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



Green Pace Secure Development Policy

Contents

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

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

[Scope 2](#_heading=h.3znysh7)

[Module Three Milestone 2](#_heading=h.4d34og8)

[Ten Core Security Principles 2](#_heading=h.2s8eyo1)

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

[Coding Standard 1 4](#_heading=h.26in1rg)

[Coding Standard 2 5](#_heading=h.lnxbz9)

[Coding Standard 3 6](#_heading=h.35nkun2)

[Coding Standard 4 7](#_heading=h.1ksv4uv)

[Coding Standard 5 8](#_heading=h.2jxsxqh)

[Coding Standard 6 9](#_heading=h.z337ya)

[Coding Standard 7 10](#_heading=h.28h4qwu)

[Coding Standard 8 11](#_heading=h.nmf14n)

[Coding Standard 9 13](#_heading=h.37m2jsg)

[Coding Standard 10 14](#_heading=h.1mrcu09)

[Defense-in-Depth Illustration 15](#_heading=h.46r0co2)

[Project One 15](#_heading=h.2lwamvv)

[1.](#_heading=h.111kx3o) Revise the C/C++ Standards 15

[2.](#_heading=h.3l18frh) Risk Assessment 15

[3.](#_heading=h.206ipza) Automated Detection 15

[4.](#_heading=h.4k668n3) Automation 15

[5.](#_heading=h.2zbgiuw) Summary of Risk Assessments 16

[6.](#_heading=h.1egqt2p) Create Policies for Encryption and Triple A 16

[7.](#_heading=h.3ygebqi) Map the Principles 17

[Audit Controls and Management 18](#_heading=h.2dlolyb)

[Enforcement 18](#_heading=h.sqyw64)

[Exceptions Process 18](#_heading=h.3cqmetx)

[Distribution 19](#_heading=h.1rvwp1q)

[Policy Change Control 19](#_heading=h.4bvk7pj)

[Policy Version History 19](#_heading=h.2r0uhxc)

[Appendix A Lookups 19](#_heading=h.1664s55)

[Approved C/C++ Language Acronyms 19](#_heading=h.3q5sasy)

# 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 | When a user enters information into the software it will need to be validated. This ensures that the software is getting information that it is expecting. This can prevent crashes and bugs. It also prevents malicious actors from entering information that corrupt or grant unwanted access to a database or server. |
| 1. Heed Compiler Warnings | Compiler warnings are added to software because they help fix known issues. Many times these compiler warnings can prevent bugs from happening later on in development but can also help secure code. Typically if the compiler is giving a warning it is flagging a known issue which the developer should interpret as a security issue or bug that has already happened and can be prevented. |
| 1. Architect and Design for Security Policies | In order to protect the integrity of the software system developers should examine how the safeguards are being implemented in their software. This implementation of the software safeguards and enforcement of the security policies can help to protect the information being handled by the system. This will protect how data is used, stored, and retrieved. |
| 1. Keep It Simple | Some of the most basic solutions are the most effective. Keeping the design of the software simple also makes it easier to secure the software. If all the parts of the software system are easily understood then the development team will be able to employ much more effective security strategies. |
| 1. Default Deny | The default mode of the software should deny access. Only users that have been authenticated and identified should have the ability to access part of the system or software. |
| 1. Adhere to the Principle of Least Privilege | One part of the program may not require information from other parts. It is important to limit the access to information to only what is needed to accomplish the task of that portion of the software. This will reduce the risk that hackers can gain access to critical parts of the system from an unexpected source. |
| 1. Sanitize Data Sent to Other Systems | It is important to specify the data that is being sent from one system to another. If a function is expecting an integer and it receives a string then errors can occur. It is important for the system to check the data that is being passed to ensure that it meets the expectations of the parts of the system that will be consuming the data. |
| 1. Practice Defense in Depth | By layering the defense strategies the team can increase the efficacy of the different security strategies. This will help the team detect attempted attacks and also makes it much harder for malicious actors to gain access to critical parts of the software system. |
| 1. Use Effective Quality Assurance Techniques | Quality assurance ensures that the software being produced meets the standards of the company. These techniques should be incorporated into the development process to ensure that the final product is secure and meets the needs of the stakeholders. |
| 1. Adopt a Secure Coding Standard | Unsafe coding practices can result in the loss of information. This can also damage the reputation of the company. It is important for all team members to adopt secure coding standards as part of their normal repertoire. |

## 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** | DCL50-CPP | Do not define a C-style variadic function |

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

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

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

|  |
| --- |
| **Principles(s):** 1. Validate Input Data - giving invalid data can have unexpected results this could result in a crash in best case and unknown error later on in development which is difficult to track down.  Sanitize Data Sent to Other Systems - It may seem unlikely to give this function incorrect data but it would be very likely to pass incorrect data to this function. When sending data to the function it is important to understand what the function is expecting. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | function-ellipsis | Fully checked |
| Clang | 3.9 | cert-dcl50-cpp | Checked by clang-tidy |
| Parasoft c/C++test | 2020.2 | CERT\_CPP-DCL50-a | Functions shall not be defined with a variable number of arguments. |
| RuleChecker | 20.10 | function-ellipsis | Fully Checked |

### Coding Standard 2

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Value** | EXP58-CPP | Pass an object of the correct type to va\_start |

|  |
| --- |
| **Noncompliant Code** |
| In this noncompliant code example, the object passed to va\_start() will undergo a default argument promotion, which results in undefined behavior. |
| #include <cstdarg>    extern "C" void f(**float** a, ...) {  **va\_list** list;  **va\_start**(list, a);  // ...  **va\_end**(list);  } |

|  |
| --- |
| **Compliant Code** |
| In this compliant solution, f() accepts a double instead of a float. |
| #include <cstdarg>    extern "C" void f(**double** a, ...) {  **va\_list** list;  **va\_start**(list, a);  // ...  **va\_end**(list);  } |

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

|  |
| --- |
| **Principles(s):** 1. Validate Input Data - giving invalid data can have unexpected results this could result in a crash in best case and unknown error later on in development which is difficult to track down.  Sanitize Data Sent to Other Systems - It may seem unlikely to give this function incorrect data but it would be very likely to pass incorrect data to this function. When sending data to the function it is important to understand what the function is expecting.  2. Heed the Compiler Warning - This code may compile with errors or warnings. It is important to fix this error early in development. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| EXP58-CPP | Unlikely | Medium | P4 | L3 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Clang | 3.9 | -Wvarargs | Does not catch the violation in the third noncompliant code example (it is conditionally supported by Clang) |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-EXP58-a | Use macros for variable arguments correctly |
| Polyspace Bug Finder | R2020a | CERT C++ EXP58-CPP | Checks for incorrect data types for second argument of va\_start (rule fully covered) |

### Coding Standard 3

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **String Correctness** | STR53-CPP | Range check element access |

|  |
| --- |
| **Noncompliant Code** |
| In this noncompliant code example, the value returned by the call to get\_index() may be greater than the number of elements stored in the string, resulting in undefined behavior. |
| #include <string>    extern std::**size\_t** get\_index();    void f() {  std::string s("01234567");  s[get\_index()] = '1';  } |

|  |
| --- |
| **Compliant Code** |
| This compliant solution uses the std::basic\_string::at() function, which behaves in a similar fashion to the index operator[] but throws a std::out\_of\_range exception if pos >= size(). |
| #include <stdexcept>  #include <string>  extern std::**size\_t** get\_index();    void f() {  std::string s("01234567");  try {  s.at(get\_index()) = '1';  } catch (std::out\_of\_range &) {  // Handle error  }  } |

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

|  |
| --- |
| **Principles(s):** 1. Validate Input Data - giving invalid data can have unexpected results this could result in a crash in best case and unknown error later on in development which is difficult to track down.  Default Deny, by default the function should fail if the input is not what is expected. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | assert\_failure |  |
| CodeSonar | 6.0p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TBA  LANG.MEM.TO  LANG.MEM.TU | Buffer overrun  Buffer underrun  Tainted buffer access  Type overrun  Type underrun |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-STR53-a | Guarantee that container indices are within the valid range |
| Polyspace Bug Finder | R2020a | CERT C++: STR-CPP | Checks for:  Array access out of bounds  Array access with tainted index  Pointer dereference with tainted offset  Rule partially covered. |

### Coding Standard 4

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **SQL Injection** | FIO30-C | Exclude user input from format strings |

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

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

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

|  |
| --- |
| **Principles(s):** Keep It Simple, this code actually simplifies the process and reduces steps while increases code security. It does not allow a user input into the system and simply bypasses it and sends a message to the user without their code being processed. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | assert\_failure | Supported via stubbing/taint analysis |
| CodeSonar | 6.0p0 | IO.INJ.FMT  MISC.FMT | Format string injection |
| Parasoft C/C++test | 2020.2 | CERT\_C-FIO30-a  CERT\_C-FIO30-b  CERT\_C-FIO30-c | Avoid calling functions printf/wprintf with only one argument other than string constant  Avoid using functions fprintf/fwprintf with only two parameters, when second parameter is a variable  Never use unfiltered data from an untrusted user as the format parameter |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2020a | CERT C: Rule FIO30-C | Checks for tainted string format (rule partially covered) |

### Coding Standard 5

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Memory Protection** | MEM31-C | Free dynamically allocated memory when no longer needed |

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

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

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

|  |
| --- |
| **Principles(s):** Architect and Design for Security Policies these floating pointers may result in a security risk to the system. By freeing up the memory allocation the function is ensuring that data can only be accessed with the function is running. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 |  | Supported, but no explicit checker |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++test | 2020.2 | CERT\_C-MEM31-a | Ensure resources are freed |
| [Polyspace Bug Finder](https://wiki.sei.cmu.edu/confluence/display/c/Polyspace+Bug+Finder) | R2020a | CERT C: Rule MEM31-C | Checks for memory leak (rule fully covered) |

### Coding Standard 6

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Assertions** | DCL56-CPP | Avoid cycles during initialization of static objects |

|  |
| --- |
| **Noncompliant Code** |
| This noncompliant example attempts to implement an efficient factorial function using caching. Because the initialization of the static local array cache involves recursion, the behavior of the function is undefined, even though the recursion is not infinite. |
| #include <stdexcept>    **int** fact(**int** i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    static const **int** cache[] = {  fact(0), fact(1), fact(2), fact(3), fact(4), fact(5),  fact(6), fact(7), fact(8), fact(9), fact(10), fact(11),  fact(12), fact(13), fact(14), fact(15), fact(16)  };    if (i < (sizeof(cache) / sizeof(**int**))) {  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

|  |
| --- |
| **Compliant Code** |
| This compliant solution avoids initializing the static local array cache and instead relies on zero-initialization to determine whether each member of the array has been assigned a value yet and, if not, recursively computes its value. It then returns the cached value when possible or computes the value as needed. |
| #include <stdexcept>    **int** fact(**int** i) noexcept(false) {  if (i < 0) {  // Negative factorials are undefined.  throw std::domain\_error("i must be >= 0");  }    // Use the lazy-initialized cache.  static **int** cache[17];  if (i < (sizeof(cache) / sizeof(**int**))) {  if (0 == cache[i]) {  cache[i] = i > 0 ? i \* fact(i - 1) : 1;  }  return cache[i];  }    return i > 0 ? i \* fact(i - 1) : 1;  } |

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

|  |
| --- |
| **Principles(s):**Keep It Simple, the compliant code is less verbose and accomplishes the same task without addition code and memory allocation. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| LDRA tool suite | 9.7.1 | 6 D | Enhanced Enforcement |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-DCL56-a | Avoid initialization order problems across translation units by replacing non-local static objects with local static objects |

### Coding Standard 7

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Exceptions** | ERR50-CPP | Do not abruptly terminate the program |

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

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

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

|  |
| --- |
| **Principles(s):** This is a catch all that will hide the error. Heed Compiler Warnings - the compiler should be to detect most issues before the function crashes. By listening to the compiler warnings and handling errors it will be easy to make bug free software. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | stdlib-use | Partially checked |
| CodeSonar | 6.0p0 | BADFUNC.ABORT  BADFUNC.EXIT | Use of abort  Use of exit |
| SonarQube C/C++ Plugin | 4.10 | S990 |  |
| PRQA QA-C++ | 4.4 | 5014 |  |

### Coding Standard 8

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| Containers | CTR50-CPP | Guarantee that container indices and iterators are within the valid range |

|  |
| --- |
| **Noncompliant Code** |
| This noncompliant code example shows a function, insert\_in\_table(), that has two int parameters, pos and value, both of which can be influenced by data originating from untrusted sources. The function performs a range check to ensure that pos does not exceed the upper bound of the array, specified by tableSize, but fails to check the lower bound. Because pos is declared as a (signed) int, this parameter can assume a negative value, resulting in a write outside the bounds of the memory referenced by table. |
| #include <cstddef>    void insert\_in\_table(**int** \*table, std::**size\_t** tableSize, **int** pos, **int** value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

|  |
| --- |
| **Compliant Code** |
| In this compliant solution, the parameter pos is declared as size\_t, which prevents the passing of negative arguments. |
| #include <cstddef>    void insert\_in\_table(**int** \*table, std::**size\_t** tableSize, std::**size\_t** pos, **int** value) {  if (pos >= tableSize) {  // Handle error  return;  }  table[pos] = value;  } |

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

|  |
| --- |
| **Principles(s):** Validate Input Data - giving invalid data can have unexpected results this could result in a crash in best case and unknown error later on in development which is difficult to track down.  Sanitize Data Sent to Other Systems - It may seem unlikely to give this function incorrect data but it would be very likely to pass incorrect data to this function. When sending data to the function it is important to understand what the function is expecting. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | overflow\_upon\_dereference |  |
| CodeSonar | 6.0p0 | LANG.MEM.BO  LANG.MEM.BU  LANG.MEM.TO  LANG.MEM.TU  LANG.MEM.TBA  LANG.STRUCT.PBB  LANG.STRUCT.PPE | Buffer overrun  Buffer underrun  Type overrun  Type underrun  Tainted buffer access  Pointer before beginning of object  Pointer past end of object |
| LDRA tool suite | 9.7.1 | 45 D, 47 S, 476 S, 489 S, 64 X, 66 X, 68 X, 69 X, 70 X, 71 X, 79 X | Partially implemented |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-CTR50-a | Guarantee that container indices are within the valid range |

### Coding Standard 9

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| Integers | INT50-CPP | Do not cast to an out-of-range enumeration value |

|  |
| --- |
| **Noncompliant Code** |
| This noncompliant code example attempts to check whether a given value is within the range of acceptable enumeration values. However, it is doing so after casting to the enumeration type, which may not be able to represent the given integer value. On a two's complement system, the valid range of values that can be represented by EnumType are [0..3], so if a value outside of that range were passed to f(), the cast to EnumType would result in an unspecified value, and using that value within the if statement results in unspecified behavior. |
| enum EnumType {  First,  Second,  Third  };    void f(**int** intVar) {  EnumType enumVar = static\_cast<EnumType>(intVar);    if (enumVar < First || enumVar > Third) {  // Handle error  }  } |

|  |
| --- |
| **Compliant Code** |
| This compliant solution checks that the value can be represented by the enumeration type before performing the conversion to guarantee the conversion does not result in an unspecified value. It does this by restricting the converted value to one for which there is a specific enumerator value. |
| enum EnumType {  First,  Second,  Third  };    void f(**int** intVar) {  if (intVar < First || intVar > Third) {  // Handle error  }  EnumType enumVar = static\_cast<EnumType>(intVar);  } |

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

|  |
| --- |
| **Principles(s):** This follows the principle of validating data taken in from the user. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-INT50 |  |
| Parasoft C/C++test | 2020.2 | CERT\_CPP-INT50-a | An expression with enum underlying type |
| PVS-Studio | 7.07 | V1016 |  |
| PRQA QA-C++ | 4.4 | 3013 |  |

### Coding Standard 10

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| Object Oriented Programming | OOP50-CPP | Do not invoke virtual functions from constructors or destructors |

|  |
| --- |
| **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):** [Name the principle and explain how it maps to this standard.] |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 20.10 | virtual-call-in-constructor  invalid\_function\_pointer | Fully checked |
| Axivion Bauhaus Suite | 6.9.0 | CertC++-OOP50 |  |
| Clang | 3.9 | clang-analyzer-alpha.cplusplus.VirtualCall | Checked by clang-tidy |
| LDRA tool suite | 9.7.1 | 467 S, 92 D | Fully implemented |

## 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 | High | 2 |
| DCL50-CPP | High | Probable | Medium | P12 | L1 |
| EXP58-CPP | Medium | Unlikely | Medium | P4 | L3 |
| STR53-CPP | High | Unlikely | Medium | P6 | L2 |
| FIO30-C | High | Likely | Medium | P18 | L1 |
| MEM31-C | Medium | Probable | Medium | P8 | L2 |
| DCL56-CPP | Low | Unlikely | Medium | P2 | L3 |
| ERR50-CPP | Low | Probable | Medium | P4 | L3 |
| CTR50-CPP | High | Likely | High | P9 | L2 |
| INT50-CPP | Medium | Unlikely | Medium | P4 | L3 |
| OOP50-CPP | Low | Unlikely | Medium | P2 | L3 |

## Create Policies for Encryption and Triple A

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

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

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

|  |  |
| --- | --- |
| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| Encryption in rest | Ensures that data stored on disk is encrypted which makes it more difficult for a hacker to access the data in the event of theft. |
| Encryption at flight | This is the process of encrypting data while it is being transmitted. |
| Encryption in use | A method for using data while it is in it’s encrypted state so that decryption is not necessary for an application to run. This can be done by trusting the hardware or trusting the end application. |

|  |  |
| --- | --- |
| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| Authentication | The process is verifying the identity of the user with user logins via password or two-factor authentication. |
| Authorization | The process of determining the users level of access. Specify what an authenticated user is allowed to change on a database and how often they can access the database. Addition of new users should be a limited privilege granted only to administrative users. |
| Accounting | The process of keeping records of all user actions in logs. This should include all of the files accessed by the user. |

**\***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 |  |
| 2.0 | 04/11/2021 | Revised Template | Ryan O’Connor | [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 |