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



Green Pace Secure Development Policy

Contents

**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. |
| * ValidateInput Data | Validating user input is crucial for ensuring that all data received is safe and accounted for before processing and storing. This is a preventative measure to verify that the data matches the systems expectations and does not lead to unexpected behaviors, errors, and vulnerabilities. Common strategies used to prevent Sql injections, buffer overflows, and cross site scripting are data type checking, input sanitization, and parameterization. These methods of validating input data helps protect the integrity of the system and the security of the data held within it. |
| * Heed Compiler Warnings | As a developer, we must address compiler warnings as they illustrate potential problems or vulnerabilities within the code such as potential memory leaks, unhandled exceptions, and other potential security issues. Aiming to eliminate all compiler warnings and errors promptly should be prioritized as it leads to a more maintainable and secure code base. Compiler warnings should never be ignored except for the rare case in which the developer can confidently say that it does not introduce immediate additional risks or vulnerabilities. |
| * Architect and Design for Security Policies | Establishes and enforces security policies within the software design and architecture. Common security policies include role-based access control for establishing privileges for viewing and editing data, data encryption, and user authentication. |
| * Keep It Simple | Minimizing complexity in design and implementation can reduce the likelihood of a security flaw or vulnerability. Avoiding complexity in favor of minimalism and simplicity can make the code base easier to maintain and secure. This is because the code base will become easier to navigate and identify potential issues. Furthermore, sticking to thoroughly tested solutions over newer, complex ones reduces the chance of introducing unreliability and unforeseen vulnerabilities. |
| * Default Deny | Core security principle that sets default access to systems or networks to denied unless granted access/whitelisted. Explicitly setting access to denied by default prevents unforeseen access from being granted to malicious users minimizing the risk of unauthorized users gaining access to systems that may contain private information. A system that does not explicitly deny unauthorized users may implicitly grant them access. Privileges and authorization should be regularly monitored and updated to accurately safeguard sensitive information and control over the system. |
| * Adhere to the Principle of Least Privilege | Ensuring that the absolute minimum permission is granted to perform necessary functions is crucial for reducing the potential impact of a malicious attack. Reducing access while maintaining necessary function can also assist in identifying misuse of privilege as the number of individuals with higher level privilege is smaller. It is also beneficial for identifying those with lower-level access who suddenly attempt to give themselves access to higher-level functions or data raising a red flag that the user may be attempting to bypass their restrictions or have had their access compromised. |
| * Sanitize Data Sent to Other Systems | Sanitizing data sent to other systems not only upholds the integrity of the data being transmitted but also safeguards the other system from receiving data that has been corrupted or manipulated. Common sanitization strategies include data masking, using standardized formats, and data encryption. Cleaning and transforming the data being transmitted from one system to another reduces the points of failure to which data breaches, data corruption, and injection attacks could occur. Overall data sanitization will prevent data from being stolen and interpreted during transmission and prevent the receiving system from receiving data that has been infected with malicious code. |
| * Practice Defense in Depth | The security strategy defense in depth establishes a wide variety of overlapping security measures to protect systems and their data. The benefit of implementing this strategy is that if one layer is compromised other layers will remain in place to provide protection and mitigate the impact of the attack. Protecting against multiple points of failure reduces the impact of vulnerabilities and increases the rate at which exploits can be fixed. |
| * Use Effective Quality Assurance Techniques | Utilizing effective quality assurance techniques significantly reduces the number of vulnerabilities that make their way into production. Thorough testing of software alerts developers of improper handling of errors and data such as buffer overflows, invalid input data, and data integrity issues. Common effective quality assurance techniques include penetration testing, failure testing, code reviews, and performance testing. These techniques may be employed during or after development to establish more secure systems. |
| * Adopt a Secure Coding Standard | Secure coding standards establish strict guidelines and rules for reducing vulnerabilities and improving overall code quality. Employing best coding practices early on can proactively prevent vulnerabilities by identifying issues as they occur and establish a standard for high quality code. |

**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] | Avoid implicit data type conversion to prevent unexpected behavior |

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code demonstrates data loss by converting a larger data type to a smaller one without explicitly converting it. This can lead to unexpected data loss. |
| int main(){  double decimal = 99.87672738292848732;  int number = decimal;  return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code demonstrates explicit data type conversion with cast ensuring code is more maintainable and reduces the risk of unexpected data loss. |
| int main(){  double decimal = 99.87654;  int number = static\_cast<int>(decimal+1);  return 0;  } |

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

|  |
| --- |
| **Principles(s):** Heed compiler warnings – Many compilers will throw out warnings much like MSVC in visual studios detailing a possible loss of data when converting a bigger data type value to a smaller one. Heeding this warning may ensure the developer is aware of this issue and may even explicitly use cast to convert the data.  Use effective quality assurance techniques – Adopting effective quality assurance techniques may result in discovering data integrity issues and buffer overflows when converting valid input data to other data types. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Medium | Likely | Medium | Medium | L3 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 24.04 | -- | Supported via MISRA C:2012 Rules 10.1, 10.3, 10.4, 10.6, and 10.7 |
| Coverity Analysis | 2020.03 | MISRA\_CAST checker | Detects implicit casting |
| Clang static analyzer | 1.2.2.5 | Alpha.core.Conversion | Warns loss of sign/precision in implicit conversions |
| SonarQube | 2025.1 | C++ static code analysis – Implicit casts should not lower precision | Narrowing conversion is detected and recommendations include utilizing a type that does not result in data loss or explicit casting |

**Coding Standard 2**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Data Value** | [STD-002-CPP] | Do not reuse variable names in sub scopes |

[**https://wiki.sei.cmu.edu/confluence/display/c/DCL01-C.+Do+not+reuse+variable+names+in+subscopes**](https://wiki.sei.cmu.edu/confluence/display/c/DCL01-C.+Do+not+reuse+variable+names+in+subscopes)

|  |
| --- |
| **Noncompliant Code** |
| The non-compliant code attempts a for loop initializing an integer i as value 0 and checks each iteration if the value in i is less than 10. Once the first iteration begins a string is declared with name i and is equal to "something" leading the code to not compile due to 2 variables having the same name in scope. |
| void main(){  for(int i = 0; i < 10; i++){  std::string i = "Something";  }  } |

|  |
| --- |
| **Compliant Code** |
| The compliant code successfully compiles as there are no variables with the same name in scope. |
| void main(){  for(int i = 0; i<10;i++){  std::string name = "Name";  std::cout << name << endl;  }  } |

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

|  |
| --- |
| **Principles(s):** Keep it simple – Using proper variable and function naming conventions that are both accurate and simple may significantly reduce vulnerabilities that can occur from overly complex variable name usage. Keeping variable names simple and directly correlated with the data they are storing may offer better readability and maintainability as the reuse of vague variable names in sub scopes becomes reduced.  Adopt a secure coding standard – Embedding secure coding standards into development early on can significantly improve the quality of the code written inherently reducing vulnerabilities from occurring like overwriting data by reusing variable names in sub scopes. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 24.04 | -- | Supported indirectly via MISRA C:2012 Rule 5.3. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-DCL01 | Enforces the Cert C guideline warning against the reuse of variable names in sub scopes |
| CodeSonar | 8.3p0 | LANG.ID.ND.NEST | Non-distinct identifiers: nested scope |
| ECLAIR | 1.2 | CC2.DCL01 | Fully implemented |

**Coding Standard 3**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **String Correctness** | [STD-003-CPP] | Always allocate enough space for string data and null terminator to prevent unexpected behavior |

[**https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator**](https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator)

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code expects user input that is unbounded. If a user were to type more than 19 characters buffer overflow may occur leading to vulnerabilities in the program. |
| Int main(){  char input[20];  std::cin >> input;  Return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code demonstrates use of string data type to dynamically allocate memory for storing string characters and null terminator. |
| int main(){  std::string input;  std::cout << "Enter your name: " << std"::endl;  std::cin >> input;  std::cout << "You entered " << input << std::endl;  return 0;  } |

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

|  |
| --- |
| **Principles(s):** validate input data – Validating that the data type utilized for storing string input data and the null terminator has enough space to store the data can prevent unexpected behavior like buffer overflows which can cause memory corruption, overwriting, and leakage.  Adopt a secure coding standard – Adopting a secure coding standard can ensure that buffers for strings are appropriately sized to prevent security vulnerabilities such as buffer overflows from occurring.  Sanitize data sent to other systems – When sending string data to other systems we want to ensure that not only is the data being allocated enough space on the current system but the data being sent is sanitized to fit the space allocated in the receiving end aswell. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Likely | Medium | High | L5 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 24.04 | -- | Supported  Astrée reports all buffer overflows resulting from copying data to a buffer that is not large enough to hold that data. |
| Axivion Bauhaus Suite | 7.2.0 | CertC-STR31 | Detects calls to unsafe string function that may cause buffer overflow Detects potential buffer overruns, including those caused by unsafe usage of fscanf() |
| CodeSonar | 8.3p0 | 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 |
| 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] | Prevent SQL injections |

[**https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection**](https://wiki.sei.cmu.edu/confluence/display/java/IDS00-J.+Prevent+SQL+injection)

[**https://medium.com/@ajay.monga73/parameterized-queries-c-guide-how-to-prevent-sql-injection-with-parameterized-queries-94b8105cacbd**](https://medium.com/@ajay.monga73/parameterized-queries-c-guide-how-to-prevent-sql-injection-with-parameterized-queries-94b8105cacbd)

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code example illustrates sql query directly concatenating the users input allowing for sql injection to occur like 1=1 vulnerability. |
| std::string username, password;  std::cout << "Enter username: " << std::endl;  std::cin >> username;  std::cout << "Enter password: " << std::endl;  std::cin >> pwd;  std::string query = "SELECT \* FROM db\_users WHERE username = '" + username + '" AND password = '" + password + "';"; |

|  |
| --- |
| **Compliant Code** |
| Compliant code example illustrates sql query using a standard parameterized query statement preventing malicious tampering of sql query statement like sql injection. User input is also being converted into a string to prevent 1=1 vulnerability. |
| std::string user = "Username and 1=1"  std::string pwd = "Password"  std::string query = "SELECT \* FROM db\_users WHERE username = ? AND password = ?";  sql::PreparedStatement\* stmt = conn->prepareStatement(query);  stmt->setString(1, user);  stmt->setString(2, pwd);  stmt->executeQuery(); |

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

|  |
| --- |
| **Principles(s):** Validate input data – Validating that the data provided does not contain harmful SQL statements such as 1=1 can mitigate security vulnerabilities like sql injections from occurring.  Sanitize data sent to other systems – By sanitizing input data being passed into an sql query of any special characters or keywords that are common in sql injections we are proactively restricting database calls to only allow valid queries.  Ahere to the principle of least privilege – By adhering to the principle of least privilege we can restrict users from accessing data that is beyond the scope of what is needed to perform their roles functions not only assisting in mitigating sql injections but preventing further damage to the integrity of the system if injection does occur.  Default deny – By adhering to the security principle of default deny all users who are not explicitly granted the ability to read, write, and update the sql database should be denied as limiting the number of possible bad actors is crucial for maintaining the integrity of the database. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Likely | High | High | L5 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 8.3p0 | IO.INJ.SQL | Detects unsafe input handling directly used in sql statements without sanetization |
| Fortify | 21.1 | SQL Injection Check | SQL injection prevention by detection of patterns such as lack of input validation or sanitization and direct concatenation in sql queries. |
| Parasoft C/C++ test | 2024.2 | CERT\_C-STR02-c | Protects against SQL injection |
| Clang | 1.0 | Optin.taint.TaintPropogation:config | Implemented |

**Coding Standard 5**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Memory Protection** | [STD-005-CPP] | Do not access freed memory |

[**https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/MEM50-CPP.+Do+not+access+freed+memory)

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code illustrates a pointer pointing to an integer with value 99 which has been dynamically allocated memory. The memory is then dereferenced and printed. The memory is then freed with the pointer still pointing to the memory location and is trying to be accessed which can cause undefined behavior. |
| int memoryUnprotection(){  int\* pointer = nullptr;  pointer = new int(99);  std::cout << \*pointer << std::endl  delete pointer;  std::cout << \*pointer << std::endl  Return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code illustrates a pointer being established that points to a newly allocated integer in memory. The integer is then dereferenced and printed. The memory is then freed and the dangling pointer is set to nullptr to avoid being accidently used which can later cause undefined behavior. |
| void memoryProtection(){  int\* pointer = nullptr;  pointer = new int(99);  std::cout << \*pointer << std::endl;  delete pointer;  pointer = nullptr;  } |

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

|  |
| --- |
| **Principles(s):** Adopt a secure coding standard – Establishing guidelines for secure coding will warn the developer against accessing freed memory to prevent vulnerabilities like the use after free vulnerability that can lead to unexpected behavior occurring like crashes.  Keep it simple – Keeping it simple by utilizing smart pointers over raw pointers will mitigate vulnerabilities by avoiding manual memory management preventing dangling pointers.  Heed compiler warnings – Compilers will often warn the developer when a pointer is dangling to inform them that further attempts to access freed memory will result in unexpected behavior. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Probable | High | High | L5 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 22.10 | Dangling\_pointer\_use | Identified instances in which memory is freed but pointers continue to reference it. |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-MEM50 | Enforces CERTC++MEM50 guideline preventing undefined behavior from dangling pointers |
| Clang | 3.9 | Clang-analyzer-cplusplus.NewDelete  Clang-analyzer-alpha.security.ArrayBoundV2 | Checked by clang-tidy, but does not catch all violations of this rule. |
| CodeSonar | 8.3p0 | ALLOC.UAF | Use after free |

**Coding Standard 6**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Assertions** | [STD-006-CPP] | Utilize assertions for debugging of programs not verifying the absence of runtime errors |

[**https://wiki.sei.cmu.edu/confluence/display/java/MSC60-J.+Do+not+use+assertions+to+verify+the+absence+of+runtime+errors**](https://wiki.sei.cmu.edu/confluence/display/java/MSC60-J.+Do+not+use+assertions+to+verify+the+absence+of+runtime+errors)

|  |
| --- |
| **Noncompliant Code** |
| Assertion is utilized for runtime error instead of debugging leading to no protection against cases where 0 could be the divisor. Instead use logical statements and exceptions to validate. |
| double divide(double dividend, double divisor){  assert(divisor!=0 && "divisor must not be 0");  return dividend/divisor;  }  int main(){  double divisor = 5.0;  double dividend = 10.0;  double result = divide(dividend,divisor);  return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code uses logical operations and exceptions to detect and prevent runtime errors rather than relying on assertions which should be used in debugging. |
| double divide(double dividend, double divisor){  if(divisor == 0){  throw std::invalid\_argument("Divisor can not be equal to 0!");  }  return dividend/divisor;  }  int main(){  try{  double divisor = 5.0;  double dividend = 10.0;  double result = divide(dividend, divisor);  } catch(const std::invalid\_argument& e){  std::cerr << "Invalid argument: " << e.what() << std::endl;  }  return 0;  } |

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

|  |
| --- |
| **Principles(s):** Use effective quality assurance techniques – Following effective quality assurance techniques will deter developers from relying on assertions for verifying absence of runtime errors as they will become informed on how assertions are disabled at runtime and how there are proper testing strategies for quality assurance that should be used like unit testing.  Practice Defense In Depth – Defense in depth strategy may entail utilizing assertions for adding an extra layer of debugging but will also include defense strategies such as unit testing and fuzzy testing for runtime validations. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| Low | Probable | Low | Low | L1 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Cppcheck | 8.3p0 | CheckAssert  AssertWithSideEffects  AssignmentInAssert | Detects when an assert contains operations that contain sideeffects that may be required in release. |
| Parasoft C/C++ test | 2024.2 | CERT\_C-PRE31-b  CERT\_C-PRE31-c  Cert\_c-pre31-D | Assertions should not contain assignments, increment, or decrement operators Assertions should not contain function calls nor function-like macro calls Avoid side effects in arguments to unsafe macros |
| Coverity | 2017.07 | ASSERT\_SIDE\_EFFECTS | Partially implemented detects when an assertion contains an operation/function call that may have side effects |
| Axivion Bauhaus Suite | 7.2.0 | CERT-PRE31 | Warns to avoid side effects in assert marcos enforcing sei cert c rule. |

**Coding Standard 7**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Exceptions** | [STD-007-CPP] | Properly handle thrown exceptions |

[**https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions**](https://wiki.sei.cmu.edu/confluence/display/cplusplus/ERR51-CPP.+Handle+all+exceptions)

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code illustrates an exception thrown by valueChecker function but is not handled in main which can lead to unexpected problems like crashing the program. |
| void valueChecker(int value){  if(value != 5){  throw std::invalid\_argument("Value must be equal to 5");  }  std::cout << "Value is equal to " << value << std::endl;  }  int main(){  int value = 6;  valueChecker(value);  return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code utilizes a try catch block attempting to check if the number variable's value is equal to 5 and throws an invalid argument exception when it is not equal. The catch blocks also catch any standard exceptions, and a general catch block is placed to ensure any other exceptions are handled. |
| void valueChecker(int value) {  if(value != 5){  throw std::invalid\_argument("Value must be equal to 5");  }  std::cout << "Value is equal to " << value << std::endl;  }  int main() {  try {  int value = 6;  valueChecker(value);  }catch(const std::invalid\_argument& e) {  std::cerr << "Invalid argument: " << e.what() << std::endl;  } catch(const std::exception& e){  std::cerr << "Error has occured: " << e.what() << std::endl;  } catch(...){  std::cerr << "Unknown error occurred" << std::endl;  }  return 0;  } |

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

|  |
| --- |
| **Principles(s):** Use effective quality assurance techniques – Quality assurance testing often includes exceptional error handling to ensure errors are caught and handled instead of causing undefined behavior like crashes and data corruption.  Practice defense in depth – Catching and handling a vast number of runtime errors is an important additional layer of defense against errors that may lead to sensitive data being exposed or unexpected crashes. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Probable | Medium | High | L4 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 22.10 | Main-function-catch-all  Early-catch-all | Partially checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC++-ERR51 | Enforces CertC++-ERR51 rule of handling all exceptions. |
| CodeSonar | 8.3p0 | LANG.STRUCT.UCTCH | Unreachable Catch |
| Helix QAC | 2024.4 | C++4035, C++4036, C++4037 | -- |

**Coding Standard 8**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Memory**  **Safety** | [STD-008-CPP] | Do not read uninitialized memory |

[**https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory**](https://wiki.sei.cmu.edu/confluence/display/c/EXP33-C.+Do+not+read+uninitialized+memory)

|  |
| --- |
| **Noncompliant Code** |
| The non-compliant code attempts to print the value stored in the int variable uninitializedInteger but the variable has not been initialized leading to undefined behavior. |
| void uninitialized(){  int uninitializedInteger;  std::cout << uninitializedIntege;  } |

|  |
| --- |
| **Compliant Code** |
| The compliant code successfully prints the value stored in the int variable initializedInteger as it has been initialized as value 2 before attempting to print. |
| void initialized(){  int initializedInteger = 2;  std::cout << initializedInteger;  } |

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

|  |
| --- |
| **Principles(s):** Heed compiler warnings – The compiler will often provide warnings about uninitialized variables helping you to identify and prevent unexpected behaviors and security vulnerabilities when attempting to read uninitialized memory.  Architect and design for security policies – Enforces that design must adhere to coding standards and security policies which ensures that all variables are initialized before use to mitigate possible cases of memory being read and not initialized. |

**Threat Level**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Probable | Medium | High | L5 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Astrée | 24.04 | Uninitialized-local-read  Uninitialized-variable-use | Fully checked |
| Axivion Bauhaus Suite | 7.2.0 | CertC-EXP33 | Enforces CERTC-EXP33 rule preventing memory from being read and uninitialized |
| CodeSonar | 8.3p0 | LANG.MEM.UVAR | Uninitialized variable |
| Coverity | 2017.07 | UNINIT | Implemented |

**Coding Standard 9**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Naming convention** | [STD-009-CPP] | Use meaningful and descriptive naming conventions |

|  |
| --- |
| **Noncompliant Code** |
| Noncompliant code illustrates no meaningful and descriptive naming conventions for variables leading to weak readability and maintainability as variables will be unrecognizable. |
| int main(){  int integer = 10;  std::string x = "My age is currently: ";  std::string birthday = x+std::to\_string(x)  return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code demonstrates the use of meaningful and descriptive variable names to promote easy readability of code and maintain code base understanding. |
| int main(){  int ageInYears = 10;  std::string firstName = "Derek";  std::string lastName = "Clark";  std::cout << "My name is " << firstName << " " << lastName << " " << ageInYears << std::endl;  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 – The utilization of meaningful and descriptive naming conventions is regarded as a secure coding practice by ensuring the design and flow of the program is clear.  Keep it simple – A simple descriptive name will contribute to the readability of a program as it will ensure simplicity and clearly indicate the purpose of the variable or function.  Use effective quality assurance techniques – Effective quality assurance testing relies heavily of the readability and understanding of what the code does, by ensuring easy readability QA testers will have an easier time testing and identifying potential issues found within the code leading to a higher quality maintainable code base. |

**Threat Level**

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

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| Parasoft | [Insert text.] | Built-in static analysis rules | Can establish custom rules for naming. |
| Clang-tidy | 21.0.0 | Readability-identifier-naming | Tool that identifies naming style mismatches and attempts to enforce coding guidelines on naming. |

**Coding Standard 10**

|  |  |  |
| --- | --- | --- |
| **Coding Standard** | **Label** | **Name of Standard** |
| **Memory Management** | [STD-010-CPP] | Avoid manual memory management when possible. |

|  |
| --- |
| **Noncompliant Code** |
| Non-compliant code provides example of manual memory management that has led to a memory leak due to its failure in deallocating the memory that was used to store the int. Although the code would appear fine with deallocating it is good practice to avoid unexpected problems. |
| int main(){  int\* pointer = new int(77);  std::cout << \*ptr << std::endl;  return 0;  } |

|  |
| --- |
| **Compliant Code** |
| Compliant code demonstrates the use of smart pointer: unique pointer which automatically manages memory avoiding manual memory management that could lead to unexpected behavior. |
| int main(){  std::unique\_pointer<int> ptr(new int(77));  std::cout << \*ptr << std::endl;  return 0;  } |

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

|  |
| --- |
| **Principles(s):** Validate input data – By utilizing dynamic memory allocation for storing input data the program is effectively reducing the likelihood of memory management issues from invalid data as space is being automatically resized for the data being stored.  Practice defense in depth – Dynamic memory management can provide an additional layer of defense against memory related vulnerabilities like use after free. |

**Threat Level**

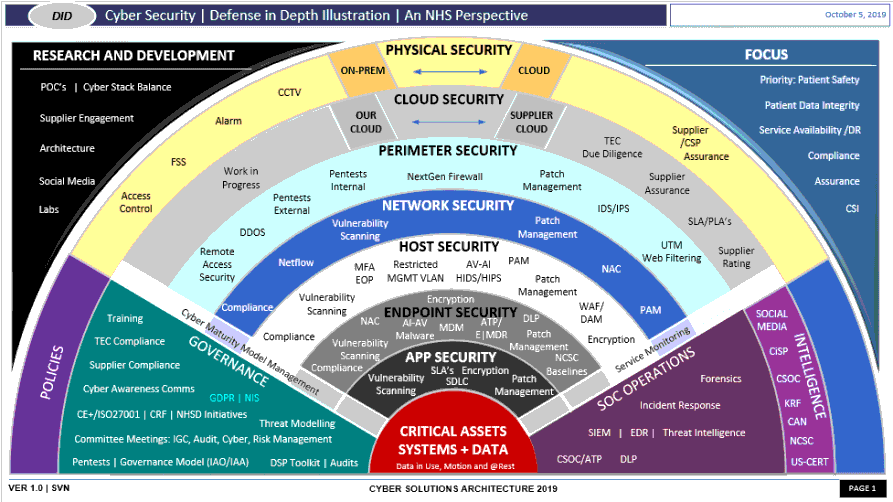
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| High | Probable | Medium | High | L5 |

**Automation**

|  |  |  |  |
| --- | --- | --- | --- |
| **Tool** | **Version** | **Checker** | **Description Tool** |
| CodeSonar | 8.3p0 | Memory should not be managed manually | Rule raises an issue when manual memory management takes place. |
| Parasoft | 10.3.3 | RUN-MEM-UNINT  RUN-MEM-FREEDANG  RUN-MEM-LEAK  RUN-MEM-CORRUPT | Checks and flags memory management issues indirectly calling for more efficient memory management methods. |
| Clang Static Analyzer | 1.1.2.4 | Cplusplus.NewDelete (C++)  Cplusplus.NewDeleteLeaks | Checks for double free, use-after-free problems, and memory leaks tracing memory by new delete. |

**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**

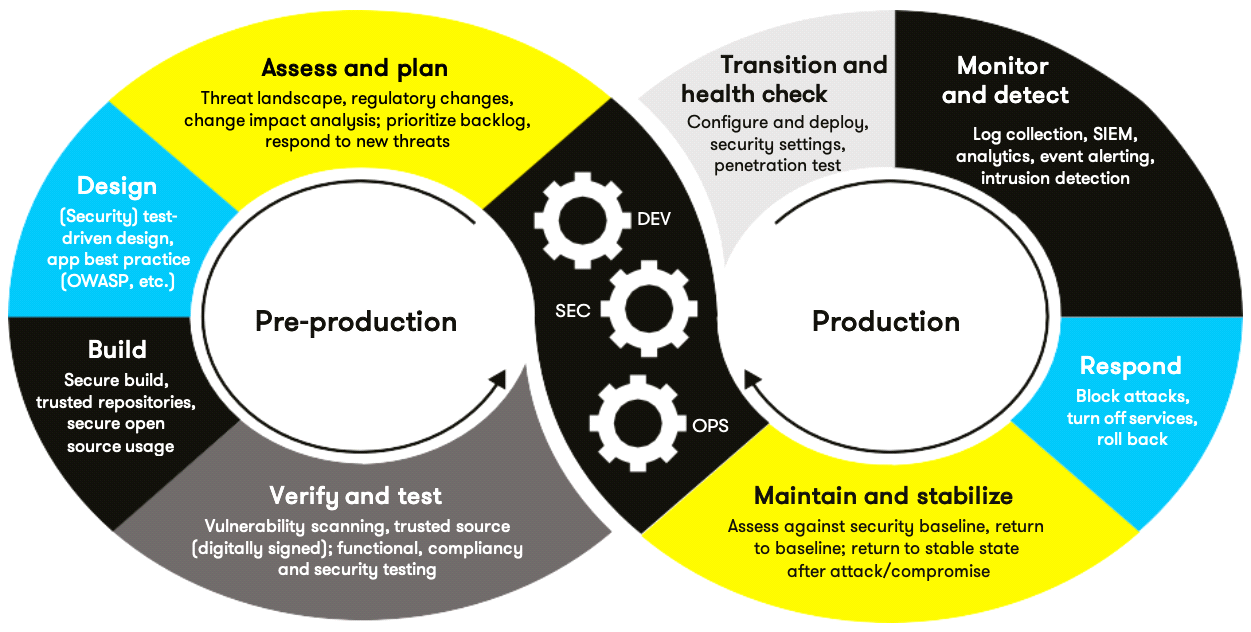
Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority,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



To ensure compliance with the standards outlined in this security policy Green Pace must integrate automation across the entire DevOps process, transforming it into a DevSecOps process. By embedding security into the CI/CD pipeline, we enforce secure coding practices, policy/regulation compliance, and risk mitigation at every stage of development. The current process in place relies heavily on the verification and testing phase to identify bugs and vulnerabilities after the code has been merged which does not comply with the “Don’t leave security to the end” best practice. Instead, the use of automation tools should be incorporated early and continuously across pre-production and production phases.

Automation should take place in pre-production, to proactively eliminate all vulnerabilities before code reaches production. During the planning and design phase automated threat modeling tools such as IriusRisk should be utilized to proactively identify potential threats in the architecture and tech stack. Threat modeling will not only detect risks but provide automated security recommendations before and during development. During the build and testing phase the use of static analysis tools such as cppcheck and sonarqube (IDE) should be frequently used to identify vulnerabilities and ensure compliance with coding standards before code is pushed to production. The testing phase should also incorporate automated testing tools such as google test to not only debug the code but automatically identify and run test eliminating the need for manual test registration.

Automation should also be applied in production to ensure continuous compliance with security policies and coding standards. Automated dependency checking tools such as OWASP dependency checker should be integrated into the maintenance phase ensuring that all dependencies are both up to date and secure mitigating vulnerabilities identified in outdated versions. Updates made to the code base should continuously monitored and tested using the previously mentioned automation tools to ensure no additional vulnerabilities are introduced.

**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 | Medium | Likely | Medium | Medium | 3 |
| STD-002-CPP | Medium | Likely | Medium | Low | 2 |
| STD-003-CPP | High | Likely | Medium | High | 5 |
| STD-004-CPP | High | Likely | High | High | 5 |
| STD-005-CPP | High | Probable | High | High | 5 |
| STD-006-CPP | Low | Probable | Low | Low | 1 |
| STD-007-CPP | High | Probable | Medium | High | 4 |
| STD-008-CPP | High | Probable | Medium | High | 5 |
| STD-009-CPP | Low | Probable | Low | Medium | 2 |
| STD-010-CPP | High | Probable | Medium | High | 5 |

**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***.***

* Explain each type of encryption, how it is used, and why and when the policy applies.
* 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.

|  |  |
| --- | --- |
| * **Encryption** | **Explain what it is and how and why the policy applies.** |
| Encryption at rest | Encryption at rest refers to the use of encryption to protect data that is currently being stored on a disk, database, or other data storage device to ensure that it is only accessible and readable to authorized users with the decryption key. Following standard security principles decryptions keys should always be stored separate from encrypted data. Storing sensitive data such as user logins, financial information, or user data should always utilize the encryption algorithm AES-256 for high level security. Utilizing high level encryption on sensitive data that is stored will prevent that data from being understood if access were ever granted to an unauthorized user. |
| Encryption in flight | Encryption in flight refers to the use of encryption when transmitting data across networks ensuring that if the connection between endpoints is hijacked the sensitive data that may be intercepted is not to be understood without the decryption key. Encryption in flight is utilized by encrypting the data before transmission either through asymmetric encryption with a public key and private key or symmetrically with shared secret keys, the data is transmitted over the network, and the receiver uses the secret key or private key to decrypt the data. The encryption keys must be stored separately and securely with 3-month rotations of keys to prevent compromising the integrity of the encrypted data. Encryption in flight should be utilized for all transmissions of sensitive data this includes emails, api calls, and database connections. |
| Encryption in use | Encryption in use refers to the use of encryption to protect data that is currently being accessed or utilized without the need for decryption. Encryptions in use may be utilized for performing computations such as basic mathematical operations like addition or multiplication, statistical operations like finding the average value in a dataset, or even to search and edit file content. Encryption in use minimizes the risk of data exposure when the data is actively being used unlike encryption in flight and at rest. |

|  |  |
| --- | --- |
| * **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| Authentication | Authentication is the process in which a user of a system can be identified. Authentication is typically used to verify a user’s access to a system by matching the user’s login credentials with those that are stored within a login database. A user of a system will typically create an account or have an account provided to them for accessing the system. This account information will usually contain a username or email identifying the user’s identity and a password for verifying that the digital entity is being utilized by the owner of the account. Passwords will have to comply with security standards of being at least 10 characters long, including at least 2 numbers, and one symbol. Additional authentication may be required when a user of the system attempts to make changes to the database, ensuring that all changes are tied to verified user accounts. |
| Authorization | Authorization is the process in which varying levels of system permissions are granted and enforced. Authorization enforces that access is only granted to users with explicit permission. Authorization may permit higher ranking users to access restricted files and datasets and may deny lower ranking users as they are not permitted via role-based access control. Authorization also plays a role in the addition of new users as users with explicit permissions to do so may create new users for onboarding. Authorization is applied to protect sensitive data, ensure compliance with legal requirements/regulations, and significantly reduces the attack surface by limiting the number of users with high level access. |
| Accounting | Accounting is the process in which activities or events in a system are tracked to help detect suspicious activity and find the root cause of an issue. Working alongside authentication accounting may use login credentials like usernames and emails to identify specific users performing actions in the system to help identify which users are doing what. Accounting may also work alongside authorization as lower access users attempting to perform actions in a system may be flagged for further inspection assisting in finding malicious users or safe users who have had their information compromised. Accounting may also assist in identifying what users are making changes to databases which can then allow database administrators to connect bad database changes to users who may require further training. |

**\***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 | 01/05/2025 | Initial Template | David Buksbaum |  |
| 1.5 | 01/26/2025 | Security principles and coding standards developed including compliant and non-compliant examples. | Derek Clark | Derek Clark |
| 2.0 | 2/20/2025 | Automation tools and threat levels have been added along with guidance for integrating automation into CI/CD. | Derek Clark | Derek Clark |

**Appendix A Lookups**

**Approved C/C++ Language Acronyms**

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