**Bradley Byard: Security Policy Guide**



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** | Explanation |
| --- | --- |
| 1. ValidateInput Data | Don’t trust and validate all user input. This may be by filtering the input. This is a preventative measure that can protect and application from accidental misuse of a program and also against deliberate attacks, such as SQL injection, where a user attempts to inject malicious commands into the code base. |
| 1. Heed Compiler Warnings | Don’t ignore compiler warnings. In C++ the compiler has a good grasp of what is going on. Ignoring compiler warnings could lead to erroneous application behavior. If you pay attention to compiler warning you could avoid this. |
| 1. Architect and Design for Security Policies | Developers need to consider every feature of an application they are designing and be confident that the process that’s surrounding these features are as safe as possible. |
| 1. Keep It Simple | Developers should try to avoid making an application too sophisticated when making security controls. When you have architecture or too many complex mechanisms, you then can increase risk of error. |
| 1. Default Deny | If you don’t specifically allow something, then you should be denying it. Come up with and define what you allow, and allow nothing else. This helps keep applications secure. |
| 1. Adhere to the Principle of Least Privilege | A user of a system should only be allowed privileges needed to complete it’s task. All other tasks should not be accessible. If a user does not need access ability, the user should not have the right. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data, such as command shells and relational databases. Attackers may invoke unused fucntionality through SQL, command, and other injection attacks. The calling process understands the context, so it is in charge of sanitizing data before invoking subsystem. |
| 1. Practice Defense in Depth | Defense in Depth states that having many security controls that approach risks in lots of different ways is the best way to secure an application. This means having multiple layers of security, and adding security and logging tools. |
| 1. Use Effective Quality Assurance Techniques | These techniques can identify and eliminate vulnerabilities. Security reviews can lead to a much more secure system. They can bring a new perspective such as identifying and fixing invalid assumptions. |
| 1. Adopt a Secure Coding Standard | It’s best to develop a secure coding standard for your development language and platform in all instances of software. All software needs to be secure and should follow some type of secure standard. |

## 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-C++] | **C++ Secure and Valid Data Type Standard** -- It is important to use correct data types for variables. Data types are used to tell the variables the type of data it can store. |

| **Noncompliant Code** |
| --- |
| The variable thisVariable is declared as data type int yet is defined as a string. This in incorrect. The variable secondVariable is declared as a string but defined as an integer. If you were to write String secondVariable = ‘2’; you would store 2 as a string and this would be correct. |
| int main()  {  int thisVariable = ‘Hello World’;  string secondVariable = 2;  } |

| **Compliant Code** |
| --- |
| The variable thisVariable is declared as data type int and defined as the integer value 2, this is correct. The variable secondVariable is declared as a string and defined as a string ‘2’. |
| int main()  {  int thisVariable = 2;  string secondVariable = ‘2’;  } |

|  |
| --- |
| **Principles(s):** *Heed Compiler Warning:* The compiler will warn you when declare an int and then define it as a string. It will give you this message in this instance “a value of type const char\* cannot be used to initialize an entity of type int. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Low | Medium | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err34-c | Checked by clang-tidy |
| CodeSonar | 6.1p0 | BADFUNC.ATOF  BADFUNC.ATOI  BADFUNC.ATOL  BADFUCN.ATOLL | Use of atof  Use of atoi  Use of atoll  Use of atoll |
| LDRA Tool Suite | 9.7.1 | 44 S | Fully Implemented |

### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | [STD-002-C++] | **C++ Proper Data Values** -- It is important to use correct data values for variables because data types have limits of the number of bits each data type can be. Data types are used to tell the variables the type of data it can store. All code should be able to handle user input of data values, and not allow for over/underflow |

| **Noncompliant Code** |
| --- |
| Here we added 1 to y and y came back as zero. This is because the range of an unsigned integer is 0 to 4294967295. If we knew that the data value of y was going to be that big, we should have declared y as a bigger type than an integer, such as an unsigned long long, which has a max value of 18446744073709551615. This is an overflow issue. |
| int main()  {  unsigned int y = 0;  std::cout << y << std::endl;  y = y + 4294967295;  std::cout << y << std::endl;  y = y + 1;  std::cout << y << std::endl;  }  OUTPUT:  0  4294967295  0 |

| **Compliant Code** |
| --- |
| Here we see that the output came out correctly because we user a proper data type for this data value. An unsigned long long that supports a max value of 18446744073709551615. |
| int main()  {  unsigned long long y = 0;  std::cout << y << std::endl;  y = y + 4294967295;  std::cout << y << std::endl;  y = y + 1;  std::cout << y << std::endl;  }  OUTPUT:  0  4294967295  4294967296 |

|  |
| --- |
| **Principles(s):** *Architect and Design for Security Policies:* This code problem would be unsecure and provide invalid output making the program fail to output proper data. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unikely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | JAVA.MATH.ABSTRAND  JAVA.ARITH.OFLOW | Abs on random  Cast: int Computation to long (Java) |
| Coverity | 7.5 | BAD\_SHIFT | Implemted |

### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | [STD-003-C++] | C++ String Correctness – [Making sure that Strings have sufficient space for storage and character data and the null terminator](https://wiki.sei.cmu.edu/confluence/display/cplusplus/STR50-CPP.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator) |

| **Noncompliant Code** |
| --- |
| This user input of a character array gives bounds (12) to the array or string. We should try to avoid giving boundaries to user input when we can, because the user input is unbounded. If a user enters too long of a value, this could lead to buffer overflow. |
| int main()  {  Char test[12];  Std::cin >> test;  } |

| **Compliant Code** |
| --- |
| This solution shows allowing user input with no bounds on string declaration. This is a valid solution for avoiding buffer overflow with Strings. |
| int main()  {  std::string input;     std::string stringOne, stringTwo;     std::cin >> stringOne >> stringTwo;  } |

|  |
| --- |
| **Principles(s):** *Validate Input Data:* We are taking user input here with this data. We also want to be able to validate that input. We have no control over what the user inputs into the system so we need to make sure that no matter what input comes in, it does not cause a system error. A bounded array can cause this problem. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astrée | 20.10 |  | Supported: Astrée can detect subsequent code defects that this rule aims to prevent. |
| ECLAIR | 1.2 | CC2.STR36 | Fully implemented |

### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | [STD-004-C++] | **C++ Preventing SQL Injection** – SQL injection can destroy a database, or give hackers access to the information in a database, causing a data leak. |

| **Noncompliant Code** |
| --- |
| This SQL query line takes user input and uses in directly in the code. This allows the user to inject a malicious statement into the code. If a user entered something like ‘myUsername OR 1=1’ then since 1=1 is always true, this could return all rows from the “Users” table, giving a hacker that information. |
| {  txtUserId = getRequestString("UserId");  txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;  } |

| **Compliant Code** |
| --- |
| To protect from malicious SQL injection, you can use SQL parameters and parameterize user input. Parameters are shown in the code by the @ sign. The SQL engine would now check every parameter to make sure that its columns are correct and treated literally, not as part of the SQL to be actually executed. |
| {  txtUserId = getRequestString("UserId");  txtSQL = "SELECT \* FROM Users WHERE UserId = @0";  db.Execute(txtSQL,txtUserId);  } |

|  |
| --- |
| **Principles(s):** *Sanitize Data Sent to Other Systems, Validate User Input*: Here you again are taking user input from someone. You don’t know what their input will be. You need to validate it and not allow user to inject malicious commands be securely parameterizing user input which will more securely validate it. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | Medium | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| The Checker Framework | 2.1.3 | Tainting Checker | Trust and Security Errors |
| CodeSonar | 6.1p0 | JAVA.IO.INJ.SQL | SQL Injection |
| Coverity | 7.5 | SQLI | Implemented |
| Fortify | 1.0 | SQL\_Injection\_persistence  SQL\_Injection | Implemented |

### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | [STD-005-C++] | **C++ Memory Protection** – Every variable created in programming represents a storage space in computer memory. This means that each variable uses an area of memory to store its value and we must make sure we are taking care of that stored memory properly. |

| **Noncompliant Code** |
| --- |
| In this example, local\_pointer() returns a pointer to an integer. The local integer, is allocated only while local\_pointer() is running. When it exits, all of its locals are deallocated. We want the integer to exist, but it gets deallocated automatically. |
| int\* local\_pointer()  {  int temp = 100;  // returns a pointer to the local int  return(&temp);  }  int main()  {  int\* ptr = local\_pointer();  return 0;  } |

| **Compliant Code** |
| --- |
| These variables are considered local, to show that their memory lifetime is tied to the function where they are declared at. When this function is running and used in code, its variables are then allocated in memory. When the function exits, then the local variables are deallocated. |
| int SquareNumber(int number) {  int answer;  result = number \* number;  return answer;  } |

|  |
| --- |
| **Principles(s):** *Architect and Design for Security Polices:* This is a big problem with any computer program. To be secure we have to make sure memory access and allocation is not overwriting, deleting, or change things that it shouldn’t be or this could cause many security problems. The architect and design of any code should follow this policy |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 7.5 | CHECKED\_RETURN | Finds Inconsistencies in how function call return values are handled |
| Parasoft | 2021.1 | CERT\_CPP-MEM52-a  CERT\_CPP-MEM52-b | Do not allocate resources in function argument list because the order of evaluation of a functions parameters is undefined. |
| LDRA tool suite | 9.7.1 | 45D | Partially Implemented |

### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | [STD-006-C++] | **C++ Assertions** -- Assertions are entirely appropriate in C++ code. Assertions are for debugging. If an assertion is shown, the code needs to be fixed. |

| **Noncompliant Code** |
| --- |
| Asserts should only be used for testing things the developer has assumed. Your program should never rely on the assert macro running. |
| Int main()  {  X=0;  assert(x++);  assert(function());  } |

| **Compliant Code** |
| --- |
| The non-compliant example above is fixed here. The methods shown below user assert in a much safer way. |
| Int main()  {  x=0;  assert(x);  x++;  int variable = function();  assert(variable);  } |

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

|  |
| --- |
| **Principles(s):** *Use effective Quality Assurance Techniques:* Adding security testing with using things like “assert” properly in unit testing, proves that you are trying to make your code secure. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Unlikely | High | Medium | 1 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| CodeSonar | 6.1p0 | Custom | Users can implement a custom check that reports uses of the assert() macro |
| Compass/ROSE |  |  | Detects Violations by looking for calls to assert() |
| ECLAIR | 1.2 | CC2.DCL03 | Implemented |
| LDRA tool suite | 9.7.1 | 44 S | Implemented |

### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | [STD-007-C++] | **C++ Exceptions** – Exception handling is very important in secure coding because abnormal exception termination is a typical channel for denial-of-service attacks. |

| **Noncompliant Code** |
| --- |
| No matching handler is found for the exception that is thrown here, so the process terminates or aborts. |
| void throw\_func() noexcept(false);    void f() {    throw\_func();  }    **int** main() {    f();  } |

| **Compliant Code** |
| --- |
| This code fixes the above code by adding in a try-catch methodology. This handles all exceptions, and makes sure that no exceptions cause abnormal termination. |
| void throw\_func() noexcept(false);    void f() {    throw\_func();  }    **int** main() {    try {  f();  } catch (…){  //Handle error here  }  } |

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

|  |
| --- |
| **Principles(s):** *Use effective Quality Assurance Techniques, Default Deny:* Exceptions help handle errors and unexpected activity by the program that could be caused be user actions. They help point to areas of code that may have initially caused the error. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Low | Likely | Low | Low | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Potentially-throwing-static-initialization | Partially checked |
| Clang | 3.9 | Cert-err58-cpp | Checked by clang-tidy |
| Rule Checker | 20.10 | Potentially-throwing-static-initialization | Partially checked |
| Parasoft | 2021.1 | CERT\_CPP-ERR58-a | Exceptions shall be raised only after start-up and before termination of the program. |

### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Validate User Input** | [STD-008-C++] | **C++ validating user Input** – Whenever asking for user input it is import to validate that the user is typing in what you are looking for, especially if that value is stored in a variable of a certain data type. |

| **Noncompliant Code** |
| --- |
| In this case, a was declared as an integer. That means for it to store properly the user needs to enter an integer. If they do the program is okay, but if not the program will error, and we have nothing to handle that error. |
| Int main()  {  Int a;  cout<<”Enter an integer number\n”;  cin>>a;  } |

| **Compliant Code** |
| --- |
| This example takes input from the user, and only allows the program to store the variable data if data entered is an integer. If anything else is entered, the user is informed that they entered the wrong input. |
| int main()  {  int a;  cout<<“Enter an integer number\n”;  cin>>a;  while(1)  {  if(cin.fail())  {  cin.clear();  cin.ignore(numeric\_limits<streamsize>::max(),’\n’);  cout<<“You have entered wrong input”<<endl;  cin>>a;  }  if(!cin.fail())  break;  } |

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

|  |
| --- |
| **Principles(s):** *Validate User Input*: The code shown above, makes sure that the user enters an integer, and will only accept that. No other input allows the code to continue. This is a good way to validate what the user enters. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Likely | Medium | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Clang | 3.9 | Cert-err34-c | Checked by clang-tidy |
| LDRA Tool Suite | 9.7.1 | 44 S | Fully Implemented |

### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Closing Files Properly** | [STD-009-C++] | **C++ Closing Files Properly –** When using and loading files, after a file is used, it is important to properly close that file. |

| **Noncompliant Code** |
| --- |
| In this example, destructors are not called, std::basic\_filebuf<T> object maintained by the object is not closed properly. The command here std::fstream::close() is not called. |
| void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    std::terminate();  } |

| **Compliant Code** |
| --- |
| In this examples, errors are handled properly, and the file is closed properly. The command std::fstream::close() is called and the file should be closed properly. |
| void f(const std::string &fileName) {    std::fstream file(fileName);    if (!file.is\_open()) {      // Handle error      return;    }    // ...    file.close();    if (file.fail()) {      // Handle error    }    std::terminate();  } |

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

|  |
| --- |
| **Principles(s):** *Adopt a Secure Coding Standard:* Just a general secure coding standard all files should only be open when they are needed, and then closed when they are no longer needed. This helps keep data safe. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Unlikely | Medium | Low | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 |  | Supported |
| CodeSonar | 6.1.p0 | ALLOC.LEAK | Leak |
| Coverity | 2017.07 | RESOURCE\_LEAK | Partially implemented |
| LDRA tool suite | 9.7.1 | 49 D | Partially Implemented |

### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Iterator Ranges | [STD-010-C++] | **C++ Valid Iterator Range** – iterators muse iterate over a valid range. |

| **Noncompliant Code** |
| --- |
| In this example, the iterator values are passed in the correct order. This will result in undefined behavior. |
| void f(const std::vector<**int**> &c) {    std::for\_each(c.end(), c.begin(), [](**int** i) { std::cout << i; });  } |

| **Compliant Code** |
| --- |
| In this example, the iterator values are passed in the correct order. |
| void f(const std::vector<**int**> &c) {    std::for\_each(c.begin(), c.end(), [](**int** i) { std::cout << i; });  } |

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

|  |
| --- |
| **Principles(s):** *Heed Compiler Warning:* heed compiler warnings her, because for example, using a range of two iterators that are invalidated or do not refer into the same container results in undefined behavior, while the compiler may warn you of this. |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Probable | High | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Astree | 20.10 | Overflow\_upon\_dereference |  |
| Helix QAC | 2021.2 | C++3802 |  |
| Parasoft | 2021.1 | CERT\_CPP-CTR53-a  CERT\_CPP-CTR53-b | Do not use an iterator range that isn’t really a range, do not compare iterators from different containers |

## 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.

[**DevSecOps**](https://www.appdynamics.com/topics/what-is-devsecops) evolved from DevOps as development teams began to realize that the DevOps model didn’t adequately address security concerns (Maerz, 2021). Instead of retrofitting security into the build, DevSecOps emerged as a way to integrate the management of security earlier on throughout the development process (Maerz, 2021). Through this method, application security begins at the outset of the build process, instead of at the end of the development pipeline (Maerz, 2021). With this new approach, an engineer of DevSecOps strives to ensure that apps are secure against cyberattacks before being delivered to the user, and are continuously secure during app updates (Maerz, 2021). DevSecOps emphasizes that developers should create code with security in mind and aims to solve the issues with security that DevOps doesn’t address (Maerz, 2021).

DevSecOps approach includes the above practices, as well as:

* Common weaknesses enumeration (CWE) – improves the quality of code and increases the level of security during the CI and CD phases
* Threat modeling – implements security testing during the development pipeline to save time and cost in future
* Automated security testing – test for vulnerabilities in new builds on regular basis
* Incident management – creates a standard framework for responding to security incidents

SOURCE:

Maerz, C. (2021, July 7). *What's the difference between devops and devsecops?* AppDynamics. https://www.appdynamics.com/blog/product/devops-vs-devsecops/.

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

* 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 | Encrypting data at rest can protect the organization from unauthorized access to data when computer hardware is sent for repair or discarded. |
| Encryption at flight | The process of encrypting data while the data is being transmitted. In some applications, such as remote replication, data may be unencrypted while it is at rest on drive arrays, but encrypted while it is being transmitted to provide protection. |
| Encryption in use | Protecting data at rest is far easier than protecting [data in use](https://whatis.techtarget.com/definition/data-in-use) -- information that is being processed, accessed or read  and [data in motion](https://whatis.techtarget.com/definition/data-in-motion) -- information that is being transported between systems. Encryption plays a major role in protecting data in use or in motion. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Successful authentication occurs when one’s credentials are entered into a system resource (i.e., such as typing in a username or password) and compared against stored user information with a database, which ultimately allows a user to gain access, or be denied (Flank, n.d.) |
| Authorization | With Authorization we usually use a username and password type setup, or some other type of commonly used method of "authentication" to actually authenticate to that very system resource (Flank, n.d.) Once users have successfully identified and authenticated themselves, they then are "authorized" to perform certain functions within those system resources based on the access rights afforded to them (Flank, n.d.). Users could be a general user, or an Admin, depending on their role in the system, but have different authority, doing only the things they need with only access to what they need, blocking them from other portions. |
| Accounting | the concept of "accounting" (i.e., effectively auditing and monitoring this type of environment) includes removing aged and dormant accounts, validating access rights for privileged accounts, reviewing log reports for access rights violations, and other essential activities (Flank, n.d.) |
| Source | Flank. (n.d.). *Why Information Security Policies and Procedures are Important*. Flank. https://flank.org/faqs/what-is-aaa. |

**\***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 | 7-25-21 | Security Policy module 3 | Brad Byard |  |
| 1.1 | 8-8-21 | Security Policy Module 6 | Brad Byard |  |

# Appendix A Lookups

## Approved C/C++ Language Acronyms

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