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



Green Pace Secure Development Policy

Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

# Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

# Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

# Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

# Module Three Milestone

## Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate Input Data refers to verifying all input received from untrusted data sources. By assuming that all data received from external sources may be of malicious nature having a proper input validation method can help block access to any software vulnerabilities. |
| 1. Heed Compiler Warnings | When creating code you should compile code using the highest warning level available for your compiler, next you must actually pay attention to the warnings and bugs found by the compiler. The compiler will often find bugs that are hard to find and detect during testing. Paying attention to these warnings can help decrease the risk of allowing potential security risks within your code being pushed to Production. |
| 1. Architect and Design for Security Policies | When starting a new project, understanding the security policies and the risks that the new code could introduce to the environment, it is important to understand how the code will interact with both the software and hardware, for example, understanding how the users will be accessing and using the code being created can help during the development to ensure that it is protecting the environment from any potential security vulnerabilities. |
| 1. Keep It Simple | Keep It Simple is always the best practice to follow, the more complex a process gets, the more opportunities there are for items to be missed, and more time and resources are needed to support the project. Keeping items as simple as possible narrows the window for any possible security vulnerabilities. |
| 1. Default Deny | When granting access to your programs, allow access based on permission levels rather than exclusion. Understand how the program is going to be used by the end users and what levels of access each type of user may need and create permission levels based off of these process levels. |
| 1. Adhere to the Principle of Least Privilege | When completing your code, make processes work with the least amount of permission levels possible, when accessing with higher permission levels, create processes that require small amounts of execution time, the longer a user is accessing a program with elevated permission levels, the longer you expose the information to security vulnerabilities allowing for potential attacks. |
| 1. Sanitize Data Sent to Other Systems | Sanitizing data is eliminating unwanted characters from the input by removing, replacing, encoding or escaping the characters, by removing these items from the data it can help to prevent opening up holes for the hackers to gain access to your systems. |
| 1. Practice Defense in Depth | When looking at your security protocols, install layers of defense, if one item fails, then another layer should pick up and defend, by examining your processes and understanding where your vulnerabilities are, you can then create a strategic layered approach to your security measures. |
| 1. Use Effective Quality Assurance Techniques | Before pushing any code to Production, all new development should go through a thorough testing cycle to ensure that most bugs and defects are found before sending it out. This will help mitigate any security issues that may be open without vetting the code. Using compilers to point out any issues, development unit testing and peer review and a final quality assurance testing can help find as many bugs and vulnerabilities as possible. |
| 1. Adopt a Secure Coding Standard | Best practices and continuity in coding practices will help flush out any potential security issues, by always following the same practices and ensuring that these standards are upheld can help lessen the chances of any harmful attack on the systems and or software applications. |

## 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** | INT30-C | Ensure that unsigned integer operations do not wrap  Code needs to prevent integer wraps, without preventing this behavior buffer overflows could happen and this could also allow the execution of arbitrary code by an attacker.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensure+that+unsigned+integer+operations+do+not+wrap |

| **Noncompliant Code** |
| --- |
| Noncompliant Code can result in an unsigned integer wrap during the addition of the unsigned operands, the resulting value may be used to allocate insufficient memory for subsequent operation |
| void func(unsigned **int** ui\_a, unsigned **int** ui\_b) {    unsigned **int** usum = ui\_a + ui\_b;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Compliant code 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  Standard- Data Type  Verifies that the data has a proper input validation |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Integer-overflow | Fully checked |
| CodeSonar | 6.0p0 | **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 |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STR11-C | Do not specify the bound of a character array initialized with a string literal  Best practice is to not specify the bound of a string initialized with a string literal allowing the compiler to automatically allocate space for the string.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/STR11-C.+Do+not+specify+the+bound+of+a+character+array+initialized+with+a+string+literal |

| **Noncompliant Code** |
| --- |
| Noncompliant Code initializes an array of characters using a string literal that defines one character more than the array can hold |
| const **char** s[3] = "abc"; |

| **Compliant Code** |
| --- |
| Compliant code does not specify the bound of a character array in the array declaration, this allows the compiler to allocate sufficient storage to the enter sting literal, including the terminating null character. |
| const **char** s[] = "abc"; |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings  Standard: Data Value  Compiler will allocate sufficient space for the entire string literal |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| ÉCLAIR | 1.2 | CCS.STR36 | Fully implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_C-STR11-a | Do not specify the bound of a character array initialized with a string literal |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STR30-C | Do not attempt to modify string literals  By allowing modifications to this data, this can lead to unexpected program terminations and possibly denial of service attacks, modifying a string literal can result in an access violation since they are typically stored in read only memory.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/STR30-C.+Do+not+attempt+to+modify+string+literals |

| **Noncompliant Code** |
| --- |
| Noncompliant code the char pointer is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior |
| **char** \*str  = "string literal";  str[0] = 'S'; |

| **Compliant Code** |
| --- |
| An array initializer, a string literal specifies the initial values of characters in a array as well as the size of the array. The string stored in str can be modified safely |
| **char** str[] = "string literal";  str[0] = 'S'; |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings  Standard: String Correctness  Behavior is undefined if a program attempts to modify any portion of a string literal |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | String-literal-modification  Write-to-string-literal | Fully checked |
| Parasoft C/C++ test | 2020.2 | CERT\_C-STR30-a  CERT\_C-STR30-b | A string literal shall not be modified  Do not modify string literals |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STR02-C | Sanitize data passed to complex subsystems  Data entered should be sanitized of illegal characters to stop malicious attacks upon the software, by not checking for these items, this can easily open the doors for attackers to gain access to data.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems |

| **Noncompliant Code** |
| --- |
| Noncompliant code allows user to log onto system with elevated privileges, the call passes unsanitized data from the untrusted source as an argument to the login program |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")),    0); |

| **Compliant Code** |
| --- |
| The compliant code inserts the double dash argument before the call, the option causes getopt() to stop interpreting options in the argument list |
| (void) execl(LOGIN\_PROGRAM, "login",    "-p",    "-d", slavename,    "-h", host,    "-s", pam\_svc\_name,    "--",    (AuthenticatingUser != NULL ? AuthenticatingUser :  **getenv**("USER")), 0); |

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

|  |
| --- |
| **Principles(s):** Sanitize Data Sent to Other Systems  Standard: SQL Injection  Sanitize data to help prevent it from opening up holes for hackers to gain access |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 6.5 | TAINTED\_STRING | Fully implemented |
| Parasoft C/C++ test | 2020.2 | 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 |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | MEM30-C | Do not access freed memory  Writing to memory that has already been freed and lead to the execution of arbitrary code, it can possibly modify memory that has already been allocated and can lead to unexpected program termination.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/MEM30-C.+Do+not+access+freed+memory |

| **Noncompliant Code** |
| --- |
| Noncompliant Code p is freed from p-> next is executed so that p-> next reads memory that has already been freed |
| #include <stdlib.h>    struct node {  **int** value;    struct node \*next;  };    void free\_list(struct node \*head) {    for (struct node \*p = head; p != NULL; p = p->next) {  **free**(p);    }  } |

| **Compliant Code** |
| --- |
| Compliant code corrects error by storing a reference to p-> next in q before freeing p |
| #include <stdlib.h>    struct node {  **int** value;    struct node \*next;  };    void free\_list(struct node \*head) {    struct node \*q;    for (struct node \*p = head; p != NULL; p = q) {      q = p->next;  **free**(p);    }  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard  Standard: Memory Protection  Using the same coding process can help lessen the chance of an attack |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | dangling\_pointer\_use | Supported  Reports all accesses to freed allocated memory |
| Parasoft C/C++ test | 2020.2 | CERT\_C-MEM30-a | Do not use resources that have been freed |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | MSC11-C | Incorporate diagnostic tests using assertions  Assertion testing is used during debugging and turned off before code is deployed. This tool should be used to help find and eliminate software defects that may result in vulnerabilities  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/MSC11-C.+Incorporate+diagnostic+tests+using+assertions |

| **Noncompliant Code** |
| --- |
| Using the assert () macro to verify memory allocation succeeded would be incorrect, this could lead to an abrupt termination of the process. |
| **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** |
| --- |
| Compliant code removes the assert statement to correctly handle possibly 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):** Use Effective Quality Assurance Techniques  Standard: Assertions  Assertion testing helps find defects that may result in vulnerabilities |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | LANG.FUNCS.ASSERTS | Not enough assertions |
| Coverity | 20.17.07 | 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** | ERR56-CPP | Guarantee exception safety  The safest method for reporting errors in code is to through exceptions rather than error codes, all exceptions should be addressed prior to deployment to ensure that code is safe from vulnerabilities.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR56-CPP.+Guarantee+exception+safety |

| **Noncompliant Code** |
| --- |
| Noncompliant code deallocates array and assigns the element counter before allocating a new block of memory for the copy. |
| #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** |
| --- |
| Compliant code allocates new storage for the copy before changing the state of the object. |
| #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):** Use Effective Quality Assurance Techniques  Standard: Exceptions  All exceptions should be addressed to ensure safety from vulnerabilities |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| LDRA tool suite | 9.7.1 | 527 S, 56 D, 71 D | Partially implemented |
| Parasoft C/C++ test | 2020.2 | CERT\_CPP-ERR56-a | Ensure resources are freed |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Miscellaneous | MSC40-C | Do not violate constraints  Constraint violations encompass a broad category that can result in unexpected control flow and corrupted data  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/MSC40-C.+Do+not+violate+constraints |

| **Noncompliant Code** |
| --- |
| Noncompliant code refers to static variable with file scope and internal linkage |
| static **int** I = 12;  extern inline void func(**int** a) {  **int** b = a \* I;    /\* ... \*/  } |

| **Compliant Code** |
| --- |
| Compliant code removes the static qualifier, the variable I now has external linkage by default |
| **int** I = 12;  extern inline void func(**int** a) {  **int** b = a \* I;    /\* ... \*/  } |

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

|  |
| --- |
| **Principles(s):** Adopt a Secure Coding Standard  Standard: Miscellaneous  Following the same practices and ensuring that these standards are followed can help lessen the chances of attack |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Parasoft C/C++ test | 2020.2 | CERT\_C-MSC40-a | An inline definition of a function with external linkage shall not contain definitions and uses of static objects |
| Polyspace Bug Finder | R2020a | CERT C: Rule MSC40-C | Checks for inline constraint not respected  (rule partially covered) |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Output (FIO) | FIO42-C | Close files when they are no longer needed  Leaving files open may allow for an attack to occur, exhaust system resources and may increase the risk that data written into memory buffers may not be flushed in the event of an unexpected program exit.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/FIO42-C.+Close+files+when+they+are+no+longer+needed |

| **Noncompliant Code** |
| --- |
| Noncompliant code exists because the file opened is not closed before the func() returns |
| #include <stdio.h>    **int** func(const **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");    if (NULL == f) {      return -1;    }    /\* ... \*/    return 0;  } |

| **Compliant Code** |
| --- |
| Compliant code closes the file before returning to the caller |
| #include <stdio.h>    **int** func(const **char** \*filename) {  **FILE** \*f = **fopen**(filename, "r");    if (NULL == f) {      return -1;    }    /\* ... \*/    if (**fclose**(f) == EOF) {      return -1;    }    return 0;  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings  Standard: Input Output (FIO)  Compiler can find hard to find bugs and defects to help decrease the risk of vulnerabilities |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.0p0 | ALLOC.LEAK | Leak |
| Parasoft C/C++ test | 2020.2 | CERT\_C-FIO42-a | Ensure resources are freed |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Arrays | ARR32-C | Ensure size arguments for variable length arrays are in valid range  Code should properly specify the size of a variable length in order to prevent arbitrary code execution or stack exhaustion.  Confluence. (n.d.). Retrieved March 21, 2021, from https://wiki.sei.cmu.edu/confluence/display/c/ARR32-C.+Ensure+size+arguments+for+variable+length+arrays+are+in+a+valid+range |

| **Noncompliant Code** |
| --- |
| Noncompliant code does not declare an allowed size for the variable being entered |
| #include <stddef.h>    extern void do\_work(**int** \*array, **size\_t** size);    void func(**size\_t** size) {  **int** vla[size];    do\_work(vla, size);  } |

| **Compliant Code** |
| --- |
| Compliant code defines the max allowed size for a variable |
| #include <stdint.h>  #include <stdlib.h>    enum { MAX\_ARRAY = 1024 };  extern void do\_work(**int** \*array, **size\_t** size);    void func(**size\_t** size) {    if (0 == size || SIZE\_MAX / sizeof(**int**) < size) {      /\* Handle error \*/      return;    }    if (size < MAX\_ARRAY) {  **int** vla[size];      do\_work(vla, size);    } else {  **int** \*array = (**int** \*)**malloc**(size \* sizeof(**int**));      if (array == NULL) {        /\* Handle error \*/      }      do\_work(array, size);  **free**(array);    }  } |

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

|  |
| --- |
| **Principles(s):** Heed Compiler Warnings  Standard: Arrays  Variable length arrays are not supported by compilers |

**Threat Level**

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

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2017.07 | REVERSE\_NEGATIVE | Fully implemented |
| Parasoft C/C++ test | 2020.2 | CERT C-ARR32-a | Ensure the size of the variable length array is in valid range |

## Defense-in-Depth Illustration

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



# Project One

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

## Revise the C/C++ Standards

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

## Risk Assessment

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

## Automated Detection

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

## Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation processes can be implemented once you start the Build process of the project, by following unit testing during coding you have a chance to catch any vulnerabilities within the code, continuing the automation during the Testing phase along with manual testing processes can help increase the chances of finding any bugs or defects that could result in areas that may allow for hackers to gain access.

Once the project is pushed to production, having an automation process in place to continue testing the software to help catch any vulnerabilities that may have been introduced after the fact and help find any leaks that may have presented themselves and catch these issues before attackers have a chance to gain access to the applications and stopping the attacks from occurring.

## 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 |
| INT30-C | High | Likely | High | Medium | 2 |
| STR11-C | Low | Probable | Low | Medium | 2 |
| STR30-C | Low | Likely | Low | Medium | 2 |
| STR02-C | High | Likely | Medium | High | 1 |
| MEM30-C | High | Likely | Medium | High | 1 |
| MSC11-C | Low | Unlikely | High | Low | 3 |
| ERR56-CPP | High | Likely | High | Medium | 2 |
| MSC40-C | Low | Unlikely | Medium | Low | 3 |
| FIO42-C | Medium | Unlikely | Medium | Low | 3 |
| ARR32-C | High | Probable | High | Medium | 2 |

## Create Policies for Encryption and Triple A

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

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

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

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption in rest | This is designed to protect attackers from gaining access to unencrypted data by ensuring the data is encrypted when on disk. Encryption in rest provides defense-in-depth protection. |
| Encryption at flight | This process camouflages data traffic so it cannot be interpreted or altered. This process will protect data that is transferred over public networks such as the internet. |
| Encryption in use | Encryption in use should limit users to access data based on their roles, limiting system access to only those who need it. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Confirm that the right person is accessing the system by user login with passwords and/or security questions |
| Authorization | Give users the minimum access for their job functions by creating roles for each function that defines the level of access that each would require to complete the actions needed |
| Accounting | Monitor activities for security and compliance by using auditing and monitoring tools creating prompts or alerts when a person is trying to access data that they do not have the rights to access. |

**\***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.

|  |  |  |
| --- | --- | --- |
| **Principle** | **Standard** | **Justification** |
| Validate Input Data | Data Type | By having a proper input validation method can help block access to any software vulnerabilities |
| Heed Compiler Warnings | Data Value  String Correctness  Arrays  Input/Output FIO | Paying attention to compiler warnings can help decrease the risk of allowing potential security risks within code from being pushed to Production |
| Sanitize Data Sent to Other Systems | SQL Injection | By removing unwanted characters from the data can help to prevent opening up holes for hackers to gain access |
| Adopt a Secure Coding Standard | Memory Protection  Miscellaneous | By following same practices and ensuring that these standards are upheld can help lessen the chances of harmful attacks on the systems or software applications |
| Use Effective Quality Assurance Techniques | Assertions  Exceptions | Using compilers to point out any issues, development unit testing and peer review and a final quality assurance testing can help find as many bugs and vulnerabilities as possible |

**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 | 03/21/2021 | Module One | Michelle Fabry | [Insert text.] |
| 1.2 | 4/11/2021 | Project One | Michelle Fabry | [Insert text.] |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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