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



# Green Pace Secure Development Policy

## Contents

[Overview 1](#_Toc52464053)

[Purpose 1](#_Toc52464054)

[Scope 1](#_Toc52464055)

[Module Three Milestone 1](#_Toc52464056)

[Ten Core Security Principles 1](#_Toc52464057)

[C/C++ Ten Coding Standards 1](#_Toc52464058)

[Coding Standard 1 1](#_Toc52464059)

[Coding Standard 2 1](#_Toc52464060)

[Coding Standard 3 1](#_Toc52464061)

[Coding Standard 4 1](#_Toc52464062)

[Coding Standard 5 1](#_Toc52464063)

[Coding Standard 6 1](#_Toc52464064)

[Coding Standard 7 1](#_Toc52464065)

[Coding Standard 8 1](#_Toc52464066)

[Coding Standard 9 1](#_Toc52464067)

[Coding Standard 10 1](#_Toc52464068)

[Defense-in-Depth Illustration 1](#_Toc52464069)

[Project One 1](#_Toc52464070)

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

[2. Risk Assessment 1](#_Toc52464072)

[3. Automated Detection 1](#_Toc52464073)

[4. Automation 1](#_Toc52464074)

[5. Summary of Risk Assessments 1](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 1](#_Toc52464076)

[7. Map the Principles 1](#_Toc52464077)

[Audit Controls and Management 1](#_Toc52464078)

[Enforcement 1](#_Toc52464079)

[Exceptions Process 1](#_Toc52464080)

[Distribution 1](#_Toc52464081)

[Policy Change Control 1](#_Toc52464082)

[Policy Version History 1](#_Toc52464083)

[Appendix A Lookups 1](#_Toc52464084)

[Approved C/C++ Language Acronyms 1](#_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 | The user should validate input from all data sources. Input validation could eliminate most software vulnerabilities. The user should be suspicious of all external data sources. |
| 1. Heed Compiler Warnings | The user should compile code utilizing the highest warning level possible. The use of dynamic and static analysis tools to eliminate security flaws is also a good idea. |
| 1. Architect and Design for Security Policies | The user should design software that implements and enforces security policies. It is a good idea to divide the system into different subsystems if different privileges are required at different times. |
| 1. Keep It Simple | A user should not over-complicate their code. Making code that is too elaborate increased the changes of errors within that code. It is best to just keep things simple. |
| 1. Default Deny | The user should make access decisions based on permissions. Access should be denied by default, unless the user is meant to access the code. |
| 1. Adhere to the Principle of Least Privilege | A user should design processes that execute utilizing the least amount of privileges as possible. This prevents an attacker from being able to utilize these elevated privileges to gather vital information. |
| 1. Sanitize Data Sent to Other Systems | The user should sanitize data that is sent to complex subsystems. An attacker could potentially invoke unused functionality to access vital information. This is why sanitization is necessary. |
| 1. Practice Defense in Depth | Defense in Depth involves utilizing multiple defensive strategies layered over each other. These defense strategies are completely redundant which results in an attacker wasting a lot of time on them for nothing. This helps keep vital code secure. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance techniques can be very useful when developing secure code. These involve, essentially, attempting to attack the code to see what can be exploited. The user would then utilize this information to revamp on areas that need more security. |
| 1. Adopt a Secure Coding Standard | A user should develop and/or apply a secure coding standard for their 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** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | [STD-001-CPP] | Integer data types must be correctly ranged to prevent the wrapping (overflow) between signed and unsigned integer conversions. A conversion of value in an unsigned integer type to a narrower width can result in a truncation error. This is because if the integer can not be represented in the new type, data loss will occur. |

| **Noncompliant Code** |
| --- |
| If a value is too small to represent, truncation can occur. This can cause conversions the can result in values out of range. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 sc = (signed char)ul; /\* cast eliminates warning \*/ |

| **Compliant Code** |
| --- |
| Ranges are validated when converting from an unsigned type to a signed type. |
| 1 unsigned long int ul = ULONG\_MAX;  2 signed char sc;  3 if (ul <= SCHAR\_MAX) {  4 sc = (signed char)ul; /\* use cast to eliminate warning \*/  5 }  6 else {  7 /\* handle error condition \*/  8 } |

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

| **Principles(s):** Validate Input Data to prevent issues and use Effective Quality Assurance Techniques to mitigate problems using tools. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | 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 |
| CodeSonar | 6.1p0 | 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 |
| Polyspace Bug Finder | R2021a | CERT C: Rule INT30-C | Checks for:  Unsigned integer overflow  Unsigned integer constant overflow  Rule partially covered. |
| Coverity | 2017.07 | INTEGER\_OVERFLOW | Implemented |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-CPP] | It is important to remember that unsigned integer values should represent values that cannot become negative, whereas signed integers should represent values that can become negative. So, it is efficient to use the smallest type that can represent the range of values for the variable that is being worked on. |

| **Noncompliant Code** |
| --- |
| This code fails to consider that the unsigned integer value will wrap around. (infinite loop) |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3  4 for (unsigned int i = cnt-2; i >= 0; i--) {  5 a[i] += a[i+1];  6 } |

| **Compliant Code** |
| --- |
| As size\_t is an unsigned type, this behavior is well defined by the standard to be modulo. |
| 1 char a[MAX\_ARRAY\_SIZE] = /\* initialize \*/;  2 size\_t cnt = /\* initialize \*/;  3  4 for (size\_t i = cnt-2; i != SIZE\_MAX; i--) {  5 a[i] += a[i+1];  6 } |

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

| **Principles(s):** By heeding compiler warnings, utilizing quality assurance techniques, and validating input data, we can solve numerous issues such as: using QA analysis to catch issues such as this one, heeding compiler warnings to catch issues before they arise, and validating input data to prevent memory errors. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | 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 |
| Coverity | 2017.07 | TAINTED\_SCALAR  BAD\_SHIFT | Implemented |
| TrustInSoft Analyzer | 1.38 | Signed\_overflow | Exhaustively verified |
| Helix QAC | 2021.2 | C2800, C2801, C2802, C2803, C2860, C2861, C2862, C2863  C++2800, C++2801, C++2802, C++2803, C++2860, C++2861, C++2862, C++2863 | Implemented |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-CPP] | Buffer overflows can result from neglected buffer boundaries and incorrect string sizes. It is wise to not copy data from an unbound source as this can result in these issues. |

| **Noncompliant Code** |
| --- |
| If a user inputs more than 11 characters, it will result in an out-of-bounds write. |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[12];  5  6 std::cin >> buf;  7 std::cout << "echo: " << buf << '\n';  8 } |

| **Compliant Code** |
| --- |
| Eliminates the overflow in the previous example by setting the field width member to the size of  the character array buffer. |
| 1 #include <iostream>  2  3 int main(void) {  4 char buf[12];  5  6 std::cin.width(12);  7 std::cin >> buf;  8 std::cout << "echo: " << buf << '\n';  9 }06 string str;  07 string::iterator i;  08 for (i = str.begin(); i != str.end(); ++i) {  09 cout << \*i  10 } |

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

| **Principles(s):** By heeding compiler warnings of overflow, utilizing QA techniques, sanitizing data to prevent string attacks, and validating input with appropriate string function, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |
| CodeSonar | 6.1p0 | 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 |
| Parasoft C/C++test | 2021.1 | 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 |
| Coverity | 2017.07 | STRING\_OVERFLOW  BUFFER\_SIZE  OVERRUN  STRING\_SIZE | Fully implemented |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-CPP] | There are multiple points of attack for a string. These include environmental variables, console input, command-line arguments, text files, and network connections. All of these points of attack can be used by a hacker to strategically launch an SQL Injection attack to cause overflow.  To limit these points of attack, it is wise to review all code that utilizes EXECUTE commands as well as testing the size and data type of input to properly enforce the necessary limits. |

| **Noncompliant Code** |
| --- |
| Unfiltered code is vulnerable to SQL injection through user input |
| SqlDataAdapter myCommand =  new SqlDataAdapter("LoginStoredProcedure '" +  Login.Text + "'", conn); |

| **Compliant Code** |
| --- |
| Using the Parameters collection with Dynamic SQL |
| SqlDataAdapter myCommand = new SqlDataAdapter(  "SELECT au\_lname, au\_fname FROM Authors WHERE au\_id = @au\_id", conn);  SQLParameter parm = myCommand.SelectCommand.Parameters.Add("@au\_id",  SqlDbType.VarChar, 11);  Parm.Value = Login.Text; |

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

| **Principles(s):** By using a defense in depth strategy such as a layered defensive strategy, validating input to ensure query parameterization, and utilizing architect and design for security policies, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | 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 |
| CodeSonar | 6.1p0 | 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 |
| TrustInSoft Analyzer | 1.38 | Mem\_access | Exhaustively verified |
| Parasoft C/C++test | 2021.1 | 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 |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-CPP] | Stack overflows are a lot of times caused by stack smashing. This involves an attacker utilizes a security bug to exploit the stack and fill it with data that the attacker has supplied. This corrupts the stack, allowing the user to take control of the program in runtime. Attackers can also overwrite function pointers to transfer control to the attacker. A good way to limit this kind of attack is to store encrypted pointers and to eliminate buffer overflow conditions. |

| **Noncompliant Code** |
| --- |
| Vulnerable code shows that s is dereferenced after it has been deallocated. If this were to result in “write after free”, the vulnerability can be exploited to run arbitrary code (using the vulnerable process permissions). |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  delete s;  // ...  s->f();  } |

| **Compliant Code** |
| --- |
| The solution is to wait until the memory is no longer required to deallocate. |
| #include <new>    struct S {  void f();  };    void g() noexcept(false) {  S \*s = new S;  // ...  s->f();  delete s;  } |

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

| **Principles(s):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Polyspace Bug Finder | R2021a | CERT C++: MEM52-CPP | Checks for unprotected dynamic memory allocation (rule partially covered) |
| Helix QAC | 2021.2 | C++3225, C++3226, C++3227, C++3228, C++3229, C++4632 |  |
| Coverity | 7.5 | CHECKED\_RETURN | Finds inconsistencies in how function call return values are handled |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Check the return value of new  Do not allocate resources in function argument list because the order of evaluation of a function's parameters is undefined |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-CPP] | Assert statements should be used to handle errors that test for conditions that should not be allowed in the program. Exceptions are not quite necessary as assertions indicate code that needs to be fixed. |

| **Noncompliant Code** |
| --- |
| By relying only on the assertion to handle the error condition, any error code returned by this myGraphRoutine will be unhandled come final release. |
| myErr = myGraphRoutine(a, b);  /\* No Code to handle errors \*/  ASSERT(!myErr); // Don't do this!  \_ASSERT(!myErr); // Don't do this, either! |

| **Compliant Code** |
| --- |
| In this example, a graphic routine returns an error code or zero for success. If the error-handling code works properly, the error should be handled and myErr reset to zero before the assertion is reached. If myErr has another value, the assertion fails, the program halts, and the Assertion Failed Dialog Box appears. |
| myErr = myGraphRoutine(a, b);  /\* Code to handle errors and  reset myErr if successful \*/  ASSERT(!myErr); -- MFC version  \_ASSERT(!myErr); -- CRT version |

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

| **Principles(s):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | CERT\_C-MSC11-a | CERT\_C-MSC11-a |
| CodeSonar | 6.1p0 | 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 |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-CPP] | Exceptions should be thrown when a function call has been preventing from fulfilling its function. Exceptions can be used to check for errors that may occur within public functions. To ensure that the compiler can take care of the memory management for the exception object, it is wise to throw an exception by value and catch it by reference. |

| **Noncompliant Code** |
| --- |
| As per SEI Cert C++ Coding Standard:  This example shows a *flawed copy assignment operator*.  The **implicit invariants** of the class: 1) the array member is a valid (possibly null) pointer and 2) 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** |
| --- |
| 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. |
| #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):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| RuleChecker | 20.10 | main-function-catch-all  early-catch-all | Partially checked |
| Polyspace Bug Finder | R2021a | CERT C++: ERR51-CPP | Checks for unhandled exceptions (rule partially covered) |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-ERR51-a  CERT\_CPP-ERR51-b | Always catch exceptions  Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point |
| Helix QAC | 2021.2 | C++4035, C++4036, C++4037 |  |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Expressions | [STD-008-CPP] | Problems can arise if objects are moved from another area or memory is not initialized. Errors can occur if local variables are read before they are initialized. |

| **Noncompliant Code** |
| --- |
| An uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior. |
| #include <iostream>    void f() {  int i;  std::cout << i;  } |

| **Compliant Code** |
| --- |
| The object is initialized prior to printing its value. |
| #include <iostream>    void f() {  int i = 0;  std::cout << i;  } |

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

| **Principles(s):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-EXP54-a  CERT\_CPP-EXP54-b  CERT\_CPP-EXP54-c | CERT\_CPP-EXP54-c    Do not use resources that have been freed  The address of an object with automatic storage shall not be returned from a function  The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist |
| CodeSonar | |  |  | | --- | --- | |  | 6.1p0 | | IO.UAC  ALLOC.UAF | Use after close  Use after free |
| Polyspace Bug Finder | R2021a | CERT C++: EXP54-CPP | Checks for:  Non-initialized variable or pointer  Use of previously freed pointer  Pointer or reference to stack variable leaving scope  Accessing object with temporary lifetime  Rule partially covered. |
| Clang | 3.9 | -Wdangling-initializer-list | Catches some lifetime issues related to incorrect use of std::initializer\_list<> |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| API Vulnerability | [STD-009-CPP] | An Application Programming Interface (API) is a type of software interface that offers service to other pieces of software. When using APIs, a user must be wary of the vulnerabilities and exploits within one. It is wise to utilize and open-source API as a large number of programmers can point out vulnerabilities which will aid the user when creating their program. |

| **Noncompliant Code** |
| --- |
| Vulnerabilities assessed in image library:  CVE-2019-13499: A heap buffer overflow caused by a negative-size memcpy/memset in psdParser::UnpackRLE in the psdThumbnail component of FreeImage 3.18.0 allows an attacker to cause a denial of service via a crafted PSD file.  CVE-2019-13500: A heap buffer overflow in psdThumbnail::Read in the psdThumbnail component of FreeImage 3.18.0 allows an attacker to cause a denial of service or execute arbitrary code via a crafted PSD file.  CVE-2019-13501: A heap buffer overflow in psdParser::ReadImageLine in the psdParser component of FreeImage 3.18.0 allows an attacker to cause a denial of service or execute arbitrary code via a crafted PSD file. |
| @@ -1320,24 +1330,26 @@ void psdParser::UnpackRLE(BYTE\* line, const BYTE\* rle\_line, BYTE\* line\_end, unsi    // (len + 1) bytes of data are copied  ++len;  + // ensure we don't write beyound eol  + size = line + len > line\_end ? line\_end - line : len;    - // assert we don't write beyound eol  - memcpy(line, rle\_line, line + len > line\_end ? line\_end - line : len);  - line += len;  - rle\_line += len;  - srcSize -= len;  + memcpy(line, rle\_line, size);  + line += size;  + rle\_line += size;  + srcSize -= size; |

| **Compliant Code** |
| --- |
| Target for Libfuzzer which helped uncover these vulnerabilities: |
| #include "FreeImage.h"  extern "C" int LLVMFuzzerTestOneInput(const uint8\_t \*data, size\_t size) {  FIMEMORY \*mem;  FIBITMAP \*dib;  int length, height, bpp, y;  FreeImage\_Initialise(true);  mem = FreeImage\_OpenMemory(const\_cast<uint8\_t \*>(data), size);  if (!mem)  return 0;  dib = FreeImage\_LoadFromMemory(FIF\_PSD, mem, PSD\_DEFAULT);  if (!dib)  return 0;  bpp = FreeImage\_GetBPP(dib);  length = FreeImage\_GetWidth(dib);  height = FreeImage\_GetHeight(dib);  for (y = 0; y < height; y++)  FreeImage\_GetScanLine(dib, y);  FreeImage\_Unload(dib);  FreeImage\_CloseMemory(mem);  FreeImage\_DeInitialise();  return 0;  } |

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

| **Principles(s):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| OpenSCAP | 1.3.5 | https://www.open-scap.org | OpenSCAP |
| Metasploit |  | https://www.metasploit.com/ | Metasploit |
| OpenVAS | 21.04 | Open Vulnerability Assessment Scanner. | OpenVAS |
| Wireshark |  | https://www.wireshark.org | Wireshark |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| OOP | [STD-010-CPP] | Object Oriented Programming (OOP) is a form of programming that is based on utilizing objects to contain data and code. OOP is mistakenly thought to be more secure than functional programming, but that is not necessarily true. This is completely up to the programmer. It is wise to understand the limitations and possible exploitations of OOP. |

| **Noncompliant Code** |
| --- |
| The copy operations for A *mutate the source* operand by resetting its member variable m to 0.  When std::fill() is called, the first element copied will have the original value of **obj.m**, *12*, at which point obj.m is set to 0. The subsequent nine copies will all retain the value 0. |
| #include <algorithm>  #include <vector>    class A {  mutable int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {  other.m = 0;  }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  other.m = 0;  }  return \*this;  }    int get\_m() const { return m; }  };    void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

| **Compliant Code** |
| --- |
| The copy operations for A *no longer mutate the source operand*, ensuring that the vector contains equivalent copies of obj. Instead, A has been *given move operations* that perform the mutation when it is safe to do so. |
| #include <algorithm>  #include <vector>    class A {  int m;    public:  A() : m(0) {}  explicit A(int m) : m(m) {}    A(const A &other) : m(other.m) {}  A(A &&other) : m(other.m) { other.m = 0; }    A& operator=(const A &other) {  if (&other != this) {  m = other.m;  }  return \*this;  }    A& operator=(A &&other) {  m = other.m;  other.m = 0;  return \*this;  }    int get\_m() const { return m; }  };    void f() {  std::vector<A> v{10};  A obj(12);  std::fill(v.begin(), v.end(), obj);  } |

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

| **Principles(s):** By utilizing QA techniques to test for vulnerabilities, training to write memory allocation routines, and adopting a secure coding standard to ensure consistency, we can solve issues such as this one. |
| --- |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Helix QAC | 2021.2 | C++2810, C++2811, C++2812, C++2813, C++2814 | Helix QAC |
| Astrée | 20.10 | overflow\_upon\_dereference  invalid\_function\_pointer | Astrée |
| Parasoft C/C++test | 2021.1 | CERT\_CPP-OOP55-a | A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-OOP55 |  |

### 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 | Unlikely | Medium | 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 | Low | Probable | Medium | P4 | L3 |
| STD-008-CPP | High | Probable | High | P6 | L2 |
| STD-009-CPP | High | Probable | Medium | P4 | L3 |
| STD-010-CPP | High | Probable | High | P6 | 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 | Encryption at rest is utilized to prevent an attacker from accessing data by ensuring it is encrypted on disk. This means that a hacker may obtain an encrypted hard drive, but, without the encryption keys, cannot read the encrypted data. |
| Encryption at flight | Encrypting data at flight means to encrypt data that is being transmitted over a network. This means that from the point of leaving one user to another user or server, the data is encrypted so as to prevent an attacker from accessing the network and reading the data. |
| Encryption in use | Encryption in use involves encrypting sensitive data regardless of the source, location, or stage of the data. This method is the most secure formof encryption. It is more difficult to utilize, but it is also more secure than the other forms of encryption. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication involves identifying a user. This means that the data can only be accessed if a user is identified. This usually involves forcing a valid username and password to grant access. |
| Authorization | Authorization involves authorizing the proper user to execute certain tasks. After authenticating a user (logging in), the user may try to issue some commands that they are not authorized to run. These commands will not execute because the user does not have the authorization to run them. This is applied a lot in cases where there are users and moderators, where the moderators have more authority than the baseline user. |
| Accounting | Accounting involves keeping track of what each user does. This could include the time and day the user logged in, data sent to or from the user, and work completed by the user. This information can then be stored in an accounting server. |

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

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

### Map the Principles

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

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

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

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

## Audit Controls and Management

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

Evidence will include the following:

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

## Enforcement

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

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

## Exceptions Process

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

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

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

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

## Distribution

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

## Policy Change Control

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

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 04/05/2022 | Updated All Policies | Zachary Gonzalez | Professor Hodde |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

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