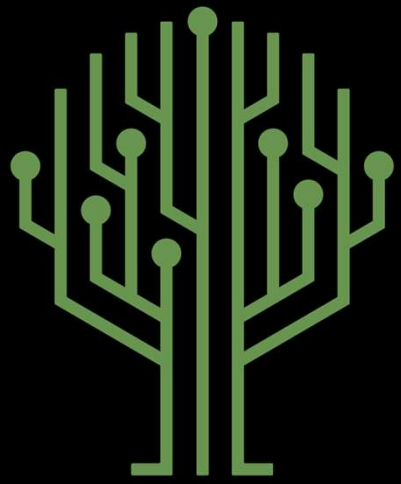
Green Pace

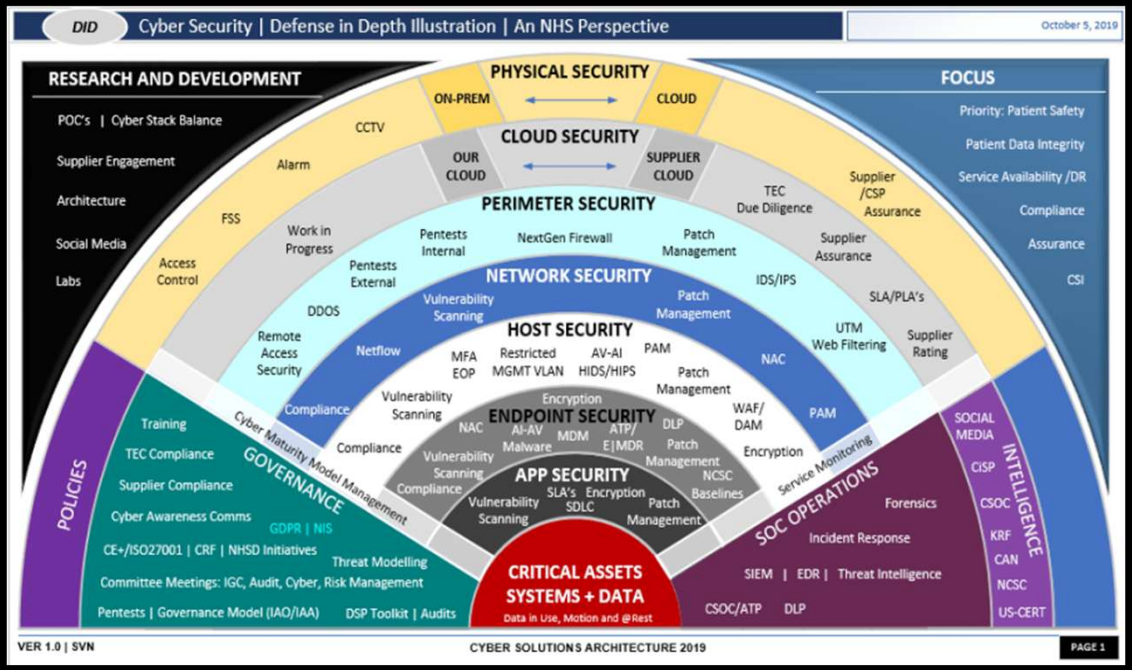
Security Policy Presentation

Developer: *Ryan Branchaud*



OVERVIEW: DEFENSE IN DEPTH

The primary purpose of this security policy is to defend our systems and data from potential threats and vulnerabilities. In our increasingly interconnected and complex digital landscape, it's essential to establish a full-bodied framework that addresses the many risks we face. This policy implements security measures at various stages of the development cycle which supports the defense-in-depth strategy by adding multiple layers of defense.



Green Pace

THREATS MATRIX

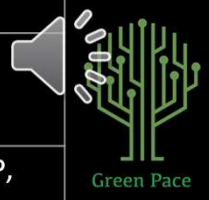
10 CPP Standards

Ordered by Likelihood and Priority

Likely Standards: 003, 004, 005, 008	Priority P18
Low Priority P1	Unlikely Standards: 006

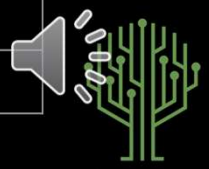
10 PRINCIPLES

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



CODING STANDARDS

Standard	Rule	Severity	Likelihood	Remediation Cost	Priority	Level
String Correctness	STD-003-CPP	High	Likely	Medium	P18	L1
SQL Injection	STD-004-CPP	High	Likely	Medium	P18	L1
Memory Protection	STD-005-CPP	High	Likely	Medium	P18	L1
Memory Protection	STD-008-CPP	High	Likely	Medium	P18	L1
String Correctness	STD-010-CPP	High	Unlikely	Medium	P6	L2
Exceptions	STD-007-CPP	Low	Probable	Medium	P4	L3
Exceptions	STD-009-CPP	Low	Probable	Medium	P4	L3
Data Type	STD-001-CPP	Low	Unlikely	Low	P3	L3
Data Value	STD-002-CPP	Low	Unlikely	Low	P3	L3
Assertions	STD-006-CPP	Low	Unlikely	High	P1	L3



ENCRYPTION POLICIES

Encryption at Rest	Encryption at rest is a security measure and refers to protecting data that is in a stored state, such as on a hard drive, SSD, etc.
Encryption in Flight	Encryption in flight is another security measure and refers to protecting data that is being transmitted from one location to another.
Encryption in Use	Encryption in use is the third encryption security measure and refers to protecting data while it is in use or actively being processed.

TRIPLE-A POLICIES

Authentication	Authentication is the first process of the Triple-A Framework, and it refers to verifying the identity of a user or device.
Authorization	Authorization is the second process of the Triple-A Framework, and it refers to determining what authenticated users are allowed to do.
Accounting	Accounting is the third process that completes the Triple-A Framework, and it refers to tracking and logging all activities performed by users within the network.

Unit Test 1

STD-001-CPP: Never qualify a reference type with const or volatile.

```
void funcOne(const int& ref) {
}

void funcTwo(volatile int& ref) {
}

void funcThree(int& ref) {
}

int main() {

    // Value for function testing
    int val = 88;

    // Failure function testing
    assert((funcOne(val), false) && "Function One should not accept a const reference.");
    assert((funcTwo(val), false) && "Function Two should not accept a volatile reference.");

    // Passing function testing
    funcThree(val);

    return 0;
}
```

Unit Test 2

STD-002-CPP: Do not declare or define a reserved identifier.

```
void checkReservedId(const std::string& identifier) {

    std::regex reservedId1(R"(_[A-Z])");
    std::regex reservedId2(R"(__)");
    assert(!std::regex_search(identifier, reservedId1) && !std::regex_search(identifier, reservedId2) && "Detected: Reserved Id");
}

int main() {

    // Failure testing
    checkReservedId("_ReservedId");
    checkReservedId("__reserved_id");

    // Passing testing
    checkReservedId("validIdentifier");

    return 0;
}
```

Unit Test 3

STD-003-CPP: Do not attempt to create a std::string from a null pointer.

```cpp
void checkString(const char* str) {
    assert(str != nullptr && "Cannot create string from nullptr.");
    std::string s(str);
}

int main() {
    try {
        checkString(nullptr);
    }

    // Failure testing
    catch (const std::exception& e) {
    }

    // Passing testing
    checkString("Test passed!");

    return 0;
}
```

Unit Test 4

STD-004-CPP: Guarantee that storage for strings has sufficient space for character data and the null terminator.

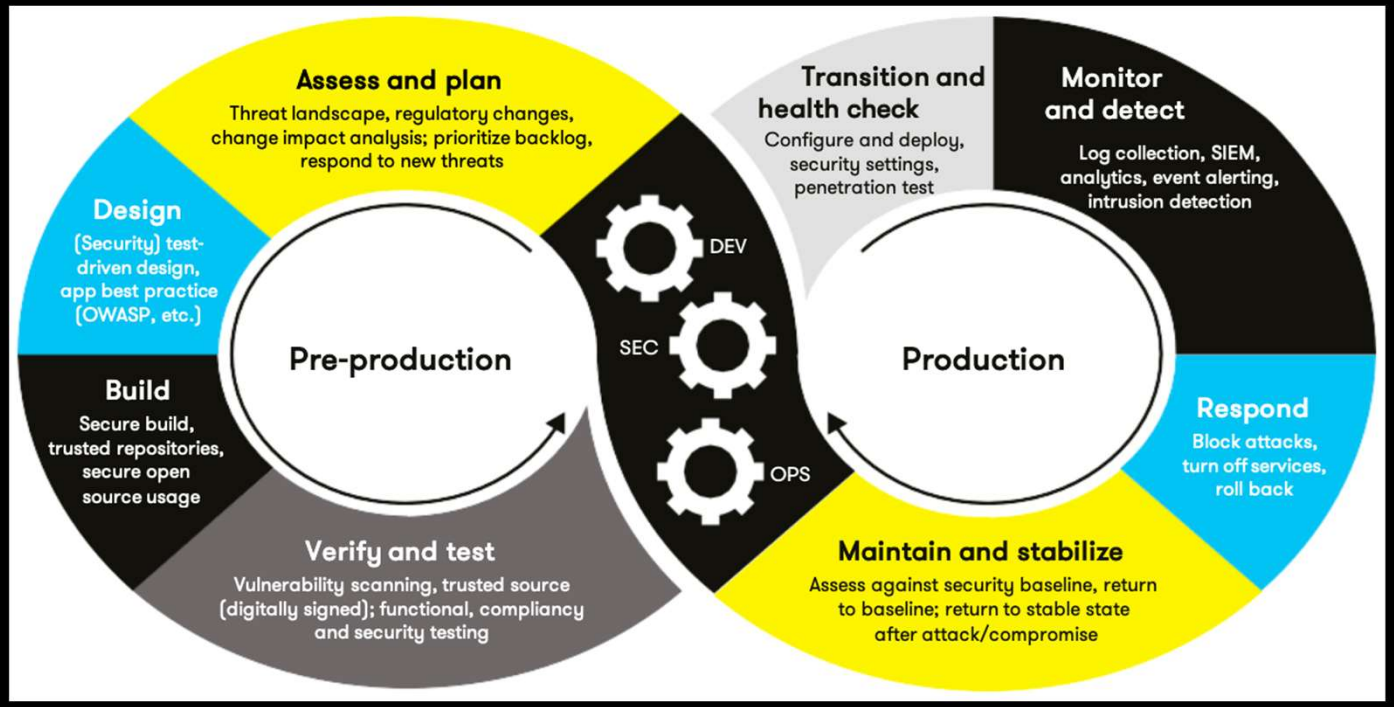
```
void checkAllocation(const char* source, size_t bufferSize) {
    assert(bufferSize > strlen(source) && "Insufficient space!");
    char* buffer = new char[bufferSize];
    strcpy(buffer, source);
    delete[] buffer;
}

int main() {

    // Failure testing
    checkAllocation("There's not enough space here!", 5);

    // Passing testing
    checkAllocation("There's sufficient space for the null terminator here!", 6);

    return 0;
}
```

Unit Test 5

STD-005-CPP: Do not access freed memory.

```
void checkFreedMem() {
    int* ptr = new int(88);
    delete ptr;

    // Failure testing if ptr is not reset
    assert(ptr != nullptr && "Warning: Freed Memory is being accessed.");

    // Pointer reset to prevent future issues
    ptr = nullptr;
}

int main() {

    // Catches the error
    checkFreedMem();
    return 0;
}
```

AUTOMATION SUMMARY



Green Pace

TOOLS

Notable external tools utilized in the DevSecOps Pipeline include:

- CPPCheck – identifies bugs, undefined behavior, and violations during coding and building stages.
- SonarQube – identifies bugs and vulnerabilities across multiple languages during coding, build, and testing stages.
- OWASP – provides tools and guidelines to perform security assessments and ensure compliance during testing and deployment stages.
- Parasoft – provides testing software tools that automates statis analysis, unit testing, and code coverage during coding, build, and testing stages.
- Coverity – is a statis analysis tool that will identify vulnerabilities and defects by automatically scanning our code during the build stage.
- Jenkins – is an automation server that facilitates continuous integration and continuous delivery throughout the entire pipeline.
- Gitlab – manages our code repositories, automate continuous integration and continuous delivery pipelines, and integrate our security tools throughout the entire pipeline.

RISKS AND BENEFITS

Risks of Waiting: Harm to the brand, customer trust, financial costs, loss or harm of data.

Benefits of Early Action: Ensure customer trust and brand respect, mitigate threats, reduce overhead testing, protection of data.

“There are two very important reasons to incorporate security principles and design early and throughout the development lifecycle-cost and efficacy. Security is often not considered until a product is essentially ready for release. At that point, most of the design and engineering decisions are pretty much set in stone and it would be very costly to try and go back and fix security issues that are baked into the core of the product.” (Bradley, 2021)

RECOMMENDATIONS

1. Conduct regular security gap assessments and audits. This is a proactive approach to help mitigate risks and ensure industry regulation compliance.
2. Perform regular code reviews and audits. Another proactive approach to help mitigate risks early.
3. Create a culture of security awareness. Provide regular training and resources to help everyone understand the importance of this policy.
4. Ensure a feedback loop is established. Continuous improvement is essential to security. Collecting feedback from developers, teams, stakeholders, or anyone involved with this policy will be a vital system to ensure prolonged protection.

CONCLUSIONS

Our existing policy is a strong start, but it is not without its gaps. As mentioned earlier, our current 10 standards don't touch on every one of the 10 principles of security. I strongly suggest we round out our policy by adding the following standards:

- Architect and Design of Security Principles, I'd suggest adding DCL50-CPP: Do not define a C-style variadic function.
- Default Deny, I'd suggest adding MSC51-CPP: Ensure your random number generator is properly seeded.
- Adhere to the Principle of Least Privilege, I'd suggest adding MEM51-CPP: Properly deallocate dynamically allocated resources.

REFERENCES

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

- Magnusson, A. (2023, October 5). *What is AAA Security? Authentication, Authorization, and Accounting*. Discover.strongdm.com. https://www.strongdm.com/blog/aaa-security

- Bradley, T. (2021, January 20). *Better Security through the Security Development Lifecycle*. TechSpective. https://techspective.net/2021/01/19/better-security-through-the-security-development-lifecycle/