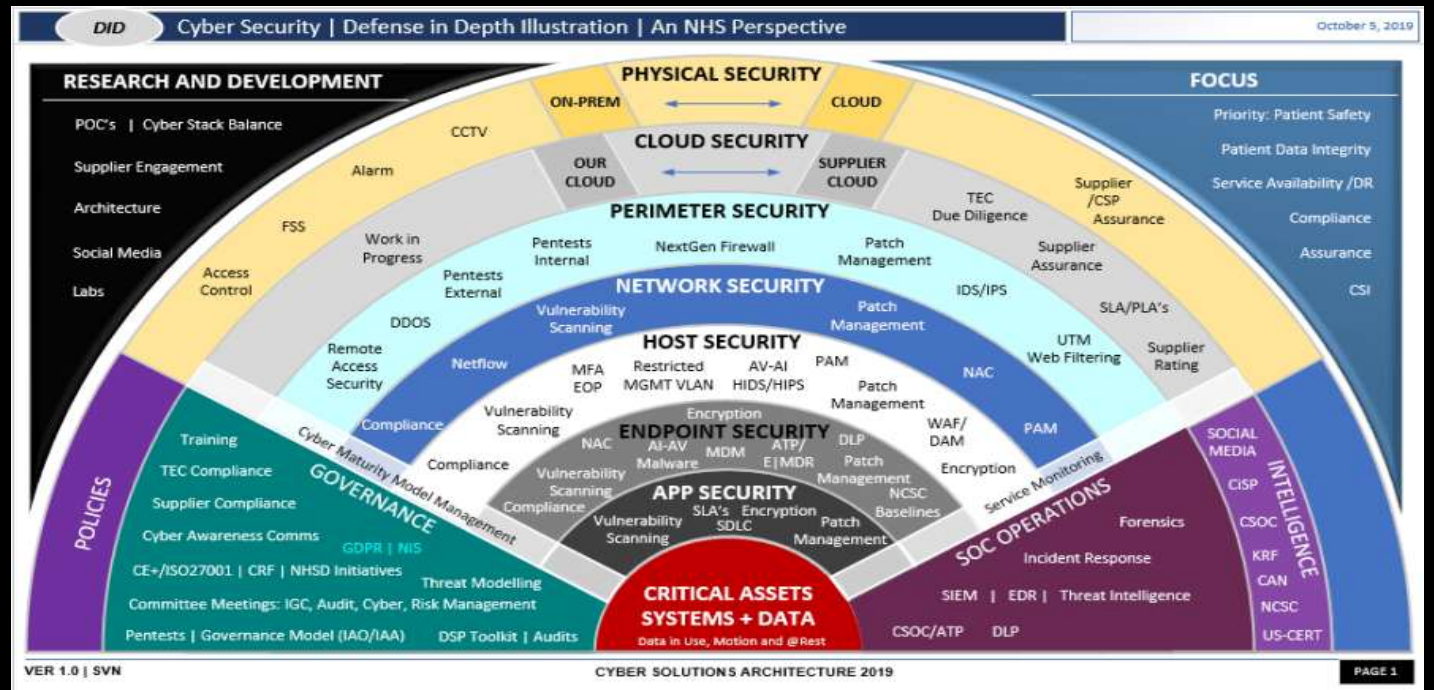
Green Pace

Security Policy Presentation

Developer: *Dylan Cavazos*

OVERVIEW: DEFENSE IN DEPTH

- The Green Pace Security Policy establishes consistent security practices to protect systems and data while addressing evolving cybersecurity challenges and team expansion.
- It defines core principles, including C/C++ coding standards, data encryption standards, and the AAA framework (Authentication, Authorization, and Auditing).
- By embedding security into each phase of the software development lifecycle and leveraging the defense-in-depth strategy, the policy mitigates risks like SQL injection and improper memory management.
- Through proactive measures such as secure coding, automated testing, and continuous monitoring, it reduces vulnerabilities and fosters alignment with best practices.



Green Pace

THREATS MATRIX

Likely (Medium Severity / Moderate Probability)

- **STD-003-CPP – Do not attempt to modify string literals:** Treat string literals as immutable to prevent undefined behavior, thus protecting against potential attacks.

Priority (High Severity / High Probability)

- **STD-001-CPP – C Style variadic functions**: Variadic functions lack type safety and can lead to unpredictable behavior or vulnerabilities.
- **STD-002-CPP – Do not read uninitialized memory:** Accessing uninitialized memory can cause system crashes or expose sensitive data.
- **STD-004-CPP – Prevent SQL Injection**: Ensures input validation and parameterized queries to prevent attackers from exploiting databases.
- **STD-005-CPP – Properly deallocate dynamically allocated resources:** Prevents memory leaks and ensures efficient memory management.
- **STD-010-CPP - Do not access free memory:** Accessing freed memory can result in undefined behavior or data breaches.

Low priority (Low Severity / Moderate Probability)

- **STD-009-CPP – Do not modify objects with a temporary lifetime:** This prevents attempts to modify temporary objects which may no longer exist, thereby avoiding undefined behavior.
- **STD-007-CPP – Handle all exceptions:** Ensures exceptions are handled properly to prevent system issues.

Unlikely (Low Severity / Low Probability)

- **STD-006-CPP – Use static assertions to test constant expressions:** Verifies conditions at compile time, reducing runtime errors.
- **STD-008-CPP – Use correct integer precisions:** Ensures appropriate integer types are used to avoid overflow or precision loss, helping to prevent exploitation.

10 PRINCIPLES

Validate Input Data	Heed Compiler Warnings	Architect and Design for Security Policies	Keep it Simple	Default Deny
• STD-001-CPP • STD-002-CPP • STD-003-CPP • STD-004-CPP-SQL • STD-006-CPP • INT35-008-CPP • EXP35-009-CPP	• STD-002-CPP • STD-003-CPP • STD-006-CPP • STD-007-CPP • INT35-008-CPP • EXP35-009-CPP • STD50-010-CPP	• STD-004-CPP-SQL	• STD-001-CPP • STD-002-CPP • STD-003-CPP • STD-005-CPP • STD-006-CPP • STD-007-CPP • INT35-008-CPP • EXP35-009-CPP • STD50-010-CPP	• STD-003-CPP

Adhere to the Principle of Least Privilege:	Sanitize Data Sent to Other Systems	Practice Defense in Depth:	Use Effective Quality Assurance Techniques	Adopt a Secure Coding Standard
• STD-004-CPP-SQL • STD-005-CPP	• STD-004-CPP-SQL • STD-003-CPP • STD-002-CPP • INT35-008-CPP • EXP35-009-CPP	• STD-002-CPP • STD-004-CPP-SQL • STD-005-CPP • STD-007-CPP • INT35-008-CPP • EXP35-009-CPP • STD50-010-CPP	• STD-004-CPP-SQL • STD-005-CPP • STD-007-CPP • STD50-010-CPP	• STD-001-CPP • INT35-008-CPP

CODING STANDARDS

1. **STD-001-CPP** – Avoid C-style variadic functions.
2. **STD-002-CPP** – Do not read uninitialized memory.
3. **STD-003-CPP** – Do not attempt to modify string literals.
4. **STD-004-CPP-SQL** – Prevent SQL injection.
5. **STD-005-CPP** – Properly deallocate dynamically allocated resources.
6. **STD-006-CPP** – Use static assertions to test constant expressions.
7. **STD-007-CPP** – Handle all exceptions.
8. **STD-008-CPP** – Use correct integer precisions.
9. **EXP35-009-CPP** – Do not modify objects with a temporary lifetime.
10. **STD50-010-CPP** – Do not access freed memory.

- My ranking system is based off the Ranking system adapted by the **National Vulnerability Database(NVD**) using the **Common Vulnerability Scoring System (CSS**) (Chapple, 2024).
- For example, vulnerabilities will be ranked between 0 – 10 depending on the severity of risk they impose.
- Higher scores will reflect critical threats like admin exploits, while a lesser score would be applied to a Denial-of-Service (DoS) attack.
- The Ranking system integrates with the coding standards in the security policy by aligning standards to the type of vulnerability. For instance, Validating Input will align with high-severity (8-10 score) vulnerabilities, i.e. SQL injection attacks.

ENCRYPTION POLICIES

Encryption at Rest

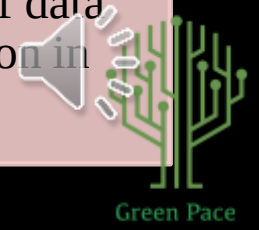
- Encryption at rest pertains to securing data that is stored on physical or digital devices. Data that is encrypted on storage devices cannot be read in plain text without access to proper decryption methods.
- Common examples that utilize encryption at rest include data stored in databases, cloud storage, hard drives, USB devices, and other storage media.

Encryption in Flight

- Encryption in flight, or encryption in transit protects data that is in transit across networks or internal systems. The internet is an example of data that is encrypted while in transit.
- Robust encryption protocols are often used to help protect data, i.e. Transport Layer Security (TLS) or HTTPS are often utilized.

Encryption in Use

- Encryption in use pertains to data that is actively being used, processed, or modified, either by an application or user.
- Encryption in use helps protect data that is actively being handled; such as, protecting sensitive customer data while it is being viewed or edited, financial data, or healthcare data. All are examples of data that rely on the encryption in use policy.



TRIPLE-A POLICIES

Authentication

- Implement a policy that mandates strong password rules along with enforcing multi-factor authentication.
- Utilize digital certificates to guarantee authenticity and identity verification.
- Remove unnecessary credentials that are no longer required.

Authorization

- Enforce a role-based access control policy that defines what each user role accesses and performs.
- Apply the principle of least privilege by limiting user accesses to only what is required.
- Validate permissions upon every request. This ensures that even minor requests are properly authenticated. (Authorization - OWASP Cheat Sheet Series, n.d.)
- Create unit test cases to validate authorization logic. Deploying custom logic for authorization helps detects flaws in access control logic.

Accounting

- Maintain a policy that requires detailed logging for all user activities, including but not limited to access times, any commands executed, programs run, etc.
- Implement into the policy regular log reviews to help identify for any unusual patterns from users.

Unit Testing

The coding vulnerability I chose to test is **Improper handling of collections and out-of-bounds access**.

- **Improper Handling of Collections:** This includes ensuring collections are properly set up and managed. For example:
  - Ensuring that a collection is empty upon creation
  - Clearing or resizing a collection to avoid incorrect data
- **Out-of-Bounds Access:** This happens when you try to access elements outside the valid range of a collection. For example:
  - Attempting to access an invalid index in an empty collection should throw an out_of_range exception to prevent crashes or security issues.

Is Collection Empty on Creation?

- Test Type: Positive
- Description: Verifies that the collection is empty upon initialization.
- Result: Passed – The collection size was zero when created, confirming it was initialized empty.
- How to Improve: Test different collection types, e.g., lists, deques.

```
// Test that a collection is empty when created.
TEST_F(CollectionTest, IsEmptyOnCreate)
{
    // is the collection empty?
    ASSERT_TRUE(collection->empty());

    // if empty, the size must be 0
    ASSERT_EQ(collection->size(), 0);
}
```

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

Can Add Multiple Entries?

- Test Type: Positive
- Description: Tests whether adding multiple entries to the collection correctly increases its size.
- Result: Passed – The collection size increased as expected, matching the number of added entries.
- How to Improve: Test larger data sets, like 1000 entries and validate integrity of entries.

```
// TODO: Create a test to verify adding five values to collection
TEST_F(CollectionTest, CanAddFiveValuesToVector)
{
    add_entries(5);

    ASSERT_FALSE(collection->empty());
    ASSERT_EQ(collection->size(), 5);
}
```

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

Does Resizing to Zero Work?

- Test Type: Positive
- Description: Checks if resizing the collection to zero effectively clears all elements.
- Result: Passed – Resizing the collection to zero resulted in a size of zero, clearing all elements.
- How to improve: Test with custom collections and validate collection is of size zero after resizing.

```
// TODO: Create a test to verify resizing decreases the collection to zero
TEST_F(CollectionTest, VerifyResizingDecreaseCollectionZero)
{
    add_entries(5);
    // resize the collection to 0 elements
    collection->resize(0);
    // verify that the collection size is 0
    ASSERT_EQ(collection->size(), 0);
}
```

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

Does Accessing Invalid Index Throw Exception?

- Test Type: Negative
- Description: Verifies that attempting to access an index beyond the collection size throws a std::out_of_range exception.
- Result: Passed – Accessing an invalid index raised the expected exception, ensuring proper error handling.
- How to improve: Test various edge cases, such as negative indices or extremely large indices.

```
// TODO: Create a 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, VerifyOutOfRangeException)
{
    add_entries(5);
    // verify that accessing an index beyond hte size of the collection throws an exception
    ASSERT_THROW((void)collection->at(6), std::out_of_range);
}
```

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

Does Clear Erase Collection?

- Test Type: Positive
- Description: Verifies that calling clear() removes all elements in the collection.
- Result: Passed – Calling clear() emptied the collection, confirming the operation works as intended.
- How to improve: Test with more complex data structures and validate that collection is cleared.

```
// TODO: Create a test to verify clear erases the collection
TEST_F(CollectionTest, VerifyClearErasesCollection)
{
    add_entries(5);
    // clear all ements in the collection
    collection->clear();
    // verify that the collection is empty
    ASSERT_TRUE(collection->empty());
}
```

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

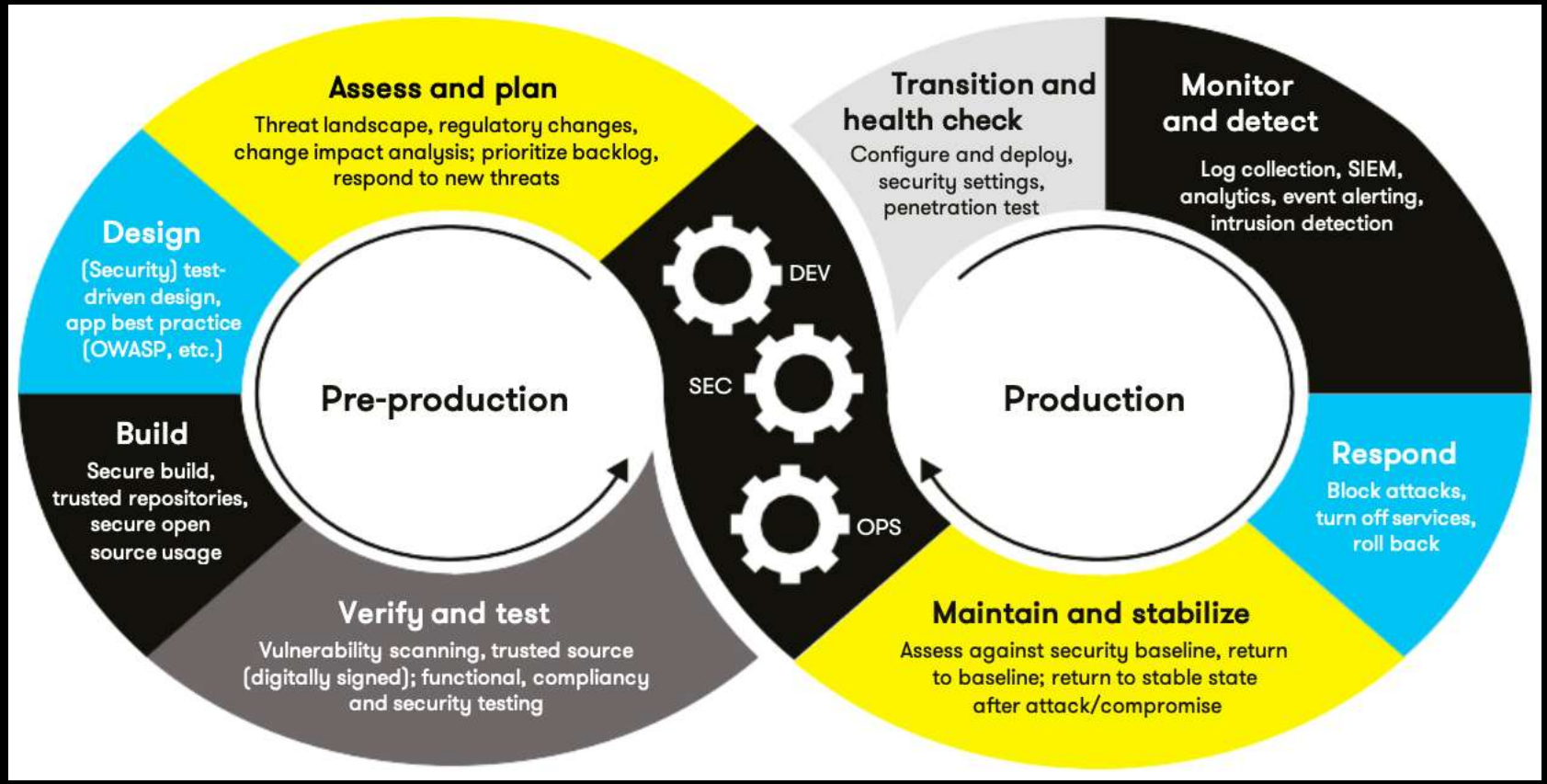
Does Accessing Empty Collection Throw Exception?

- Test Type: Negative
- Description: Ensures that accessing an element from an empty collection throws a std::out_of_range exception.
- Result: Passed – Attempting to access an empty collection triggered the expected exception, preventing undefined behavior.
- How to improve: Test additional methods like front() or back().

```
// TODO: Create 2 unit tests of your own to test something on the collection - do 1 positive & 1 negative
TEST_F(CollectionTest, ThrowOutOfRange)
{
    ASSERT_TRUE(collection->empty());
    // verify that accessing the front element of an empty collection throws an exception
    ASSERT_THROW((void)collection->at(0), std::out_of_range);
}
```

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

AUTOMATION SUMMARY



Green Pace

Automation Summary - Tools

Overview of the DevSecOps Pipeline:

- Integrates security into each phase of the DevOps lifecycle.
- Facilitates seamless collaboration between development and security.
- Emphasizes continuous integration, along with testing and monitoring.

External Tools used in the DevSecOps Pipeline

- Threat Modeling & Risk Analysis – Primarily used in the Assess and Plan phase to identify vulnerabilities early and prioritize security goals.
- IDE Plugins & Compilers– Applied during the Design & Build phase to automate secure coding checks and validate code integrity.
- SAST & DAST - Implemented in the Verify and Test phase to scan for vulnerabilities before production.
- Splunk & OpenSCAP: Provide real-time monitoring and compliance checks in the Monitor and Respond phase (GeeksforGeeks, 2024).
- SOAR Splunk Phantom: Automates incident response to quickly mitigate security breaches (GeeksforGeeks, 2024).

RISKS AND BENEFITS

Problems:

- Delayed security increases vulnerabilities and risks.
- Reactive strategies lead to poor security, data risks, and reputational damage.

Solutions:

- Implement a Shift-Left in Security mindset. (Yadav, 2024).
- Conduct security assessments.
- Automate security testing
- Monitor and measure security performance.
- Train development teams on secure coding practices(Sacolick, 2021).

Risks and Benefits:

- Risks:
  - Higher upfront costs
  - Potential delays during integration
- Benefits:
  - Reduces attack likelihood
  - Improves collaboration and communication
  - Faster Time-to-Market
  - Long-term cost savings(

Risks of Waiting:

- Increased attack likelihood
- Reactive strategies raise costs and reputational damage

Steps to Take:

- Shift-left security in DevOps pipeline
- Automate testing in DevSecOps
- Start early threat assessments

RECOMMENDATIONS

Current Gaps in Security Policy

- Employee Training
  - Train staff on phishing and sensitive data handling
- Vendor Risk Management
  - Assess third-party risks to reduce vulnerabilities
- Weak Identity and Access Management
  - Implement Multi-Factor Authentication and limit account privileges

Real World Scenario

- NotPetya Attack (2017)
  - Highlighted the need for vendor risk assessments and stronger identity controls(Email Breach Chronicles: The NotPetya Catastrophe - Global Havoc in 2017 | Zoho Workplace, 2023).
- Lessons Learned
  - Better MFA, least privilege access, and proactive training could have mitigated damages.
- Proactive Response:
  - Lack of an incident response plan delayed recovery.

CONCLUSIONS

Recommended Standards and Improvements

- Integrate security into every phase of DevSecOps
- Address gaps in threat monitoring and response
- Conduct security assessments before adopting new tools
- Leverage AI and machine learning for threat detection
- Perform regular audits to enhance security standards
- Focus on proactive improvements over reactive measures

REFERENCES

- Authorization - OWASP Cheat Sheet series. (n.d.). https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html
- Bowman, B. (2019, March 18). The AAA Framework for Identity Access Security - Security Boulevard. Security Boulevard. https://securityboulevard.com/2019/03/the-aaa-framework-for-identity-access-security/
- Email breach chronicles: The NotPetya catastrophe - global havoc in 2017 | Zoho Workplace. (2023, November 27). Zoho Workplace. https://www.zoho.com/workplace/articles/notpetya-cyberattack.html
- GeeksforGeeks. (2023b, March 20). Graylog vs Splunk. GeeksforGeeks. https://www.geeksforgeeks.org/graylog-vs-splunk/
- GeeksforGeeks. (2024, July 8). What is SOAR (Security Orchestration, Automation and Response) ? GeeksforGeeks. https://www.geeksforgeeks.org/what-is-soar/?ref=header_outind
- Cooper, S., & Cooper, S. (2024, March 22). The best DAST tools for 2024. Comparitech. https://www.comparitech.com/net-admin/dast-tools/
- Sacolick, I. (2021, March 8). 6 security risks in software development and how to address them. InfoWorld. https://www.infoworld.com/article/2262633/6-security-risks-in-software-development-and-how-to-address-them.html
- Vaghela, E. (2024, November 13). How Automation is Transforming DevSecOps for Better Security. DEV Community. https://dev.to/enna/how-automation-is-transforming-devsecops-for-better-security-1co
- Yadav, N. (2024, December 9). How DevSecOps enables a Shift-Left Security approach & Testing. SquareOps. https://squareops.com/blog/how-devsecops-enables-a-shift-left-approach-in-security/

Green Pace