**Vincent Yanez**  
**April 19, 2025**  
**Project Two: Security Policy Presentation**  
**YouTube Link:** <https://www.youtube.com/watch?v=s8c3M3-sTXQ>

**Slide 1: Title Slide**

**Green Pace Security Policy Presentation**  
This presentation outlines the secure coding policies, external testing methods, and future recommendations for Green Pace developers.

**Slide 2 – Overview: Defense in Depth**

Green Pace requires all software to follow strong and consistent security practices. This security policy is needed to protect our systems and data by guiding how we build secure applications.  
The policy includes secure coding standards in C and C++, rules for authentication, authorization, and auditing, and data encryption guidelines.  
By following this policy, we support a defense-in-depth approach, meaning we protect our systems at every layer, not just one. It helps prevent vulnerabilities and ensures all developers follow the same secure process.

**Slide 3 – Threats Matrix**

In our threat matrix, we’ve classified issues by likelihood and impact.  
Likely issues include unsafe string use, SQL injection risks, and unchecked data values—all of which can cause overflows or security breaches.  
Priority issues like memory protection and data value management are critical.  
Low issues such as code readability, missing comments, and magic numbers affect maintainability more than security.  
Unlikely issues like data type overflows and buffer overflows are rare but still possible if input is not carefully checked.

**Slide 4 – 10 Principles**

**Here are the 10 key secure coding principles:**

1. Validate all input data
2. Pay attention to compiler warnings
3. Design with security in mind
4. Keep it simple
5. Deny access by default
6. Use least privilege
7. Sanitize data passed to other systems
8. Use defense-in-depth
9. Apply quality assurance techniques
10. Follow secure coding standards

**Slide 5 – Coding Standards**

We’ve outlined specific standards:

* Use input validation to prevent SQL injection
* Always initialize data to avoid undefined behavior
* Use fixed-width types for consistency
* Rely on smart pointers to protect memory
* Handle strings safely
* Use assertions to catch logic errors
* Handle exceptions clearly
* Write readable code
* Comment complex logic
* Replace magic numbers with named constants

**Slide 6 – Encryption Policies**

We apply encryption in three key areas:

* **At rest**: We use AES-256 to secure stored data
* **In flight**: We protect transmissions using HTTPS or TLS
* **In use**: We secure data even while it’s active in memory

**Slide 7 – Triple-A Policies**

* Authentication verifies who users are using passwords or MFA.
* Authorization ensures users can only access what they need.
* Accounting logs and tracks user actions, helping us detect issues.

**Slides 8 to 13 – Unit Testing**

**Positive Tests**

* Slide 8: Confirmed adding elements to collections works correctly.
* Slide 9: Verified collections are empty by default.
* Slide 10: Ensured smart pointers are initialized safely.

**Negative Tests**

* Slide 11: Out-of-range access throws correct exceptions.
* Slide 12: Accessing empty collections results in errors.
* Slide 13: Forced failures verify the test framework catches errors properly.

**Slide 14 – Automation Summary (Graph Shown)**

This graph summarizes automation coverage across planning, building, testing, deploying, and monitoring phases.

**Slide 15 – Tools**

DevSecOps integrates security from the start.

* **Plan/Design**: Threat modeling tools spot issues early.
* **Build**: Compilers enforce secure syntax.
* **Test**: Static and dynamic scans catch flaws.
* **Deploy**: Security checks validate containers.
* **Monitor**: Logs and alerts help track real-time threats.

**Slides 16 to 18 – Risks and Benefits**

**Current Issues:**

* Practices aren’t written down
* Inconsistent use by new developers
* Formal testing is lacking

**Benefits if We Act Now:**

* Better consistency and security
* Easier training
* Fewer issues down the line

**If We Wait:**

* Higher risk
* More difficult fixes
* Lack of team alignment

**Strategy Gaps:**

* No formal enforcement
* Missing documentation
* Limited vulnerability testing

**Fixes:**

* Write and follow policies
* Train developers
* Start testing
* Keep policies updated

**Slide 19 – Recommendations**

Addressing gaps:

* Document all standards
* Limit third-party access
* Automate testing
* Apply encryption consistently

**Example:** Target’s 2013 breach shows what happens when access control is weak.

**Future Actions:**

* Adopt DevSecOps
* Enforce Triple-A
* Provide regular training
* Follow OWASP and CERT C++
* Automate checks across the pipeline

**Slide 20 – Conclusions**

Final key recommendations:

* Validate inputs to prevent attacks
* Test and review code thoroughly
* Use strong encryption
* Apply least privilege
* Keep software up to date
* Have an incident response plan
* Train the entire team regularly

These practices support security as the company scales and evolves.