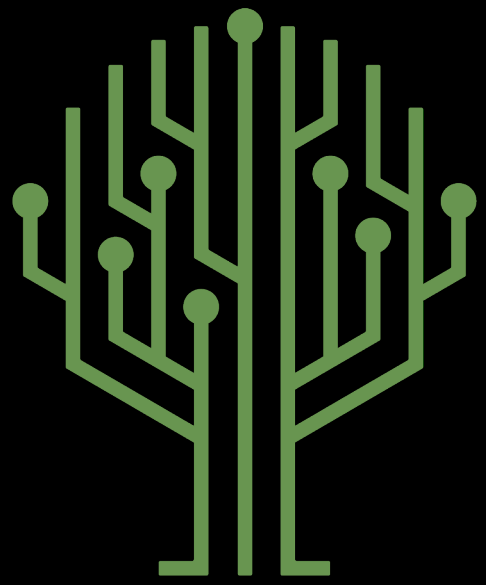
Green Pace

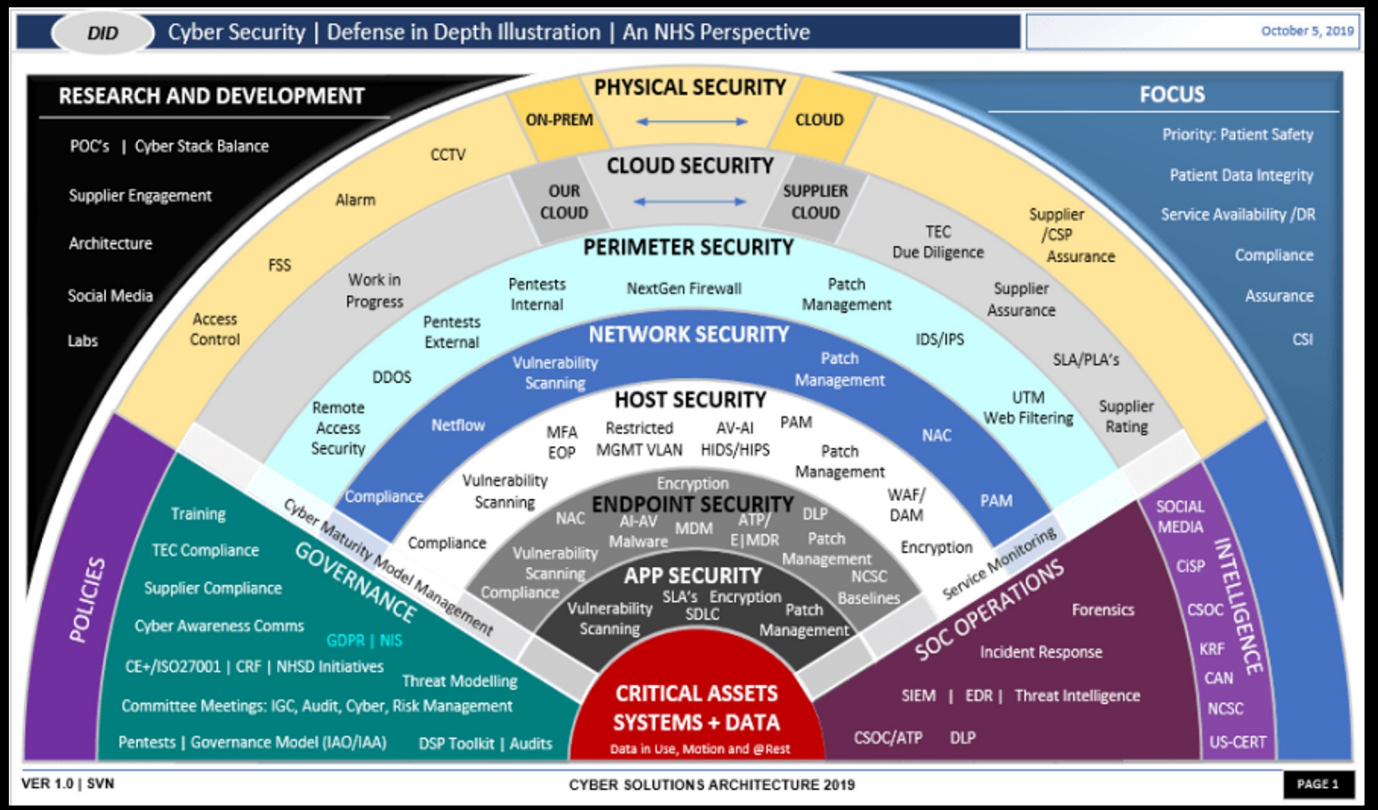
Security Policy Presentation

Developer: Eric Slutz



OVERVIEW: DEFENSE IN DEPTH

The purpose of this security policy is set a collection of standards and best practices to maintain the safety of our software and data. The different parts of this policy make up some of the layers the Defense in Depth strategy.



Green Pace

THREATS MATRIX

Low Priority: Unlikely to happen with a low severity

Likely: Probable or likely to happen with a low severity

Unlikely: Unlikely to happen but with a high severity

Priority: Probable or likely to happen with a high severity

Likely STD-002-CPP STD-007-CPP	**Priority** STD-004-CPP STD-005-CPP
Low Priority STD-001-CPP STD-006-CPP STD-009-CPP STD-010-CPP	**Unlikely** STD-003-CPP STD-008-CPP

Green Pace

10 PRINCIPLES

1	Validate Input Data	STD-004-CPP
2	Heed Compiler Warnings	STD-001-CPP, STD-002-CPP
3	Architect and Design for Security Policies	STD-006-CPP, STD-007-CPP, STD-008-CPP
4	Keep It Simple	STD-002-CPP, STD-004-CPP, STD-008-CPP, STD-010-CPP
5	Default Deny	STD-009-CPP
6	Adhere to the Principle of Least Privilege	STD-009-CPP
7	Sanitize Data Sent to Other Systems	
8	Practice Defense in Depth	STD-005-CPP
9	Use Effective Quality Assurance Techniques	STD-003-CPP, STD-005-CPP, STD-007-CPP
10	Adopt a Secure Coding Standard	STD-003-CPP, STD-008-CPP, STD-010-CPP

CODING STANDARDS

(in order of priority)

STD-004-CPP	Sanitize data passed to complex subsystems
STD-005-CPP	Properly deallocate dynamically allocated resources
STD-003-CPP	Range check element access
STD-008-CPP	Do not modify the standard namespaces
STD-002-CPP	Value-returning functions must return a value from all exit paths
STD-007-CPP	Handle all exceptions
STD-001-CPP	Never qualify a reference type with const or volatile
STD-006-CPP	Understand the termination behavior of assert() and abort()
STD-009-CPP	Close files when they are no longer needed
STD-010-CPP	Write constructor member initializers in the canonical order

Green Pace

ENCRYPTION POLICIES

Encryption in rest	Encryption at rest refers to data being in an encrypted state while it is in storage. This means that even if access to the data is gained, the data itself is unreadable unless you have the key to decrypt the data. This policy is important and should be used so that if there were a breach and the data was taken, the information within the data would still be safe because it could not be read without the key.
Encryption at flight	Encryption at flight refers to data being in an encrypted state while it is in transit from one place to another. While data is in route form one place to another, it must be encrypted. This ensures that if the data is intercepted, the information within the data would still be safe because it could not be read without the key.
Encryption in use	Encryption in use refers to the encryption of data as it is being used by the system. It is used to keep data secure at all times, even as it is changing. This is important because as more data is secured with encryption at rest and in transit, then that leave as data is being used as the weakest link to be exploited. For that reason, it is important to figure out how to keep the data encrypted as it is being used.

Green Pace

TRIPLE-A POLICIES

Authentication	Authentication is the process of confirming that a user is who they say they are. It is typically used when adding new users or when a user attempts to login. This policy applies whenever a user is attempting to gain access to a system or something within the system. You need to be sure that the person within the system is who you expect it to be. This is typically accomplished by verify user identification information such as a username and password.
Authorization	Authorization is the process of confirming that a user is supposed to have access to what they are requesting. It determines the level of access a user has with the system. This policy applies whenever a user is attempting to access a system or something within the system. Once authenticated, authorization will determine if the user can gain entry to the system. This helps to keep users out of places they shouldn't be.
Accounting	Accounting is the process of tracking and logging all requests and transactions made by a user. Examples of that would be changes made to a database or files accessed by a user. This policy is important so that you can know what is going on with a system. For example, if the system starts behaving in an unexpected manor, you can review the logs to track down the issue and then hopefully be able to figure out a solution.

Green Pace

Unit Testing

CanAddFiveValuesToCollection

```cpp
// Test to verify adding five values to an empty collection.
TEST_F(CollectionTest, CanAddFiveValuesToCollection)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Add five elements to the collection.
    add_entries(5);

    // Test that collection is not empty.
    ASSERT_FALSE(collection->empty());

    // Test that collection size is 5.
    ASSERT_EQ(collection->size(), 5);
}
```

```
[ RUN      ] CollectionTest.CanAddToEmptyCollection
[       OK ] CollectionTest.CanAddToEmptyCollection (0 ms)
```

Green Pace

Unit Testing

MaxSizeGreaterThanSize

```cpp
// Test to verify that max size is greater than or equal to size for 0, 1, 5, 10 entries.
TEST_P(ParameterizedCollectionTest, MaxSizeGreaterThanSize)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Add number of elements specified by the parameter to the collection.
    add_entries(GetParam());

    // Test that collection size matches parameter value.
    ASSERT_EQ(collection->size(), GetParam());

    // Test that collection max size is greater than collection size.
    ASSERT_GT(collection->max_size(), collection->size());
}
```

```
[ RUN      ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/0
[       OK ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/0 (0 ms)
[ RUN      ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/1
[       OK ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/1 (0 ms)
[ RUN      ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/2
[       OK ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/2 (0 ms)
[ RUN      ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/3
[       OK ] CollectionSizes/ParameterizedCollectionTest.MaxSizeGreaterThanSize/3 (1 ms)
```

Unit Testing

ResizeIncreasesCollectionSize

```
// Test to verify resizing increases the collection.
TEST_F(CollectionTest, ResizeIncreasesCollectionSize)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Resize the collection to 5.
    collection->resize(5);

    // Test that collection size is increased to 5.
    ASSERT_EQ(collection->size(), 5);
}
```

```
[ RUN      ] CollectionTest.ResizeIncreasesCollectionSize
[       OK ] CollectionTest.ResizeIncreasesCollectionSize (0 ms)
```

Green Pace

Unit Testing

ClearCollection

```cpp
// Test to verify clear erases the collection
TEST_F(CollectionTest, ClearCollection)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Add five elements to the collection.
    add_entries(5);

    // Test that collection is not empty.
    EXPECT_FALSE(collection->empty());

    // Test that collection size is 5.
    EXPECT_EQ(collection->size(), 5);

    // Clear the collection.
    collection->clear();

    // Test that collection is empty.
    ASSERT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    ASSERT_EQ(collection->size(), 0);
}
```

```
[ RUN      ] CollectionTest.ClearCollection
[       OK ] CollectionTest.ClearCollection (0 ms)
```

Unit Testing

OutOfRangeExceptionThrown

```cpp
// Test to verify the std::out_of_range exception is thrown when calling at() with an index out of bounds
// NOTE: This is a negative test
TEST_F(CollectionTest, OutOfRangeExceptionThrown)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Test that expected exception is thrown.
    ASSERT_THROW(collection->at(5), std::out_of_range);
}
```

```
[ RUN      ] CollectionTest.OutOfRangeExceptionThrown
[       OK ] CollectionTest.OutOfRangeExceptionThrown (3 ms)
```

Unit Testing

IncreaseCollectionReserveAboveMaxSize

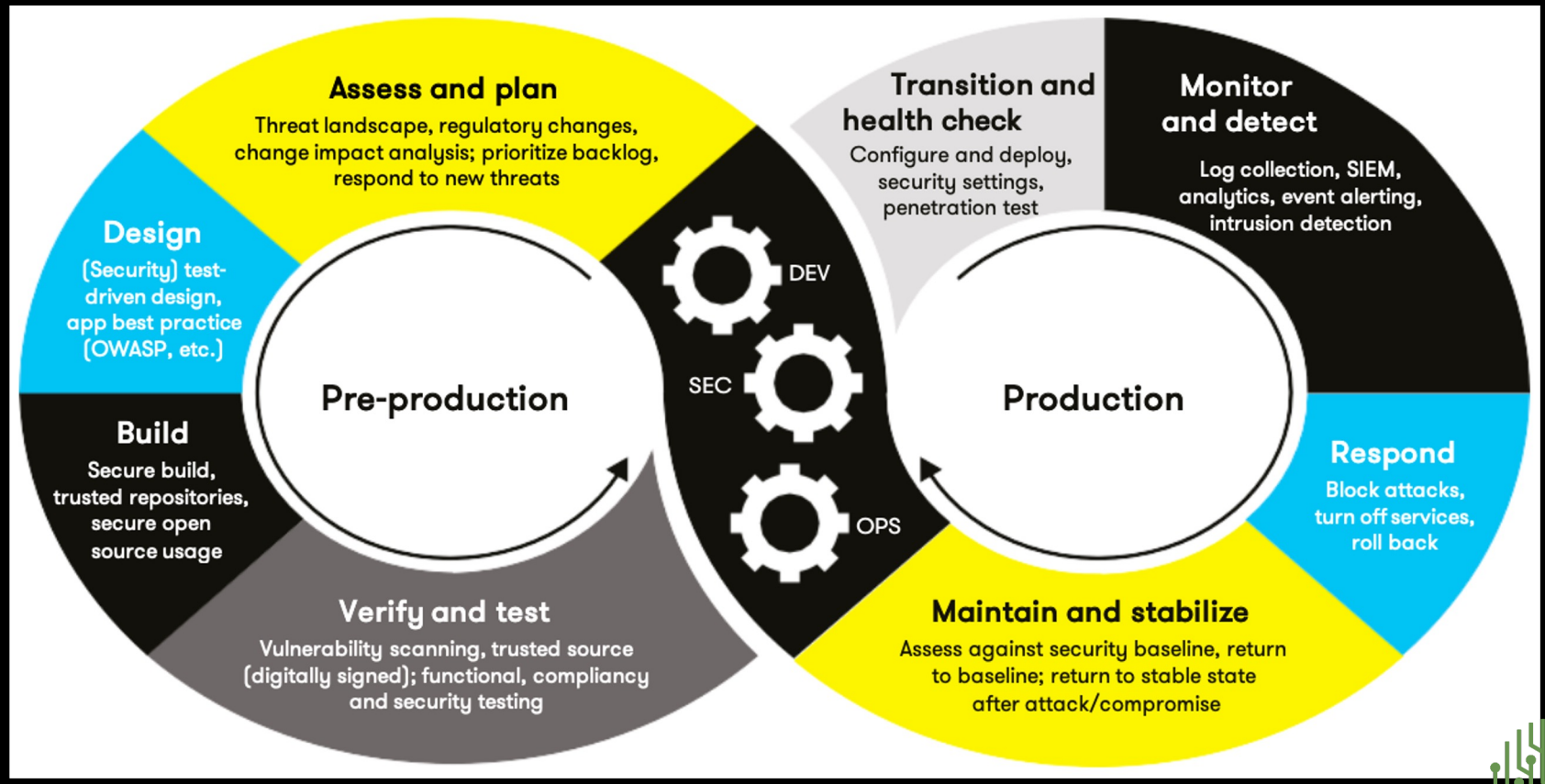
```
// Test to verify collection cannot have a reserve larger than the collection max size.
// NOTE: This is a negative test
TEST_F(CollectionTest, IncreaseCollectionReserveAboveMaxSize)
{
    // Test that collection is empty.
    EXPECT_TRUE(collection->empty());

    // Test that if collection empty, then size must be 0.
    EXPECT_EQ(collection->size(), 0);

    // Test reserving a size larger than the max size od the collection.
    ASSERT_THROW(collection->reserve(collection->max_size() + 10), std::length_error);
}
```

```
[ RUN      ] CollectionTest.IncreaseCollectionReserveAboveMaxSize
[       OK ] CollectionTest.IncreaseCollectionReserveAboveMaxSize (2 ms)
```

Green Pace

AUTOMATION SUMMARY

- DevSecOps can help to ensure you are creating the most secure application possible and automation can help with this process
- The polices and standards set within this security policy document should be utilized during the DevSecOps process



Green Pace

TOOLS

- The DevSecOps pipeline shows the flow the development process goes through
  - Pre-production: steps to design and test the software
  - Production: steps for deploying, monitoring, and maintaining the software

- Design stage – OWSAP or Security Policy for creating a secure design
- Build stage – IDE/compiler for compiling and running the code
- Verify and Test stage – Static analysis and unit testing tools
- Monitor and Detect stage – Logging and application monitoring tools

RISKS AND BENEFITS

- Should security be implemented now or at a later time?
  - NOW!
- Security of your application and data is serious and must be treated that way
- Security should be though of as soon as planning begins
- Benefits
  - Easier to implement security when planned from the start
  - More robust/reliable code
  - More consistent code
- Risks
  - Not enough buy-in
  - Can be difficult and not at secure to go back and add security later
  - Application breached and data exposed if taking a wait and see approach

Green Pace

RECOMMENDATIONS

- A security policy does no good if it is not understood and followed
- DevSecOps training for developers
- Automated monitoring of software dependencies for vulnerabilities and patches to prevent supply chain attacks
  - https://learn.microsoft.com/en-us/nuget/concepts/security-best-practices

CONCLUSIONS

- Security must be considered from the beginning
- Training must be provided on security and the security policy
- Enforcement mechanisms should be in place to ensure the security policy is followed
  - Deployment of an application should not be possible if the security policy requirements have not been met
- The security policy should be regularly reviewed and updated where needed

Green Pace

REFERENCES

- Microsoft. (2022, October 11). Best practices for a secure software supply chain. Microsoft Learn. https://learn.microsoft.com/en-us/nuget/concepts/security-best-practices

- SEI CERT C++ Coding Standard - SEI CERT C++ Coding Standard - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682

Green Pace