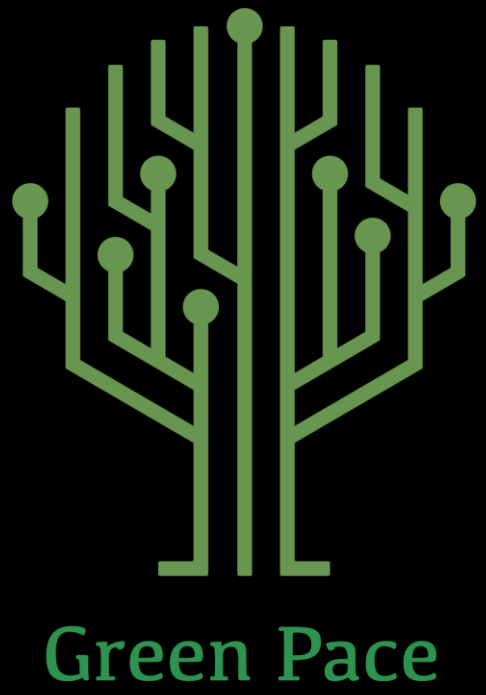
Green Pace

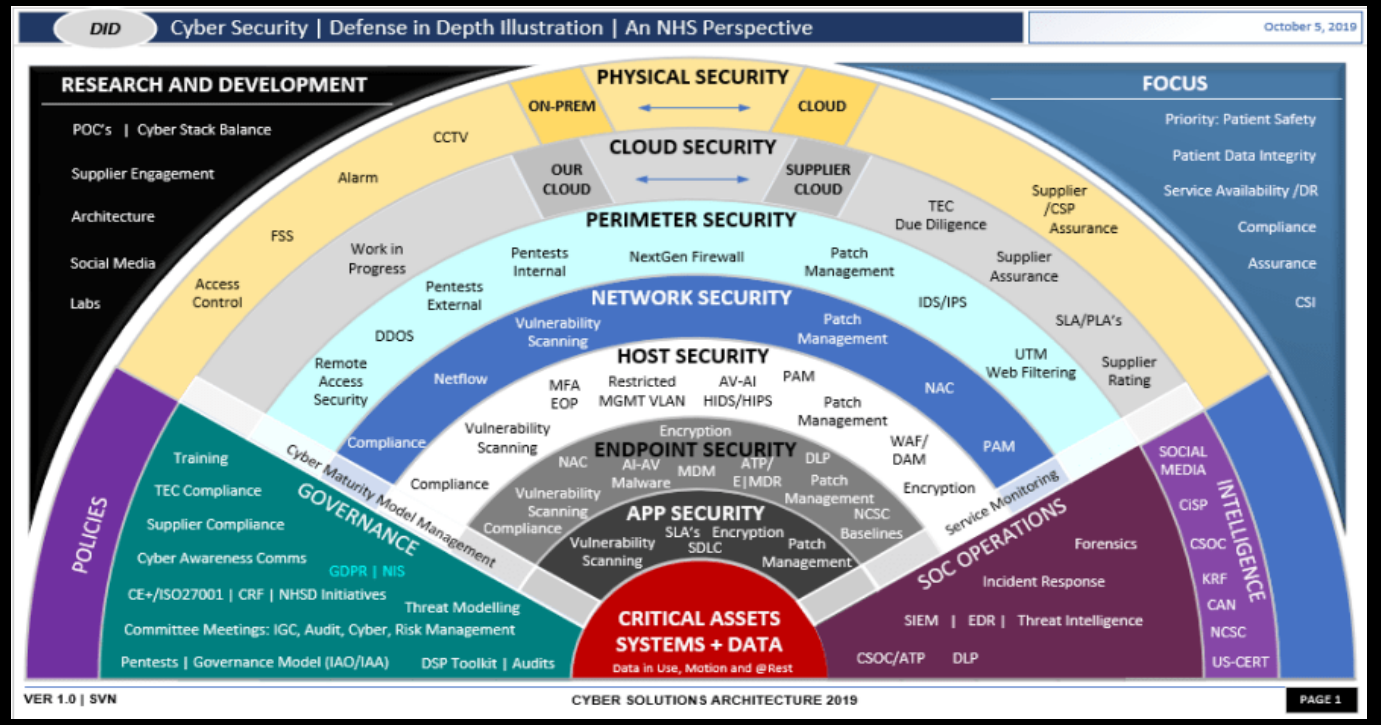
Security Policy Presentation

Developer: *Michelle Powers*



OVERVIEW: DEFENSE IN DEPTH

This policy's overall purpose is to outline security standards that Green Pace employees can incorporate during software development, allowing for consistent design while protecting both company and end user data.



Green Pace

THREATS MATRIX

Threats are categorized by coding standard number. Each category is color-coded by potential impact, from low to high: green, yellow, orange, red.

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

10 PRINCIPLES

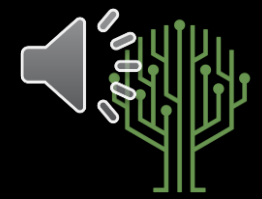
Validate Input Data: *STD-004-JAV, STD-007-CPP*
Heed Compiler Warnings: *STD-001-CPP, STD-002-CPP, STD-010-CPP*
Architect and Design for Security Policies: *STD-009-CPP, STD-010-CPP*
Keep It Simple: *STD-003-CPP, STD-005-CPP*
Default Deny: *STD-008-CPP*
Adhere to the Principle of Least Privilege: *STD-008-CPP*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

CODING STANDARDS

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
STD-003-CPP	High	Likely	Medium	P18	L1
STD-004-JAV	High	Likely	Medium	P18	L1
STD-005-CPP	High	Likely	Medium	P18	L1
STD-007-CPP	High	Likely	High	P9	L2
STD-006-JAV	Medium	Probable	Medium	P8	L2
STD-010-CPP	Medium	Probable	Medium	P8	L2
STD-002-CPP	High	Probable	High	P6	L2
STD-009-CPP	Medium	Unlikely	Low	P6	L2
STD-008-CPP	Medium	Unlikely	Medium	P4	L3
STD-001-CPP	Low	Unlikely	Low	P3	L3

ENCRYPTION POLICIES

Encryption at rest: All data without exception should be encrypted before and after its access. This works as a safeguard against data breaches, which could potentially expose sensitive user data or propriety company information.
Encryption in flight: Data should always be encrypted during transit by both the sender and receiver – never sent as plaintext. This helps to protect against Man-in-the-Middle (MITM) attacks, or other interception techniques.
Encryption in use: Both of the above policies should be used here as well, and files being currently read or written to should have the data encrypted and decrypted in real time. This policy helps to specifically prevent active threats such as SQL injection, or above-mentioned MITM attacks.

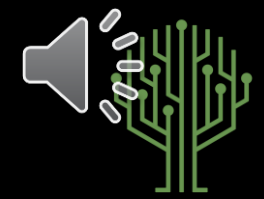


TRIPLE-A POLICIES

Authentication: The main way employees are identified, and their permissions and level of access are determined by their unique identifier(s) such as a username and password. Their username identifier is also used for system logging purposes.

Authorization: By default, new users have the lowest level of access which is expanded upon solely based on absolute necessity, and this access is re-evaluated when job status changes.

Accounting: Any changes to either local files or databases are automatically logged with the time, date, and username of the employee who made the change.



Unit Test 1: Does a new vector contain data?

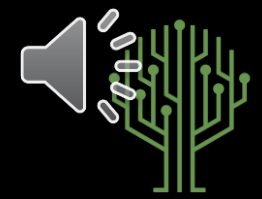
Purpose: To check for memory tampering by ensuring that a newly created vector is empty as intended.

Result:

```
[ RUN      ]
[       OK ]
```

```
// 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);
}
```



Green Pace

Unit Test 2: Has overflow occurred after data input?

Purpose: To check for possible buffer overflow of newly inserted data.

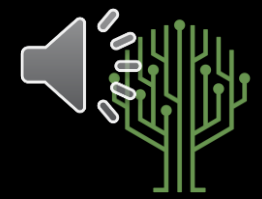
Result:

```
[ RUN      ]
[       OK ]
```

```cpp
// Test to verify an overflow occurs after inserting a value over the signed integer limit
// NOTE: This is a negative test
TEST_F(CollectionTest, CheckVectorValueOverflow)
{
    // is the collection empty?
    EXPECT_TRUE(collection->empty());
    // if empty, the size must be 0
    EXPECT_EQ(collection->size(), 0);

    // Add a new very high int value, using push_back
    collection->push_back(3000000000);

    // is the result negative? (overflow occurred)
    ASSERT_LT(collection->at(0), 0);
}
```



Unit Test 3: Is capacity appropriately increased?

Purpose: To check that the capacity is increased as intended after adding several new entries.

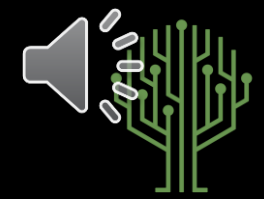
Result:

```
[ RUN      ]
[       OK ]
```

```
// Test to verify that capacity is greater than or equal to size for 0, 1, 5, 10 entries
TEST_F(CollectionTest, CheckVectorCapacity)
{
    // is the collection empty?
    EXPECT_TRUE(collection->empty());
    // if empty, the size must be 0
    EXPECT_EQ(collection->size(), 0);

    // Add 15 entries - above the maximum of 10
    add_entries(15);

    // is the capacity >= 0?
    ASSERT_GE(collection->capacity(), 0);
    // is the capacity >= 1?
    ASSERT_GE(collection->capacity(), 1);
    // is the capacity >= 5?
    ASSERT_GE(collection->capacity(), 5);
    // is the capacity >= 10?
    ASSERT_GE(collection->capacity(), 10);
}
```



Unit Test 4: Has data been properly added?

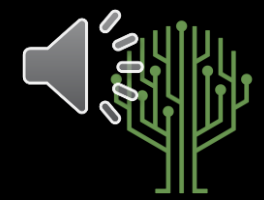
Purpose: To ensure that data corruption did not occur, by verifying a value and its location.

Result:

```
[ RUN      ]
[       OK ]
```

```cpp
// Test to verify that a specific value is added at a specific location
TEST_F(CollectionTest, InsertValueAtPosition)
{
    // Add 5 entries
    add_entries(5);
    // Replace the value at index 3 with 15
    collection->emplace(collection->begin() + 3, 15);

   // is the value at position 3 now 15, as expected?
   ASSERT_EQ(collection->at(3), 15);
}
```



Green Pace

Unit Test 5: Is an out of bounds exception properly thrown?

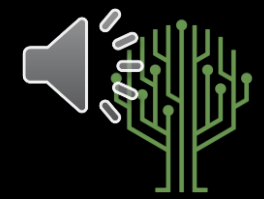
Purpose: To ensure that undefined behavior does not occur, by throwing an exception if data or objects are not accessed outside of their bounds or lifetime.

Result:

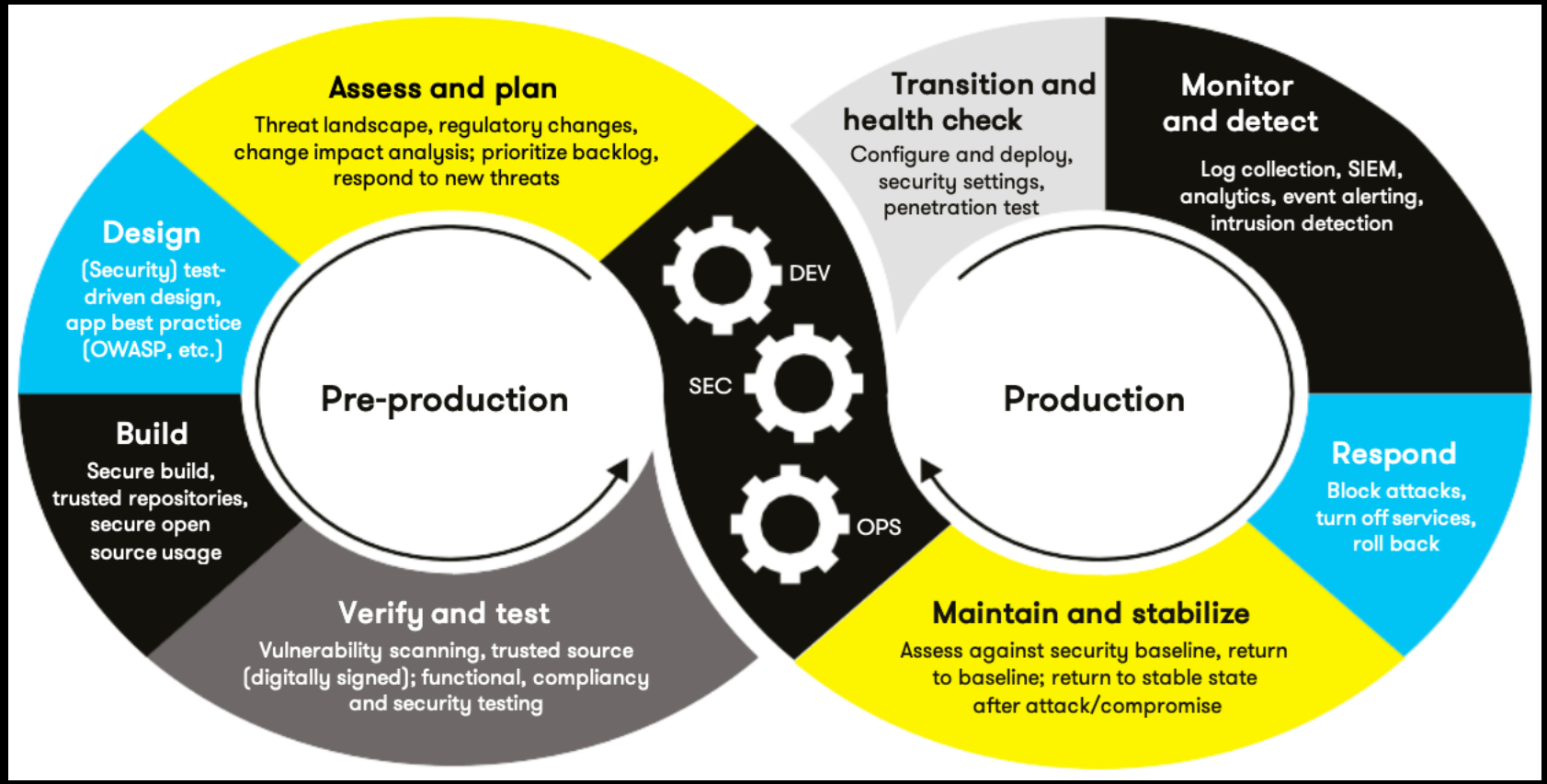
```
[ RUN      ]
[       OK ]
```

```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, CheckOutOfRange)
{
    // Add a single entry
    add_entries(1);

    // Check index 5, which should be out of bounds
    ASSERT_THROW(collection->at(5), std::out_of_range);
}
```



AUTOMATION SUMMARY



Green Pace

TOOLS

Pre-Production:

- Compile, and fix all errors or warnings found regardless of severity.

Post-build:

- A project is run through all static analysis tools cumulatively used in the "Automation" sections of the policy's coding standards; examples include Parasoft C/C++ (https://www.parasoft.com/products/parasoft-c-ctest/) or CodeSonar (https://codesecure.com/our-products/codesonar/). These can help catch vulnerabilities missed by the compiler.

Production:

- For very high-traffic applications, it may be beneficial to incorporate tools such as LogicMonitor (https://www.logicmonitor.com/) or ELK Stack (https://www.elastic.co/elastic-stack) which can parse large log files and allow for quicker threat response than would happen through manually monitored log analysis.
- In the case that a threat does occur, automatic rollbacks could also be performed after the vulnerability has been detected.

RISKS AND BENEFITS

Example 1: In cases of **malware/ransomware attacks**, waiting to act could cause damages in proportion to time. As seen in the WannaCry attack (Whittaker, 2019), they can get out of hand very quickly.
- The benefits to quick action include lower remediation cost and less widespread access for the attacker if a data breach occurred.

Example 2: **Phishing attempts and social engineering** can prove extremely disastrous if not handled promptly as well (Fortra, 2024); the Triple-A policies are meant to protect employees from these types of attacks, but they are not foolproof.
- Acting quickly can allow the policy to be better streamlined, as there no specific guidelines in place yet which address phishing attempts.

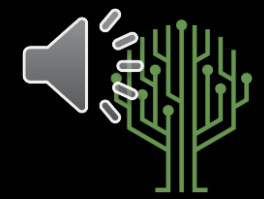
RECOMMENDATIONS

Security Principles and Standards:

- Additional system and coding standards could be added on top of the current ten.
- Procedures and standards for DDoS (Distributed Denial-of-Service) could be added.

DevSecOps:

- Vulnerability testing could possibly be moved up to the *build* phase. Doing so would give more flexibility in changing the early design if necessary and allowing for close monitoring before moving onto production.
- Within production, log collection could additionally be automated as part of the health check, immediately after deployment.

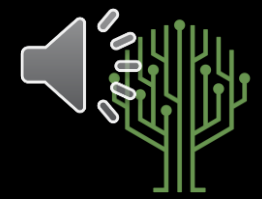


CONCLUSIONS

The policy is overall very well-defined and thorough, but it could be expanded to cover certain topics in more in-depth ways.

- Protocols could be added to address potential inter-company attacks ("Spy Hackers"), or other types of attackers with alternative motivations (McAfee, 2011).

- Guidelines and standards for unit testing, as these are currently largely left up to developer discretion.



REFERENCES

- Whittaker, Z. (2019, May 12). *Two years after WannaCry, a million computers remain at risk*. TechCrunch. https://techcrunch.com/2019/05/12/wannacry-two-years-on/
- Fortra. (2024, November 29). *9 Examples of Social Engineering Attacks*. Terranova Security. https://www.terranovasecurity.com/blog/examples-of-social-engineering-attacks/
- McAfee. (2011, March 16). *7 Types of Hacker Motivations. McAfee security*. https://web.archive.org/web/20210413134404/https://www.mcafee.com/blogs/consumer/family-safety/7-types-of-hacker-motivations/