**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_heading=h.gjdgxs)

[Purpose 2](#_heading=h.30j0zll)

[Scope 2](#_heading=h.1fob9te)

[Module Three Milestone 2](#_heading=h.3znysh7)

[Ten Core Security Principles 2](#_heading=h.4d34og8)

[C/C++ Ten Coding Standards 3](#_heading=h.2s8eyo1)

[Coding Standard 1 4](#_heading=h.3rdcrjn)

[Coding Standard 2 5](#_heading=h.26in1rg)

[Coding Standard 3 6](#_heading=h.lnxbz9)

[Coding Standard 4 7](#_heading=h.35nkun2)

[Coding Standard 5 8](#_heading=h.1ksv4uv)

[Coding Standard 6 9](#_heading=h.2jxsxqh)

[Coding Standard 7 10](#_heading=h.z337ya)

[Coding Standard 8 11](#_heading=h.28h4qwu)

[Coding Standard 9 13](#_heading=h.nmf14n)

[Coding Standard 10 14](#_heading=h.37m2jsg)

[Defense-in-Depth Illustration 15](#_heading=h.1mrcu09)

[Project One 15](#_heading=h.46r0co2)

[1.](#_heading=h.2lwamvv) Revise the C/C++ Standards 15

[2.](#_heading=h.111kx3o) Risk Assessment 15

[3.](#_heading=h.3l18frh) Automated Detection 15

[4.](#_heading=h.206ipza) Automation 15

[5.](#_heading=h.4k668n3) Summary of Risk Assessments 16

[6.](#_heading=h.2zbgiuw) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.1egqt2p) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.3ygebqi)

[Enforcement 18](#_heading=h.2dlolyb)

[Exceptions Process 18](#_heading=h.sqyw64)

[Distribution 19](#_heading=h.3cqmetx)

[Policy Change Control 19](#_heading=h.1rvwp1q)

[Policy Version History 19](#_heading=h.4bvk7pj)

[Appendix A Lookups 19](#_heading=h.2r0uhxc)

[Approved C/C++ Language Acronyms 19](#_heading=h.1664s55)

## 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 all input from unreliable data sources, at the very least. Most software vulnerabilities can be eliminated with proper input validation. Most external data sources, such as command line arguments, network interfaces, environmental variables, and user-controlled files, should be regarded with suspicion. |
| 1. Heed Compiler Warnings | Use your compiler's highest warning level while compiling code, and then change the code to remove all warnings. |
| 1. Architect and Design for Security Policies | Design your program with a software architecture to implement and enforce security policies. Consider segmenting your system into discrete, communicative subsystems, each with the proper set of privileges, if, for instance, your system requires various privileges at various times. |
| 1. Keep It Simple | Make the design as straightforward and compact as you can. There is a higher chance that complex designs will be implemented, configured, and used incorrectly. Additionally, when security measures get more complicated, it takes a lot more work to obtain the right level of assurance. |
| 1. Default Deny | Instead of excluding decisions, make access decisions based on authorization. In other words, access is prohibited by default, and the protection scheme specifies the circumstances in which access is allowed. |
| 1. Adhere to the Principle of Least Privilege | Every process should run with the fewest privileges required to finish the task. Any increased permission ought to be kept for at least a short while. This strategy lessens the potential harm brought on by unintentional misuse and unauthorized use, as well as the chances of an attacker being able to run arbitrary code with elevated privileges. |
| 1. Sanitize Data Sent to Other Systems | Ensure that all information sent to complex subsystems, including command shells, relational databases, and commercial off-the-shelf (COTS) components, is sanitized. Attackers might be able to employ SQL, command, or other injection techniques to access unused functionality in these components. Since the complex subsystem being called does not comprehend the context of the call, this is not necessarily an issue with input validation. The calling process is in charge of sanitizing the data prior to launching the subsystem because it is aware of the context. |
| 1. Practice Defense in Depth | Manage risk by employing several defensive tactics such that, in the event that one layer of defense proves insufficient, a different layer of defense can stop a security fault from developing into an exploitable vulnerability and/or restrict the effects of a successful exploit. For instance, using secure programming methods in conjunction with secure runtime environments could lessen the possibility of vulnerabilities in the code still present at the time of deployment being exploited in the operational environment. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. Systems that have undergone independent security examinations may be more secure. In identifying and correcting false assumptions, for instance, external reviewers offer an unbiased viewpoint. |
| 1. Adopt a Secure Coding Standard | Implement secure coding standards for your chosen development languages and platforms. |

### 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Data Type: The sizes given to common data types are defined by a data model. It's critical to comprehend the data models that your implementation uses. However, you should include static assertions to make sure that your assumptions are true if your code depends on any that the standard does not guarantee.  Rule 04: INT32-C. Ensure that operations on signed integers do not result in  overflow |

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

| **Compliant Code** |
| --- |
| The product of two operands can always be represented using twice the number of bits than exist in the precision of the larger of the two operands. This compliant solution eliminates signed overflow on systems where long long is at least twice the precision of int: |
| #include <stddef.h>  #include <assert.h>  #include <limits.h>  #include <inttypes.h>    extern size\_t popcount(uintmax\_t);  #define PRECISION(umax\_value) popcount(umax\_value)    void func(signed int si\_a, signed int si\_b) {  signed int result;  signed long long tmp;  assert(PRECISION(ULLONG\_MAX) >= 2 \* PRECISION(UINT\_MAX));  tmp = (signed long long)si\_a \* (signed long long)si\_b;    /\*  \* If the product cannot be represented as a 32-bit integer,  \* handle as an error condition.  \*/  if ((tmp > INT\_MAX) || (tmp < INT\_MIN)) {  /\* Handle error \*/  } else {  result = (int)tmp;  }  /\* ... \*/  } |

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

| **Principles(s):** Validate input data: Out-of-bound memory errors can be avoided by correctly validating.  Heed compiler warnings: paying attention to compiler warnings can be caught before production. Use effective quality assurance techniques: employing QA techniques can be used to analyze in the pipeline to catch this kind of problem. Architect and design for security policies: Divide the system into different, interconnected subsystems, each with the proper set of privileges, as part of the secure software architectural design. In order to undertake security-critical activities, this would enable a process that does not have elevated privileges to communicate with a process that does. Make that the system has adequate deny by default, input validation, and data cleansing in place. Make the algorithms available to the public so that they can be examined and analyzed for security. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | integer-overflow | Fully checked |
| CodeSonar | 7.1p0 | 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 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| Helix QAC | 2022.3 | C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863 |  |
| Klocwork | 2022.3 | NUM.OVERFLOW  CWARN.NOEFFECT.OUTOFRANGE  NUM.OVERFLOW.DF |  |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | Partially implemented |
| Parasoft C/C++ test | 2022.1 | CERT\_C-INT32-a  CERT\_C-INT32-b  CERT\_C-INT32-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Parasoft Insure++ |  |  | Runtime analysis |
| Polyspace Bug Finder | R2022b | CERT C: Rule INT32-C | Checks for:   * Integer overflow * Tainted division operand * Tainted modulo operand   Rule partially covered. |
| PRQA QA-C | 9.7 | 2800, 2801, 2802, 2803,  2860, 2861, 2862, 2863 | Fully implemented |
| PRQA QA-  C++ | 4.4 | 2800, 2801, 2802, 2803,  2860, 2861, 2862, 2863 |  |
| PVS-Studio | 7.21 | V1026, V1070, V1081, V1083, V1085, V5010 |  |
| TrustInSoft Analyzer | 1.38 | signed\_overflow | Exhaustively verified (see one compliant and one non-compliant example). |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | Data Value: One of the most important aspects of the C programming language is data types. For describing the types of data that are normally stored there, we employ data types with functions and variables. This information may take the form of a character or a value. There may also be different character sets or value sets.  Rule 03:  INT30-C. Ensure that unsigned integer operations do not wrap |

| **Noncompliant Code** |
| --- |
| This noncompliant code example can result in an unsigned integer wrap during the addition of the unsigned operands ui\_a and ui\_b. If this behavior is unexpected, 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):** Validate input data: robust protection against data types with data corruption, buffer overflows, or malformed strings. In reality, input validation is the best defense against buffer overflow, according to Secure Coding in C and C++ (Second Edition). Use effective quality assurance techniques: Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 | integer-overflow | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-INT30 | Implemented |
| CodeSonar | 7.1p0 | 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 |
| Compass/ROSE |  |  | Can detect violations of this rule by ensuring that operations are checked for overflow before being performed (Be mindful of exception INT30-EX2 because it excuses many operations from requiring validation, including all the operations that would validate a potentially dangerous operation. For instance, adding two unsigned ints together requires validation involving subtracting one of the numbers from UINT\_MAX, which itself requires no validation because it cannot wrap.) |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |
| Helix QAC | 2022.3 | C2910, C2911, C2912, C2913, C3383, C3384, C3385, C3386  C++2910, C++2911, C++2912, C++2913 |  |
| Klocwork | 2022.3 | NUM.OVERFLOW  CWARN.NOEFFECT.OUTOFRANGE  NUM.OVERFLOW.DF |  |
| LDRA tool suite | 9.7.1 | 493 S, 494 S | Partially implemented |
| Parasoft C/C++test | 2022.1 | CERT\_C-INT30-a  CERT\_C-INT30-b  CERT\_C-INT30-c | Avoid integer overflows  Integer overflow or underflow in constant expression in '+', '-', '\*' operator  Integer overflow or underflow in constant expression in '<<' operator |
| Polyspace Bug Finder | R2022b | CERT C: Rule INT30-C | Checks for:   * Unsigned integer overflow * Unsigned integer constant overflow   Rule partially covered. |
| PRQA QA-C | 9.7 | 2910 [C], 2911 [D], 2912 [A],  2913 [S], 3383, 3384, 3385, 3386 | Partially implemented |
| PRQA QA-C++ | 4.4 | 2910, 2911, 2912, 2913 |  |
| PVS-Studio | 7.21 | V658, V1012, V1028, V5005, V5011 |  |
| TrustInSoft Analyzer | 1.38 | unsigned overflow | Exhaustively verified. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | A sequence of zero or more multibyte characters contained in double-quotes, such as "xyz," constitutes a character string literal. The only difference is that a UTF-8 string literal is preceded with u8. The only difference is that a wide string literal is prefixed with the letters L, u, or U. A string literal should not be cast to a pointer to non-const or assigned to a pointer to non-const. A pointer to (or array of) const characters must be regarded as a string literal for the purposes of this rule. A buffer overflow happens when data is copied into a buffer that is too small to hold that data. String manipulation frequently leads to buffer overflows. Limit copies by truncation or, preferable, make sure the destination is big enough to carry the character data that has to be transferred together with the null-termination character.  Rule 07:  STR30-C. Do not attempt to modify string literals  STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator |

| **Noncompliant Code** |
| --- |
| The gets() function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from stdin. This noncompliant code example assumes that gets() will not read more than BUFFER\_SIZE - 1 characters from stdin. This is an invalid assumption, and the resulting operation can result in a buffer overflow.  The gets() function reads characters from the stdin into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. |
| #include <stdio.h>    #define BUFFER\_SIZE 1024    void func(void) {  char buf[BUFFER\_SIZE];  if (gets(buf) == NULL) {  /\* Handle error \*/  }  } |

| **Compliant Code** |
| --- |
| The fgets() function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from stdin to buf cannot exceed the allocated memory: |
| #include <stdio.h>  #include <string.h>    enum { BUFFERSIZE = 32 };    void func(void) {  char buf[BUFFERSIZE];  int ch;    if (fgets(buf, sizeof(buf), stdin)) {  /\* fgets() succeeded; scan for newline character \*/  char \*p = strchr(buf, '\n');  if (p) {  \*p = '\0';  } else {  /\* Newline not found; flush stdin to end of line \*/  while ((ch = getchar()) != '\n' && ch != EOF)  ;  if (ch == EOF && !feof(stdin) && !ferror(stdin)) {  /\* Character resembles EOF; handle error \*/  }  }  } else {  /\* fgets() failed; handle error \*/  }  } |

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

| **Principles(s):** Validate input data: Validate all input from unreliable data sources, at the very least. Most software vulnerabilities can be eliminated with proper input validation. Most external data sources, such as command line arguments, network interfaces, environmental variables, and user-controlled files, should be regarded with suspicion.  Heed compiler warning: Use your compiler's highest warning level while compiling code, and then change the code to remove all warnings. Sanitize Data Sent to Other Systems: Ensure that all information sent to complex subsystems, including command shells, relational databases, and commercial off-the-shelf (COTS) components, is sanitized. Attackers might be able to employ SQL, command, or other injection techniques to access unused functionality in these components. Use Effective Quality Assurance Techniques: Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Axivion Bauhaus Suite | 7.2.0 | 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() |
| CodeSonar | 7.1p0 | LANG.MEM.BO  LANG.MEM.TO  MISC.MEM.NTERM  BADFUNC.BO.\* | Buffer overrun  Type overrun  No space for null terminator  A collection of warning classes that report uses of library functions prone to internal buffer overflows |
| Compass/ROSE |  |  | Can detect violations of the rule. However, it is unable to handle cases involving strcpy\_s() or manual string copies such as the one in the first example |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |
| Fortify SCA | 5.0 |  |  |
| Helix QAC | 2022.3 | C2840, C2841, C2842, C2843, C2845, C2846, C2847, C2848, C2930, C2931, C2932, C2933, C2935, C2936, C2937, C2938, C5009, C5038  C++0145, C++2840, C++2841, C++2842, C++2843, C++2845, C++2846, C++2847, C++2848, C++2930, C++2931, C++2932, C++2933, C++2935, C++2936, C++2937, C++2938, C++5009, C++5038 |  |
| Klocwork | 2022.3 | SV.FMT\_STR.BAD\_SCAN\_FORMAT  SV.UNBOUND\_STRING\_INPUT.FUNC |  |
| LDRA tool suite | 9.7.1 | 489 S, 109 D, 66 X, 70 X, 71 X | Partially implemented |
| Parasoft C/C++test | 2022.1 | CERT\_C-STR31-a  CERT\_C-STR31-b  CERT\_C-STR31-c  CERT\_C-STR31-d  CERT\_C-STR31-e | Avoid accessing arrays out of bounds  Avoid overflow when writing to a buffer  Prevent buffer overflows from tainted data  Avoid buffer write overflow from tainted data  Avoid using unsafe string functions which may cause buffer overflows |
| PC-lint Plus | 1.4 | 421, 498 | Partially supported |
| Polyspace Bug Finder | R2022b | CERT C: Rule STR31-C | Checks for:   * Use of dangerous standard function * Missing null in string array * Buffer overflow from incorrect string format specifier * Destination buffer overflow in string manipulation * Insufficient destination buffer size   Rule partially covered. |
| PRQA QA-C |  | 5009, 5038, 2840, 2841, 2842, 2843, 2845, 2846, 2847, 2848, 2930, 2931, 2932, 2933, 2935, 2936, 2937, 2938 | Partially implemented |
| PRQA QA-C++ |  | 0145, 2840, 2841, 2842, 2843, 2845, 2846, 2847, 2848, 2930, 2931, 2932, 2933, 2935, 2936, 2937, 2938, 5006, 5038 |  |
| PVS-Studio |  | V518, V645, V727, V755 |  |
| Splint |  |  |  |
| TrustInSoft Analyzer |  | mem\_access | Exhaustively verified (see one compliant and one non-compliant example). |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | Applications that contain SQL query components from an unreliable source are vulnerable to SQL injection. Without safeguards, the fraudulently altered query from the untrusted source could cause information leakage or data manipulation. Sanitization and validation, which are frequently used as parameterized queries and stored procedures, are the main ways to avoid SQL injection.  A value of the pointer type can be produced by adding or subtracting expressions with integral types from a pointer. The behavior of the additive operator is undefinable if the resulting pointer is not a valid member of the container or if it points past the final element of the container.  Rec 07:  STR02-C. Sanitize data passed to complex subsystems |

| **Noncompliant Code** |
| --- |
| Data sanitization requires an understanding of the data being passed and the capabilities of the subsystem. John Viega and Matt Messier provide 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() [Viega 2003]:  The risk, of course, is that the user enters the following string as an email address: |
| sprintf(buffer, "/bin/mail %s < /tmp/email", addr);  system(buffer); |

| **Compliant Code** |
| --- |
| It is necessary to ensure that all valid data is accepted, while potentially dangerous data is rejected or sanitized. Doing so can be difficult when valid characters or sequences of characters also have special meaning to the subsystem and may involve validating the data against a grammar. In cases where there is no overlap, whitelisting can be used to eliminate dangerous characters from the data.  The whitelisting approach to data sanitization is to define a list of acceptable characters and remove any character that is not acceptable. The list of valid input values is typically a predictable, well-defined set of manageable size. This compliant solution, based on the tcp\_wrappers package written by Wietse Venema, 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):** Validate input data:Validate all input from unreliable data sources, at the very least. Most software vulnerabilities can be eliminated with proper input validation. Most external data sources, such as command line arguments, network interfaces, environmental variables, and user-controlled files, should be regarded with suspicion. Architect and design for security policies: Design your program with a software architecture to implement and enforce security policies. Consider segmenting your system into discrete, communicative subsystems, each with the proper set of privileges, if, for instance, your system requires various privileges at various times. Practice defense in depth: Manage risk by employing several defensive tactics such that, in the event that one layer of defense proves insufficient, a different layer of defense can stop a security fault from developing into an exploitable vulnerability and/or restrict the effects of a successful exploit. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.04 |  | Supported by stubbing/taint analysis |
| CodeSonar | 7.1p0 | 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 | 6.5 | TAINTED\_STRING | Fully implemented |
| Klocwork | 2022.3 | NNTS.TAINTED  SV.TAINTED.INJECTION |  |
| LDRA tool suite | 9.7.1 | 108 D, 109 D | Partially implemented |
| Parasoft C/C++test | 2022.1 | 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 | 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** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Programming errors frequently result from dynamic memory management, which might result in security problems. Poor memory management can result in security problems including double-frees, dangling pointers, and heap-buffer overflows. Allocating memory, reading from and writing to memory, and deallocating memory are all aspects of memory management from the standpoint of a programmer.  Rules:  MEM50-CPP. Do not access freed memory  EXP53-CPP. Do not read uninitialized memory  MEM52-CPP. Detect and handle memory allocation errors  MEM31-C. Free dynamically allocated memory when no longer needed |

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

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

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

| **Principles(s):**Architect and Design for Security Policies:Design your program with a software architecture to implement and enforce security policies. Adopt a Secure Coding Standard: Implement secure coding standards for your chosen development languages and platforms. Use Effective Quality Assurance Techniques: Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | dangling\_pointer\_use |  |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 |  |
| Clang | 3.9 | clang-analyzer-cplusplus.NewDelete  clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 7.1p0 | ALLOC.UAF | Use after free |
| Compass/ROSE |  |  |  |
| Coverity | v7.5.0 | USE\_AFTER\_FREE | Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer |
| Helix QAC | 2022.3 | C++4303, C++4304 |  |
| Klocwork | 2022.3 | UFM.DEREF.MIGHT  UFM.DEREF.MUST  UFM.FFM.MIGHT  UFM.FFM.MUST  UFM.RETURN.MIGHT  UFM.RETURN.MUST  UFM.USE.MIGHT  UFM.USE.MUST |  |
| LDRA tool suite | 9.7.1 | 483 S, 484 S | Partially implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-MEM50-a | Do not use resources that have been freed |
| Parasoft Insure++ |  |  | Runtime detection |
| Polyspace Bug Finder | R2022b | CERT C++: MEM50-CPP | Checks for:   * Pointer access out of bounds * Deallocation of previously deallocated pointer * Use of previously freed pointer   Rule partially covered. |
| PRQA QA-C++ | 4.4 | 4303, 4304 |  |
| PVS-Studio | 7.21 | V586, V774 |  |
| Splint | 5.0 |  |  |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assertions are normally disabled prior to code deployment and are primarily used for debugging purposes by using the NDEBUG macro (typically as a flag passed to the compiler). As a result, assertions shouldn't be used for runtime error checking but rather to guard against false programming assumptions.  Assertions should never be used to verify the absence of runtime (as opposed to logic) errors, such as   * Invalid user input (including command-line arguments and environment variables) * File errors (for example, errors opening, reading or writing files) * Network errors (including network protocol errors) * Out-of-memory conditions (for example, malloc() or similar failures) * System resource exhaustion (for example, out-of-file descriptors, processes, threads) * System call errors (for example, errors executing files, locking or unlocking mutexes) * Invalid permissions (for example, file, memory, user)   Rec 48:  MSC11-C. Incorporate diagnostic tests using assertions |

| **Noncompliant Code** |
| --- |
| This noncompliant code example uses the assert() macro to verify that memory allocation succeeded. Because memory availability depends on the overall state of the system and can become exhausted at any point during a process lifetime, a robust program must be prepared to gracefully handle and recover from its exhaustion. Consequently, using the assert() macro to verify that a memory allocation succeeded would be inappropriate because doing so might lead to an abrupt termination of the process, opening the possibility of a denial-of-service attack. See also MEM11-C. Do not assume infinite heap space and void MEM32-C. Detect and handle memory allocation errors. |
| 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):** Architect and Design for Security Policies: Design your program with a software architecture to implement and enforce security policies. Consider segmenting your system into discrete, communicative subsystems, each with the proper set of privileges, if, for instance, your system requires various privileges at various times. Adopt a Secure Coding Standards: Implement secure coding standards for your chosen development languages and platforms. Use Effective Quality Assurance Techniques: Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. Systems that have undergone independent security examinations may be more secure. In identifying and correcting false assumptions, for instance, external reviewers offer an unbiased viewpoint. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | 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 a side effect |
| Parasoft C/C++test | 2022.1 | CERT\_C-MSC11-a | Assert liberally to document internal assumptions and invariants |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | When exceptions are thrown, reclaiming resources is crucial. Throwing an exception could cause cleaning code to be skipped or leave an object only partially initialized. Such a partially initialized object would violate basic exception safety, as described in ERR56-CPP. Guarantee exception safety. When objects leave their scope, it is ideal that resources be automatically released using the RAII design pattern [Stroustrup 2001]. By using this method, you can allocate resources without having to write any complicated cleanup code. Constructors, however, do not provide the same level of security. Because a constructor is engaged in resource allocation, if it stops early, no resources it allocated will be immediately released.  Rule 8:  ERR57-CPP. Do not leak resources when handling exceptions  ERR59-CPP. Do not throw an exception across execution boundaries  ERR61-CPP. Catch exceptions by lvalue reference  ERR51-CPP. Handle all exceptions  ERR56-CPP. Guarantee exception safety |

| **Noncompliant Code** |
| --- |
| The following noncompliant code example shows a flawed copy assignment operator. The implicit invariants of the class are that the array member is a valid (possibly null) pointer and that the nElems member stores the number of elements in the array pointed to by array. The function deallocates array and assigns the element counter, nElems, before allocating a new block of memory for the copy. As a result, if the new expression throws an exception, the function will have modified the state of both member variables in a way that violates the implicit invariants of the class. Consequently, such an object is in an indeterminate state and any operation on it, including its destruction, results in undefined behavior. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }      IntArray(const IntArray& that); // nontrivial copy constructor  IntArray& operator=(const IntArray &rhs) {  if (this != &rhs) {  delete[] array;  array = nullptr;  nElems = rhs.nElems;  if (nElems) {  array = new int[nElems];  std::memcpy(array, rhs.array, nElems \* sizeof(\*array));  }  }  return \*this;  }    // ...  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the copy assignment operator provides the strong exception safety guarantee. The function allocates new storage for the copy before changing the state of the object. Only after the allocation succeeds does the function proceed to change the state of the object. In addition, by copying the array to the newly allocated storage before deallocating the existing array, the function avoids the test for self-assignment, which improves the performance of the code in the common case [Sutter 2004]. |
| #include <cstring>    class IntArray {  int \*array;  std::size\_t nElems;  public:  // ...    ~IntArray() {  delete[] array;  }    IntArray(const IntArray& that); // nontrivial copy constructor    IntArray& operator=(const IntArray &rhs) {  int \*tmp = nullptr;  if (rhs.nElems) {  tmp = new int[rhs.nElems];  std::memcpy(tmp, rhs.array, rhs.nElems \* sizeof(\*array));  }  delete[] array;  array = tmp;  nElems = rhs.nElems;  return \*this;  }    // ...  }; |

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

| **Principles(s):**): Architect and Design for Security Policies:Design your program with a software architecture to implement and enforce security policies. Consider segmenting your system into discrete, communicative subsystems, each with the proper set of privileges, if, for instance, your system requires various privileges at various times.  Adopt a Secure Coding Standard: Implement secure coding standards for your chosen development languages and platforms. Use Effective Quality Assurance Techniques: Effective quality assurance procedures can help find and fix vulnerabilities. An efficient quality assurance program should include source code audits, fuzz testing, and penetration testing. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | ALLOC.LEAK | Leak |
| Helix QAC | 2022.3 | C++4075, C++4076 |  |
| LDRA tool suite | 9.7.1 | 527 S, 56 D, 71 D | Partially implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-ERR56-a  CERT\_CPP-ERR56-b | Always catch exceptions  Do not leave 'catch' blocks empty |
| Polyspace Bug Finder | R2022b | CERT C++: ERR56-CPP | Checks for exceptions violating class invariant (rule fully covered). |
| PRQA QA-C++ | 4.4 | 4075, 4076 |  |
| PVS-Studio | 7.21 | V565, V1023, V5002 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Object Oriented Programming | [STD-008-CPP] | Because member function calls can be made using virtual functions, the choice of member function calls can be made at runtime based on the dynamic type of the object being called. This approach facilitates object-oriented programming techniques like object inheritance and function overriding that are widely used. The specified function is called if a nonvirtual member function is called or a class member access expression is used to indicate a call. If not, a virtual function call is made to the last overrider in the object expression's dynamic type.  Rule 09. Object Oriented Programming (OOP):   * OOP50-CPP. Do not invoke virtual functions from constructors or destructors * OOP51-CPP. Do not slice derived objects * OOP52-CPP. Do not delete a polymorphic object without a virtual destructor * OOP53-CPP. Write constructor member initializers in the canonical order * OOP54-CPP. Gracefully handle self-copy assignment * OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members * OOP56-CPP. Honor replacement handler requirements * OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions * OOP58-CPP. Copy operations must not mutate the source object |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, the base class attempts to seize and release an object's resources through calls to virtual functions from the constructor and destructor. However, the B::B() constructor calls B::seize() rather than D::seize(). Likewise, the B::~B() destructor calls B::release() rather than D::release(). |
| struct B {  B() { seize(); }  virtual ~B() { release(); }    protected:  virtual void seize();  virtual void release();  };    struct D : B {  virtual ~D() = default;    protected:  void seize() override {  B::seize();  // Get derived resources...  }    void release() override {  // Release derived resources...  B::release();  }  }; |

| **Compliant Code** |
| --- |
| In this compliant solution, the constructors and destructors call a nonvirtual, private member function (suffixed with mine) instead of calling a virtual function. The result is that each class is responsible for seizing and releasing its own resources. |
| class B {  void seize\_mine();  void release\_mine();    public:  B() { seize\_mine(); }  virtual ~B() { release\_mine(); }    protected:  virtual void seize() { seize\_mine(); }  virtual void release() { release\_mine(); }  };    class D : public B {  void seize\_mine();  void release\_mine();    public:  D() { seize\_mine(); }  virtual ~D() { release\_mine(); }    protected:  void seize() override {  B::seize();  seize\_mine();  }    void release() override {  release\_mine();  B::release();  }  }; |

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

| **Principles(s):** Architect and design for security policies: Create a software architecture and design your software to implement and enforce security policies.Adopt a secure coding standard: Develop and/or apply a secure coding standard for your target development language and platform. 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. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | Medium | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 22.10 | virtual-call-in-constructor  invalid\_function\_pointer | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP50 |  |
| Clang | 3.9 | clang-analyzer-  alpha.cplusplus.VirtualCall | [Insert text.]Checked by clang-tidy |
| CodeSonar | 7.1p0 | LANG.STRUCT.VCALL\_IN\_CTOR  LANG.STRUCT.VCALL\_IN\_DTOR | Virtual Call in Constructor  Virtual Call in Destructor |
| Helix QAC | 2022.3 | C++4260, C++4261, C++4273, C++4274, C++4275, C++4276, C++4277, C++4278, C++4279, C++4280, C++4281, C++4282 |  |
| Klocwork | 2022.3 | CERT.OOP.CTOR.VIRTUAL\_FUNC |  |
| LDRA tool suite | 9.7.1 | 467 S, 92 D | Fully implemented |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-OOP50-a  CERT\_CPP-OOP50-b  CERT\_CPP-OOP50-c  CERT\_CPP-OOP50-d | Avoid calling virtual functions from constructors  Avoid calling virtual functions from destructors  Do not use dynamic type of an object under construction  Do not use dynamic type of an object under destruction |
| Polyspace Bug Finder | R2022b | CERT C++: OOP50-CPP | Checks for virtual function call from constructors and destructors (rule fully covered) |
| PRQA QA-C++ | 4.4 | 4260, 4261, 4273, 4274,  4275, 4276, 4277, 4278,  4279, 4280, 4281, 4282 |  |
| PVS-Studio | 7.21 | V1053 |  |
| RuleChecker | 22.10 | virtual-call-in-constructor | Fully checked |
| SonarQube C/C++ Plugin | 4.10 | S1699 |  |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Application Programming Interfaces (API) | [STD-009-CPP] | The primary issue with redundant testing by caller and callee as a sort of defensive programming is performance, which has led to widespread disapproval in the C and C++ communities. Validation is typically only required on one side of each interface in C and C++.  This standard suggests that the called function validate its parameters for purposes of safety and security. The function can survive at least certain types of erroneous usage thanks to validity checks, which also helps an application that uses the function to survive. Determining the circumstance that led to the invalid parameter can be made easier with the use of validity checks.  Rec 13:  API00-C. Functions should validate their parameters |

| **Noncompliant Code** |
| --- |
| In this noncompliant code example, setfile() and usefile() do not validate their parameters. It is possible that an invalid file pointer can be used by the library, corrupting the library's internal state and exposing a vulnerability. |
| /\* Sets some internal state in the library \*/  extern int setfile(FILE \*file);    /\* Performs some action using the file passed earlier \*/  extern int usefile();    static FILE \*myFile;    void setfile(FILE \*file) {  myFile = file;  }    void usefile(void) {  /\* Perform some action here \*/  } |

| **Compliant Code** |
| --- |
| Validating the function parameters and verifying the internal state leads to consistency of program execution and may eliminate potential vulnerabilities. In addition, implementing commit or rollback semantics (leaving program state unchanged on error) is a desirable practice for error safety. |
| /\* Sets some internal state in the library \*/  extern errno\_t setfile(FILE \*file);    /\* Performs some action using the file passed earlier \*/  extern errno\_t usefile(void);    static FILE \*myFile;    errno\_t setfile(FILE \*file) {  if (file && !ferror(file) && !feof(file)) {  myFile = file;  return 0;  }    /\* Error safety: leave myFile unchanged \*/  return -1;  }    errno\_t usefile(void) {  if (!myFile) return -1;    /\*  \* Perform other checks if needed; return  \* error condition.  \*/    /\* Perform some action here \*/  return 0;  } |

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

| **Principles(s):** 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. Use effective quality assurance techniques: Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Adopt a secure coding standard: Develop and/or apply a secure coding standard for your target development language and platform. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | High | P2 | L3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 7.1p0 | LANG.STRUCT.UPD | Unchecked parameter dereference |
| Parasoft C/C++test | 2022.1 | CERT\_C-API00-a | The validity of parameters must be checked inside each function |
| PC-lint Plus | 1.4 | 413, 613, 668 | Partially supported: reports use of null pointers including function parameters which are assumed to have the potential to be null |
| PVS-Studio | 7.21 | V781 |  |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-010-CPP] | The execution environment can be changed in C++ by calling a library I/O function, altering an object, accessing a volatile-qualified variable, or invoking a function that does one of these things. We refer to these behaviors as side effects. In terms of the order of their evaluations, any links between value computations and side effects can be characterized.  Rule 02. Expressions (EXP):   * EXP50-CPP. Do not depend on the order of evaluation for side effects * EXP51-CPP. Do not delete an array through a pointer of the incorrect type * EXP52-CPP. Do not rely on side effects in unevaluated operands * EXP53-CPP. Do not read uninitialized memory * EXP54-CPP. Do not access an object outside of its lifetime * EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type * EXP56-CPP. Do not call a function with a mismatched language linkage * EXP57-CPP. Do not cast or delete pointers to incomplete classes * EXP58-CPP. Pass an object of the correct type to va\_start * EXP59-CPP. Use offsetof() on valid types and members * EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries * EXP61-CPP. A lambda object must not outlive any of its reference captured objects * EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation * EXP63-CPP. Do not rely on the value of a moved-from object |

| **Noncompliant Code** |
| --- |
| The order of evaluation for function arguments is unspecified. This noncompliant code example exhibits unspecified behavior but not undefined behavior. |
| extern void c(int i, int j);  int glob;    int a() {  return glob + 10;  }    int b() {  glob = 42;  return glob;  }    void f() {  c(a(), b());  } |

| **Compliant Code** |
| --- |
| In this compliant solution, the order of evaluation for a() and b() is fixed, and so no unspecified behavior occurs. |
| extern void c(int i, int j);  int glob;    int a() {  return glob + 10;  }    int b() {  glob = 42;  return glob;  }    void f() {  int a\_val, b\_val;    a\_val = a();  b\_val = b();    c(a\_val, b\_val);  } |

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

| **Principles(s):** Validate input. Validate input from all untrusted data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be suspicious of most external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files. Heed compiler warnings: Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A]. Use static and dynamic analysis tools to detect and eliminate additional security flaws. Architect and design for security policies: Create a software architecture and design your software to implement and enforce security policies. Use effective quality assurance techniques: Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Adopt a secure coding standard: Develop and/or apply a secure coding standard for your target development language and platform. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-EXP50 |  |
| Clang | 3.9 | -Wunsequenced | Can detect simple violations of this rule where path-sensitive analysis is not required |
| CodeSonar | 7.1p0 | LANG.STRUCT.SE.DEC  LANG.STRUCT.SE.INC | Side Effects in Expression with Decrement  Side Effects in Expression with Increment |
| Compass/ROSE |  |  | Can detect simple violations of this rule. It needs to examine each expression and make sure that no variable is modified twice in the expression. It also must check that no variable is modified once, then read elsewhere, with the single exception that a variable may appear on both the left and right of an assignment operator |
| Coverity | v7.5.0 | EVALUATION\_ORDER | Can detect the specific instance where a statement contains multiple side effects on the same value with an undefined evaluation order because, with different compiler flags or different compilers or platforms, the statement may behave differently |
| ECLAIR | 1.2 | CC2.EXP30 | Fully implemented |
| GCC | 4.9 |  | Can detect violations of this rule when the -Wsequence-point flag is used |
| Helix QAC | 2022.3 | C++3220, C++3221, C++3222, C++3223, C++3228 |  |
| Klocwork | 2022.3 | PORTING.VAR.EFFECTS  CERT.EXPR.PARENS  MISRA.EXPR.PARENS.INSUFFICIENT  MISRA.INCR\_DECR.OTHER |  |
| LDRA tool suite | 9.7.1 | 35 D, 1 Q, 9 S, 134 S, 67 D, 72 D | Partially implemen |
| Parasoft C/C++test | 2022.1 | CERT\_CPP-EXP50-a  CERT\_CPP-EXP50-b  CERT\_CPP-EXP50-c  CERT\_CPP-EXP50-d  CERT\_CPP-EXP50-e  CERT\_CPP-EXP50-f | The value of an expression shall be the same under any order of evaluation that the standard permits  Don't write code that depends on the order of evaluation of function arguments  Don't write code that depends on the order of evaluation of function designator and function arguments  Don't write code that depends on the order of evaluation of expression that involves a function call  Between sequence points an object shall have its stored value modified at most once by the evaluation of an expression  Don't write code that depends on the order of evaluation of funct |
| Polyspace Bug Finder | R2022b | CERT C++: EXP50-CPP | Checks for situations where expression value depends on order of evaluation (rule fully covered). |
| PRQA QA-C++ | 4.4 | 3220, 3221, 3222, 3223, 3228 |  |
| PVS-Studio | 7.21 | V521, V708 |  |
| SonarQube C/C++ Plugin | 4.10 | IncAndDecMixedWithOtherOperators | Partially implemen |
| Splint | 5.0 |  |  |

### 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.

[Insert your written explanations here.]

### 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 | Likely | High | P9 | L2 |
| STD-002-CPP | High | Likely | High | P9 | L2 |
| STD-003-CPP | High | Likely | Medium | P18 | L1 |
| STD-004-CPP | High | Likely | Medium | P18 | L1 |
| STD-005-CPP | High | Likely | Medium | P18 | L1 |
| STD-006-CPP | Low | Unlikely | High | P1 | L3 |
| STD-007-CPP | High | Likely | High | P9 | L2 |
| STD-008-CPP | Low | Unlikely | Medium | P2 | L3 |
| STD-009-CPP | Medium | Unlikely | High | P2 | L3 |
| STD-010-CPP | Medium | Probable | Medium | P8 | L2 |

### 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 | By guaranteeing that the data is encrypted when it is on disk, encryption in rest is intended to stop the attacker from obtaining the unencrypted data. You are essentially transforming your customers' sensitive data into another type of data when you encrypt data while it is at rest. This often occurs using an algorithm that a user without access to the encryption key to decode it cannot comprehend. Your data will remain secure because these files will only be accessible by authorized employees. |
| Encryption at flight | The encryption of data in flight refers to data transmission through a network. This is crucial for people that use the open internet to transport data, which is a component of the majority of public cloud systems. The underlying FASP protocol employs the Cipher Feedback with Checksum technique to guarantee the security and integrity of data in flight each time you send content using an Aspera web app. |
| Encryption in use | Data is deemed to be in use when it is currently accessible and utilised. Data security must be taken care of before actual data use can start since data must be decrypted in order to be put to use. Secure Encrypted Virtualization is a technique for data encryption in use (SEV). It uses specialized hardware, an AMD EPYC processor, and an AES-128 encryption engine to encrypt RAM memory. Although this field is still relatively young, other hardware suppliers are also providing memory encryption for data in use. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Before access to a network device is given, authentication verifies identification. Based on the username and password combination given by the entity attempting to obtain access, it is the process of confirming the identity of the person or device accessing a network device. Passwords, single sign-on (SSO) systems, biometrics, digital certificates, and public key infrastructure can all be used to establish identity. As data theft and threats to information security become more sophisticated, user authentication guarantees that the proper authorization to use a system is provided. Authentication can ensure that network resources are safeguarded through a variety of authentication mechanisms, even though it cannot totally prevent identity theft. |
| Authorization | Access control is provided by authorization. It is the process of putting together a group of characteristics that specify what the user is permitted to do. Authorization often takes place after authentication, when your identity has been verified, AAA security authorization puts together the collection of attributes that characterize what you are permitted to do. User access to a network and its resources is determined by the authorization levels that are assigned to them. A user might, for instance, be able to type commands but only be allowed to execute specific ones. This may be determined by geographical limitations, time or date constraints, login frequency, or numerous logins by a single person. Route assignments, IP address filtering, bandwidth traffic control, and encryption are other forms of authorization. Even though they may have privileged access, administrators may still be limited in what they can do. |
| Accounting | Accounting offers a way to gather data, record it locally on a network device, and transfer it to a AAA server for billing, auditing, and reporting. The accounting function keeps track of each management session that is used to grant access and keeps a record of it. This data can be used to create reports for auditing and troubleshooting tasks. Accounting makes sure that during an audit, administrators will be able to log in and see what activities were taken, by whom, and when. The real accounting records must be stored on an external AAA security server, which is one restriction of the accounting part of AAA security. Network and system administrators can examine who has been trying to access what and if access has been allowed thanks to proper accounting. |

**\***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 |  |
| 1.5 | 12/05/2022 | Updated | Concepcion Ardon | [Insert text.] |
| [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] | [Insert text.] |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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