to the nearest handler with a type that matches the type of exceptions thrown. If no matching handler is directly found within the handlers for a try block in which the expression is thrown, the search for matching handler continues to dynamically search for handlers in the surrounding try blocks of the same thread. If no matching handler is found, the function std::terminate() is called, which abnormally terminate the process with the risk of denial of service attacks. |

| **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 the main() function and allows for graceful management of external ressources. |
| 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):** Adopt a secure coding standard  **Handle all exceptions** maps with the principle of adopting a secure coding standard because allowing the application to abnormally terminate can lead to resources not being freed, closed and so on. It is frequently a vector for denial of service. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++.ERR51 |  |
| Parasoft C/C++ test | 2021.2 | 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 calls path that could lead to that point |
| PRQA QA C++ | 4.4 | 4035, 4036, 4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Exclude user input from format strings** |
| --- | --- | --- |
| **Data value**  **String Injection** | STD-008-C | Never called a formatted Input/Output function with a format string containing a tainted value. An attacker who can fully or partially control the contents of a format string can crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary location. Consequently, the attacker can execute arbitrary code with the permission of the vulnerable process. |

| **Noncompliant Code** |
| --- |
| The incorrect\_password() function is in this noncompliant code example is called during identification and authentication to display an error message if the specified user is not found or the password is incorrect.The function accept the name of the user as a string referenced by user. This is an exemplar of untrusted data that originate from unauthenticated user. The function constructs an error message that is then output to stderr using the C standard fprintf() function. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/    static const **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + sizeof(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }  **fprintf**(stderr, msg);  **free**(msg);  } |

| **Compliant Code ( fputs() )** |
| --- |
| This compliant solution fixes the problem by replacing the fprintf() call with a call to fputs(), which outputs message directly to stderr without evaluating its content. |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>    void incorrect\_password(const **char** \*user) {  **int** ret;    /\* User names are restricted to 256 or fewer characters \*/    static const **char** msg\_format[] = "%s cannot be authenticated.\n";  **size\_t** len = **strlen**(user) + sizeof(msg\_format);  **char** \*msg = (**char** \*)**malloc**(len);    if (msg == NULL) {      /\* Handle error \*/    }    ret = snprintf(msg, len, msg\_format, user);    if (ret < 0) {      /\* Handle error \*/    } else if (ret >= len) {      /\* Handle truncated output \*/    }  **fputs**(msg, stderr);  **free**(msg);  } |

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

| **Principles(s):** Adopt a secure coding standard/Validate input data/Sanitize data sent to other systems  **Exclude user from input format strings** maps with the three principles because failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported via stubbing/taint analysis |
| CodeSonar | 6.2p0 | IO.INJ.FMT  MISC.FMT | Format string injection  Format string |
| Coverity | 2017.07 | TAINTED\_STRING | Implemented |
| Parasoft c/c++ test | 2021.2 | CERT\_C.FIO30-a  CERT\_C.FIO30-b  CERT\_C.FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprint/fwprintf with only two parameters, when second parameter is a variable  Never used unfiltered data from an untrusted user as the format parameter |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Ensure that unsigned integer operations do not wrap** |
| --- | --- | --- |
| Data Value | STD-009-C | A computation involving unsigned operands can never overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type. |

| **Noncompliant Code** |
| --- |
| This noncompliant example code can result in unsigned integer wrap during the addition of the unsigned ui\_and ui\_b. If this behavior is unexpected, 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 |
| **void** func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| This compliant solution performs a precondition test of the operands of the addition to guarantee there is no possibility of unsigned wrap. |
| #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):** Adopt a secure coding standard  **Ensure that unsigned integer operations do not wrap** maps with the principle of adopting a secure coding standard because integer wrap can lead to buffer overflows and arbitrary code execution by an attacker. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 6.2p0 | ALLOC.SIZE.ADDOFLOW  ALLOC.SIZE.IOFLOW  ALLOC.SIZE.MULOFLOW  ALLOC.SIZE.SUBUFLOW  MISC.MEM.SIZE.ADDOFLOW  MISC.MEM.SIZE.BAD  MISC.MEM.SIZE.MULOFLOW  MISC.MEM.SIZE.SUBUFLOW | Addition overflow of allocation size  Integer overflow of allocation size  Multiplication overflow of allocation size  Subtraction underflow of allocation size  Addition overflow of size  Unreasonable size argument  Multiplication overflow of size  Subtraction underflow of size |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Only free memory allocated dynamically** |
| --- | --- | --- |
| Memory protection | STD-010-C | Freeing memory that is not allocated dynamically can result in heap corruption and others serious errors. Do not call free () on a pointer other that one returned by a standard memory allocation function, such as malloc(), calloc(), realloc(), or aligned\_alloc(). One consequence is that the program may terminate abnormally.  This rule does not apply to null pointers. |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the pointer parameter to realloc(), buf, does not refer to dynamically allocated memory: |
| #include <stdlib.h>    enum { BUFSIZE = 256 };    void f(void) {  **char** buf[BUFSIZE];  **char** \*p = (**char** \*)**realloc**(buf, 2 \* BUFSIZE);    if (p == NULL) {      /\* Handle error \*/    }  } |

| **Compliant Code** |
| --- |
| In this compliant solution, buf refers to dynamically allocated memory.  Note that realloc () will behave properly even malloc () failed, because when given a null pointer, realloc () behaves like a call to malloc (). |
| #include <stdlib.h>    enum { BUFSIZE = 256 };    void f(void) {  **char** \*buf = (**char** \*)**malloc**(BUFSIZE \* sizeof(**char**));  **char** \*p = (**char** \*)**realloc**(buf, 2 \* BUFSIZE);    if (p == NULL) {      /\* Handle error \*/    }  } |

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

| **Principles(s):** Adopt a secure coding standard  **Only free memory allocated dynamically** maps with the principle of adopting a secure coding standard becauseit can lead to arbitrary code execution if that memory is reused by malloc() |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Invalid-free | Fully checked |
| Axivion bauhaus suite | 7.2.0 | CertC-MEM34 | Can detect memory deallocation for stack objects |
| Clang | 3.9 | Clang-analyser-unix.Malloc | Checked by clang-tidy: can detect some instance of this rule, but does not detect all |
| PC-lint Plus | 1.4 | 424, 673 | Fully supported |

### Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



The existing devops process put security at the end, but the devSecOps integrate security at the left.

In the pre-production phase, specifically at the design step, as soon as the developer commits into the folder or during the pre-build, SCA/SAST tools are used to run static analysis tools, detect and fix vulnerabilities found early in development based on the standards in this policy. The code continues to be scanned during the post-build steps, including the production phase with dynamic analysis tools (DAST) for penetration test, intrusion detection, …etc. The process follows continuous integration and a continuous deployment pipeline.

### Summary of Risk Assessments

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-C | High | Probable | high | P6 | L2 |
| STD-002-C | High | Likely | High | P9 | L2 |
| STD-003-C | High | Likely | Medium | P18 | L1 |
| STD-004-C | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-C | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | Low | Probable | Medium | P4 | L3 |
| STD-008-C | High | Likely | Medium | P18 | L1 |
| STD-009-C | High | Likely | High | P9 | L2 |
| STD-010-C | High | Likely | Medium | P18 | L1 |

### Create Policies for Encryption and Triple A

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | It is the process of translating archived data or data stored in a database into another form of data that unauthorized users can not decrypt. Green Pace encrypts all customer data stored at rest (Full disk, files, tokens, credentials, and folder )without any action required from the customer with one or more encryption mechanisms. The company uses several layers of encryption to protect data integrity and confidentiality using public encryption algorithms such as AES 256 OR AES128. |
| Encryption at flight | It is the process of translating data in transit into another form of data that unauthorized users can not intercept and decrypt. Green Pace encrypts all customer data in flight (Full disk, files, and folder ) to ensure delivery without eavesdropping, interception, and forgery. The company uses an appropriate configuration of servers to allow clients to connect securely. For example, Green pace uses transport layer security(TLS) for HTTPS and avoid self-signed certificate. |
| Encryption in use | It translates data into another form of data while using this data, protecting data integrity and confidentiality. Green Pace is compliant and encrypts all customer-sensitive data in use, such as passwords, credit cards number, and social security. The company use public encryption algorithms such as AES 256 OR AES128 |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | It allows identifying who is requesting a connexion to the application to prevent the various attacks (data breach, phishing, injection, ..etc.). Green Pace application provides a multi-layer of authentication based on user success or failure and level of access. The basic authentication mechanisms require user logins. Other types of authentication, such as multi-factor, certificate-based, token-based, and biometric, are used based on the risk assessment.  Passwords are encrypted during storage and transmission and constructed to resist attack. |
| Authorization | It allows users to access certain functionalities based on their privileges. Green Pace enforces the least privileges principle to avoid broken access control, ranked as the fifth most concerning web security vulnerability in OWASP 2017 TOP 10. The company adopts deny by default and uses both role-based access control and attribute-based access control. For example, changes to the database, addition of news users, files access, and user level of access are controlled by administrators with different privileges. Every request must be validated and exit safely when authorization checks fail. |
| Accounting | It tracks what happens after authentication and authorization, what resources have been used by who, and when. Green Pace applications store accounting records on a dedicated server to allow rapid response in an attack. |

### Map the Principles

|  |  |  |
| --- | --- | --- |
| **Standards** | **Principles** | **Explanations** |
| STD-001-C | **10,3** | **Ensure that integer conversions do not result in lost or misinterpreted data** maps with the principle of **adopting a secure coding standard** and **Architect and Design for Security Policies** because integer conversion errors can lead to buffer overflows and the execution of arbitrary code by an attacker. |
| STD-002-C | **10,3** | **Ensure that operations on signed integer do not result in overflow** maps with the principle of **adopting a secure coding standard** and **Architect and Design for Security Policies** because integer overflow can lead to buffer overflows and the execution of arbitrary code by an attacker. |
| STD-003-C | **10,3** | **Guarantee that storage for strings has sufficient space for character data, and the null terminator** maps with the principle of **adopting a secure coding standard** and **Architect and Design for Security Policies** because copying string data to a buffer that is too small to hold that data results in a buffer overflow. Attackers can exploit this condition to execute arbitrary code with permissions of the vulnerable process. |
| STD-004-C | **10,7,3** | **Sanitize data passed to complex subsystems** maps with the three principles because it can lead to injection attacks, data integrity issues, and loss of sensitive data. |
| STD-005-CPP | **10,3,4** | **Do not access freed memory** maps with the principle of **adopting a secure coding standard** and **Architect and Design for Security Policies** because dynamically allocated memory after it has been deallocated can lead to abnormal program termination and denial of service attacks. Writing memory that has been deallocated can lead to the execution of arbitrary code with the permission of the vulnerable process. |
| STD-006-C | **10,9,3** | **Incorporate diagnostic tests using assertions maps with the three principles** because assertions are valuable diagnostic tools for finding and eliminating software defects that may result in vulnerabilities. |
| STD-007-CPP | **10,3,8** | **Handle all exceptions** maps with the principle of **adopting a secure coding standard** and **Architect and Design for Security Policies** because allowing the application to abnormally terminate can lead to resources not being freed, closed, and so on. It is frequently a vector for denial of service. |
| STD-008-C | **10,7,1,3,6** | **Exclude user from input format strings** maps with the four principles because failing to exclude user input from format specifiers may allow an attacker to crash a vulnerable process, view the contents of the stack, view memory content, or write to an arbitrary memory location and consequently execute arbitrary code with the permissions of the vulnerable process. |
| STD-009-C | **10,3** | **Ensure that unsigned integer operations do not wrap** maps with the principles of **adopting a secure coding standard** and **Architect and Design for Security Policies** because integer wrap can lead to buffer overflows and arbitrary code execution by an attacker. |
| STD-010-C | **10,3** | **Only free memory allocated dynamically** maps with the principles of **adopting a secure coding standard** and **Architect and Design for Security Policies** because it leads to arbitrary code execution if that memory is reused by malloc(). |
| Authentification | **1,3** | It maps with the principles of input data validation and **Architect and Design for Security Policies** because authentication require data validation |
| Authorization | **3,5,6,8** | It maps with the four principles because :  -authorization requires access and privileges management from one side and the other to use defense-in-depth strategies (Authentication-Authorization).  -authorization requires following coding standards—for example, handling all exceptions to exit safely when authorization checks fail. |
| Accounting | **8** | It maps with practice defense-in-depth because accounting tracks what happens after authentication and authorization. |

**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 | 12/05/2021 | Coding standard | Zancran Togbe | Trevor Hodde |
| 1.2 | 2/17/2022 | Project 1 completed | Zancran Togbe | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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