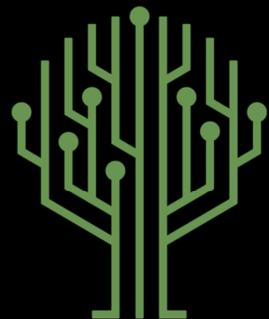


Green Pace

Security Policy Presentation
Developer: Dalton Rose

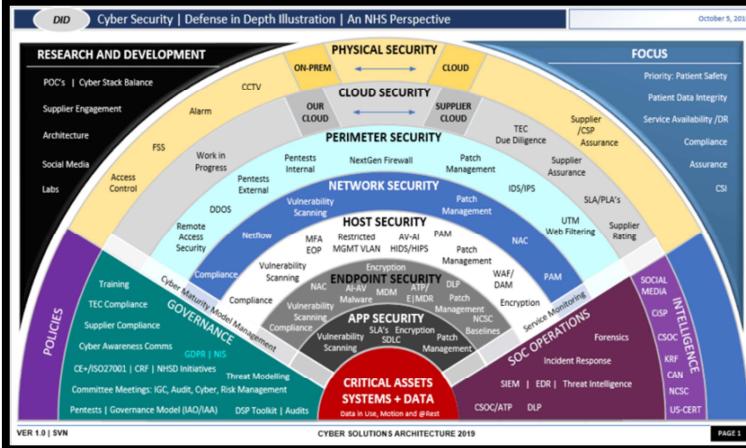


Green Pace

Hello, my name is Dalton Rose and today I'll be proposing a security policy.

OVERVIEW: DEFENSE IN DEPTH

Defense In Depth – a security strategy where multiple layers of security measures are used to secure information systems.



This security policy outlines the security practices at Green Pace including secure coding principles, secure coding standards, encryption policies, automation, and more. By following this policy, we can ensure the security of our software. Each of these elements work together to provide defense in depth.



Defense in depth refers to a security strategy where multiple layers of security measures are used to secure information systems.

This security policy is an example of defense in depth, and we'll cover security practices like secure coding principles, secure coding standards, encryption policies, automation, and more.

THREATS MATRIX

Likely	Priority
INT-002-CPP STR-001-CPP STR-002-CPP	ERR-001-CPP INT-001-CPP
Low priority	Unlikely
MEM-001-CPP EXP-001-CPP ERR-002-CPP	DCL-001-CPP MSC-001-CPP



In this Threats Matrix, you'll see the coding standards I'll outline divided by their likeliness to occur and how high priority they are. We have coding standards that are likely and unlikely to occur and some that are particularly high or low priority.

10 PRINCIPLES

- **Validate Input Data** – Perform input validation to verify user-entered data before the data is used. *Relevant Coding Standards: INT-002-CPP, STR-001-CPP*
- **Heed Compiler Warnings** – Compiler warnings can catch many potential vulnerabilities. *Relevant Coding Standards: DCL-001-CPP, EXP-001-CPP*
- **Architect and Design for Security Policies** – By incorporating security into the design process, we place importance on security. *Relevant Coding Standards: MSC-001-CPP, EXP-001-CPP*
- **Keep It Simple** – Simple code is easier to develop and maintain. *Relevant Coding Standards: ERR-001-CPP, ERR-002-CPP*
- **Default Deny** – By default, access and privileges should only be granted when necessary. *Relevant Coding Standards: STR-002-CPP*
- **Adhere to the Principle of Least Privilege** – Only the lowest level of privilege necessary should be granted. *Relevant Coding Standards: STR-002-CPP*
- **Sanitize Data Sent to Other Systems** – When sending data to other systems, data should always be sanitized to ensure the data is safe and properly formatted. *Relevant Coding Standards: STR-002-CPP*
- **Practice Defense in Depth** – Use multiple complementary layers of security. *Relevant Coding Standards: STR-002-CPP, MSC-001-CPP*
- **Use Effective Quality Assurance Techniques** – Using proper quality assurance techniques and tools helps identify defects. *Relevant Coding Standards: INT-001-CPP, STR-001-CPP, MEM-001-CPP, ERR-001-CPP, ERR-002-CPP, MSC-001-CPP, DCL-001-CPP, EXP-001-CPP*
- **Adopt a Secure Coding Standard** – By following a secure coding standard, we have formalized guidelines to follow that help ensure the security of our software. *Relevant Coding Standards: MEM-001-CPP, ERR-001-CPP, ERR-002-CPP*



Green Pace

Here are the ten principles of secure coding and their definitions. You'll also see which of the identified coding standards are most relevant to each secure coding principle.

CODING STANDARDS

Standard	Severity	Likelihood	Remediation Cost	Priority	Level
INT-001-CPP	Medium	Unlikely	Low	High	L3
ERR-001-CPP	Medium	Unlikely	Medium	High	L3
MSC-001-CPP	Medium	Unlikely	Low	High	L3
INT-002-CPP	High	Likely	Medium	Medium	L2
STR-001-CPP	High	Likely	Medium	Medium	L2
STR-002-CPP	High	Likely	Medium	Medium	L2
ERR-002-CPP	Low	Likely	Medium	Medium	L2
DCL-001-CPP	High	Unlikely	Medium	Medium	L2
MEM-001-CPP	High	Likely	Low	Low	L1
EXP-001-CPP	High	Likely	Medium	Low	L1



In this table, you'll see the ten coding standards I've identified ordered by their priority. Their priority was determined by considering factors like severity, likelihood to occur, and remediation cost.

ENCRYPTION POLICIES

- Encryption at Rest:
 - Protects data stored on disk
 - Example: data backups or databases
 - Use a secure encryption algorithm like AES
- Encryption in Flight:
 - Protects data being transmitted
 - Example: data being sent through API calls or to the client
 - Use TLS/SSL to encrypt data in flight
- Encryption in Use:
 - Protects data being actively used
 - Examples: any data stored or processed in memory
 - Use techniques like homomorphic encryption or secure multi-party computation



There are three types of data encryption we need to focus on.

First, encryption at rest. This refers to encrypting data that is stored on disk. Some examples of this are data backups and databases. To secure this data, we should use a secure encryption algorithm like AES.

Secondly, encryption in flight. This refers to encrypting data that is being transmitted. Some examples of this is any data being rendered to the client or data being sent through API calls. To secure this data, we must ensure we are using TLS/SSL.

Finally, encryption in use. This refers to encrypting data that is actively being used. An example of this is data being stored or processed in memory. We can use techniques like homomorphic encryption or secure multi-party computation to secure this data.

TRIPLE-A POLICIES

- Authentication
 - Process of verifying the identity of a user/device
 - Utilize login credentials, multi-factor authentication, and certificates to verify users/devices prevent unauthorized access
- Authorization
 - Process of checking which actions a user is allowed to take
 - Utilize role or policy-based access controls and follow the principle of least privilege to prevent unauthorized actions
- Accounting
 - Process of logging user and system activity
 - Log changes/creation of user account, changes to the database, files accessed or modified by users, user logins, and more.
 - Detailed logging plays an important role in identifying and investigating security threats



Triple-A refers to Authentication, Authorization, and Accounting.

Authentication refers to the process of verifying the identity of a user/device. We can utilize login credentials, multi-factor authentication, and certificates to verify users and devices and prevent unauthorized access.

Authorization refers to the process of checking which actions a user can perform. We can utilize role-based or policy-based access control and follow the principle of least privilege to limit users to only accessing what is strictly necessary.

Accounting refers to the process of logging user and system activity. We can log activity such as changes to user accounts, database changes, files accessed, user logins, and more. Keeping detailed logs is important for identifying and investigating security threats.

Unit Testing

```
#include "pch.h"
#include <iostream>

enum EnumType { First = 0, Second = 1, Third = 2 };

bool to_enum(int i, EnumType& out) {
    if (i < 0 || i > 2) {
        return false;
    }
    out = static_cast<EnumType>(i);
    return true;
}

TEST(EnumConvertTest, AcceptsLowerBound) {
    EnumType e;
    ASSERT_TRUE(to_enum(0, e));
    ASSERT_EQ(e, First);
}

TEST(EnumConvertTest, AcceptsUpperBound) {
    EnumType e;
    ASSERT_TRUE(to_enum(2, e));
    ASSERT_EQ(e, Third);
}

TEST(EnumConvertTest, RejectsBelowRange) {
    EnumType e;
    ASSERT_FALSE(to_enum(-1, e));
}

TEST(EnumConvertTest, RejectsAboveRange) {
    EnumType e;
    ASSERT_FALSE(to_enum(3, e));
}
```

The screenshot shows the Microsoft Visual Studio Debug window. It displays the output of a test run for the file 'gtest_main.cc'. The output shows four tests being run under the category 'EnumConvertTest'. The tests are: 'AcceptsLowerBound', 'AcceptsUpperBound', 'RejectsBelowRange', and 'RejectsAboveRange'. All four tests pass, indicated by the 'OK' status. The total time for the test run is 1 ms.

```
Running main() from D:\_work\1\s\googletest\googletest\src\gtest_main.cc
[==========] Global test environment set-up.
[ RUN      ] 4 tests from EnumConvertTest
[       OK  ] EnumConvertTest.AcceptsLowerBound (0 ns)
[       OK  ] EnumConvertTest.AcceptsUpperBound (0 ns)
[       OK  ] EnumConvertTest.RejectsBelowRange (0 ns)
[       OK  ] EnumConvertTest.RejectsAboveRange (0 ns)
[==========] 4 tests from EnumConvertTest (1 ms total)

[==========] Global test environment tear-down
[       OK  ] 4 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 4 tests.
```

[Identify the coding vulnerability you chose to test. Include four to six mixed tests for positive and negative results. Include a slide for each test. Use the

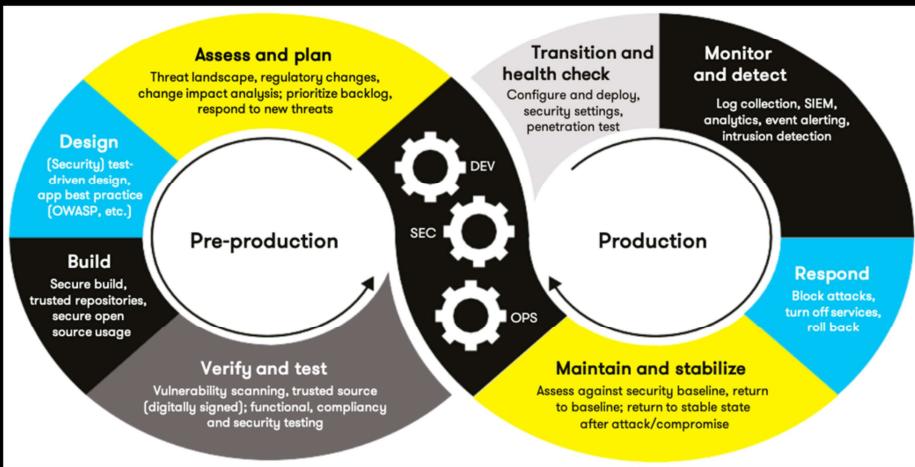


As an example, I created a simple function which converts an integer into an `EnumType` which includes logic to check whether the enum value is in range.

There are also four unit tests: two positive tests that test that an integer is successfully converted within range and two negative tests that test that when the conversion is performed with an out of range value, the conversion returns false.

As you can see, all tests pass successfully due to the error handling in the `EnumType` conversion function.

AUTOMATION SUMMARY



Here is a diagram showing the DevSecOps pipeline. Security automation tools are present throughout each phase of the DevSecOps pipeline. In the next slide, you'll see how automated security tools are used in each phase of the DevSecOps pipeline.

TOOLS

- Assess and Plan / Design: Automated threat modeling tools
- Build: Compiler, Dependency Scanning Tools, Static Application Security Testing Tools
- Verify and Test: Unit testing, Dynamic Application Security Testing tools, Interactive Application Security Testing tools
- Transition and Health Check: Automated configuration, deployment verification testing
- Monitor and Detect: Automated threat detection, logging, and alerting tools.
- Respond: Automated IP blocking, account locking
- Maintain and Stabilize: Continuous vulnerability scanning



In the Assess and Plan and Design phases, we will use automated threat modeling tools to identify potential threats before code is written.

In the Build phase, we use the compiler and dependency scanning tools to ensure secure code compilation and discover potential vulnerabilities in our dependencies.

In the Verify and Test phase, automation tools will be used to look for violations of the coding standards, verify required security functionality, and scan for other potential vulnerabilities.

In the Transition and Health Check phase, we will utilize automated tools to manage configurations and deploy our code to ensure our production environments are securely configured.

In the Monitor and Detect phase, we will utilize automated threat detection, logging, and alerting tools to monitor our systems for potential threats.

In the Respond phase, we will utilize automated tools to take defensive actions like revoking credentials if unauthorized access is suspected.

In the Maintain and Stabilize phase, we will continue to use automated tools to scan for potential vulnerabilities.

RISKS AND BENEFITS

- Problems:
 - No formalized security policy
 - Lack of established secure coding practices
 - Failure to take advantage of security automation
- Solution:
 - Implement a formalized security policy
 - Establish secure coding practices
 - Make full use of security automation
- Why Act Now?
 - Security plays a role in every phase of development
 - Risk of leaving potential vulnerabilities open



Currently, we have a few problems relating to the security of our software. Firstly, we do not have a formalized security policy outlining our security best practices. Secondly, we have not established secure coding standards. Finally, we are not fully utilizing security automation tools to secure our software.

To solve this, I propose implementing a formalized security policy outlining our security best practices. This security policy will also establish and detail the secure coding standards we will follow going forward. Additionally, we should take full advantage of security automation tools.

Acting quickly is important as leaving potential vulnerabilities open is dangerous. It could result in financial or reputational damages to our company. We may also be putting our user's data at risk. Additionally, security should not be an afterthought but something we practice throughout every phase of development.

RECOMMENDATIONS & Conclusion

- Limitations of Existing Policy:
 - Not Formalized
 - Insufficient Documentation
 - Security Separated from Development
- Future:
 - Evolving set of guidelines that guide secure software development
 - Better integrate security and development
 - Security Automation Tools
 - Review & Revise
 - Continuous Training



Our existing security policy has several limitations. Firstly, the policy is not formalized. As our company expands and new developers are onboarded, it is important that there are standardized security policies and standards we are using in our projects. Secondly, there is insufficient documentation. While the development team may be following security best practices currently, the details of that are not clearly documented. Finally, the current security policy treats security as separate from development. As we move from DevOps to DevSecOps, we must revise our security policy to more deeply integrate security into the development process.

Looking forward, our security policy should not remain static but should continue to evolve to meet new threats and follow security best practices. We also must adapt to fully take advantage of security automation tools. If a lapse in security occurs, we must review exactly what went wrong and reevaluate our policies to prevent the same mistake from occurring. Additionally, we should implement regular security training.