Christian J Busca  
CS 405 Project Two  
08/17/2025  
https://youtu.be/cvp0Y7fmjBw  
  
Slide 1 — Title  
 Hello, I’m Christian, and this is the Green Pace Security Policy Presentation. I’ll walk through our standardized policies, the threats we’re addressing, how we test and automate defenses, and the specific recommendations we’re adopting to protect our code and systems.

Slide 2 — Overview: Why a Security Policy  
 Our team has been practicing good security habits, but as we grow we need consistency and proof. This policy turns best practices into clear standards that are repeatable and auditable. It follows a defense-in-depth approach so that if one layer fails—like input validation—other layers such as parameterized queries, compiler hardening, and runtime monitoring still prevent a breach.

Slide 3 — Overview: How the Policy Is Used  
 Practically, these requirements become part of our definition of done. Pre-commit checks stop obvious mistakes. In CI we compile with strict flags and run unit tests that target vulnerabilities. Scanners catch injection patterns and dependency risks. Deploy gates block non-compliant artifacts. Every run produces evidence—logs, test reports, and SBOMs—so we can prove controls executed and respond quickly if something slips.

Slide 4 — Threats Matrix  
 This threat matrix prioritizes our risks by impact, exploitability, exposure, and detectability. Hardcoded secrets and weak cryptography are critical because compromise affects everything, so we address them first. Injection, buffer overflows, and unsafe input handling are highly exploitable and common, so they are high priority. Known vulnerabilities in dependencies are next and require fast patching. Error handling, file and path handling, and logging are medium priority because they can enable lateral movement or hinder detection. Race conditions are less frequent but can cause privilege escalation, so we mitigate them with safe concurrency patterns.

Slide 5 — Principles and Standards Mapping  
 Our ten principles are: validate input, least privilege, fail securely, defense in depth, separation of duties, minimize attack surface, secure defaults, keep security simple, distrust external systems, and ensure accountability. Each maps to one or more coding standards. For example, Validate Input maps to STD-001 Input Validation; Bounds Checking maps to STD-002 Memory Safety; Injection-Safe Data Access maps to STD-003; Secrets Management maps to STD-004; and Logging and Auditing map to STD-010. This shows our principles are concrete and enforceable in code.

Slide 6 — Coding Standards and Ranking  
 Here are our ten standards in priority order. We rank using a weighted model: Impact forty percent, Exploitability thirty percent, Exposure twenty percent, and Detectability ten percent—where less detectable means higher risk. Secrets management and cryptography are critical due to full-system blast radius. Injection safety, bounds checking, input validation, and dependency hygiene are high. Error handling, file and path safety, concurrency, and logging follow as medium to low. This keeps our focus on the most dangerous issues first while still addressing breadth.

Slide 7 — Encryption Strategy  
 In flight, we require TLS 1.3 with HSTS and strong ciphers; older protocols are disabled. At rest, we use AES-256 with keys created, rotated, and access-controlled by KMS or HSM, and we use envelope encryption to separate roles. In use, we minimize how long secrets live in memory, never log secrets, and consider trusted execution environments for the most sensitive workloads. These controls reduce interception, data theft, and key misuse.

Slide 8 — Triple-A: Authentication, Authorization, Accounting  
 For authentication, we enforce SSO with MFA for admins and developers, with strong password and session rules and modern second factors. For authorization, we use RBAC and least privilege, with policy-as-code so risky permissions can’t be deployed. For accounting, we centralize immutable logs with correlation IDs, protect them from tampering, and alert on anomalies such as failed login spikes or sudden privilege changes.

Slide 9 — Unit Test 1: Block the classic ' OR 1=1 --  
 This test contrasts an intentionally unsafe query with our parameterized query. The unsafe path broadens results, while the safe path returns zero rows. This proves the data-access layer blocks tautology injections by default.

Slide 10 — Unit Test 2: Valid inputs still work  
 Security controls must not break features. With a known good username, the DAL returns exactly one row for the correct user. This confirms our validation and parameterization don’t cause false negatives for normal use.

Slide 11 — Unit Test 3: Encoded and Unicode variants  
 We try URL-encoded and Unicode versions of injection payloads. The safe path treats operators literally and returns zero rows. This closes common bypasses that rely on input canonicalization gaps.

Slide 12 — Unit Test 4: Stacked statements and terminators  
 We simulate a payload like quote, semicolon, drop table, and comment. After the call, the users table still exists. Our driver and DAL prevent multi-statement exploits and errors fail safely.

Slide 13 — Unit Test 5: Generic error messages  
 On malformed or oversized inputs, user-facing errors stay generic while internal details are sanitized and logged. Attackers learn nothing about our schema or stack, while operators still get actionable diagnostics.

Slide 14 — Unit Test 6: Operators treated literally  
 We combine a valid email with operator-like text. Because of parameterization, the operators are literal text. The mixed input yields zero rows, while an exact email still returns one. This is the essence of being injection-safe.

Slide 15 — Demo A: Compiler flags an overflow at build time  
 In overflow\_demo we intentionally copy thirty-three bytes into a sixteen-byte buffer. With strict compiler flags, the build emits a string-overflow warning. In our pipeline, warnings are treated as errors, so this would block the pull request and keep memory-corruption bugs from shipping.

Slide 16 — Demo B: Runtime detects stack smashing  
 Running the same demo triggers “stack smashing detected” and aborts the program. Together, compiler hardening and runtime guards show how external testing and toolchain defenses detect vulnerabilities even if a bug reaches execution.

Slide 17 — Automation Summary: DevSecOps flow  
 Our pipeline is: Develop, then CI with strict compilers and unit tests; SAST and secrets scanning; software composition analysis with SBOM generation and signing; infrastructure-as-code policy checks; DAST against a short-lived preview environment; then deploy with admission controls; and finally runtime monitoring and alerting. Evidence from each stage—test results, SBOMs, signatures, and logs—is archived and used to improve tests and policies over time.

Slide 18 — Risks if we wait, and residual risks  
 If we delay, we leave a window open for SQL injection, overflow, and information leaks. Technical debt grows, and compliance findings become likely. Even after adoption, we must manage pipeline friction, tool sprawl, and ensure high availability for signing and provenance systems. Preview environments must be isolated and auto torn down, and centralized logs must enforce PII scrubbing and retention policies.

Slide 19 — Benefits of acting now  
 Acting now immediately reduces high-impact risk with parameterized queries, hardened builds, and sanitized errors. It improves audit readiness with consistent evidence, enables safer deployments through policy gates and progressive rollouts, and reduces rework by catching defects early. Developers gain clarity on what “secure enough to merge” means.

Slide 20 — Recommendations: Gaps to close  
 Our immediate gaps are: adopt a formal risk-acceptance process and SLAs; expand beyond SQLi and overflow to include XSS, CSRF, SSRF, path traversal, and unsafe deserialization; standardize mTLS and token validation service-to-service; implement data classification and clear encryption-in-use guidance; enforce CIS hardening and runtime policies in Kubernetes; require SBOM generation and artifact signing for all services; add DAST for authenticated flows and fuzz or property-based tests; implement pre-commit secrets scanning; and finalize PII log scrubbing, retention, and incident response runbooks with restore drills.

Slide 21 — Conclusion: Standards to adopt  
 To close those gaps we will adopt: NIST SSDF, OWASP SAMM, and ISO 27001/27002 for governance; OWASP ASVS and API Top 10, CWE Top 25, SEI CERT C++, C++ Core Guidelines, and strict compiler plus sanitizer flags for code; NIST 800-63B, OAuth 2.1 and OpenID Connect, and WebAuthn for identity; TLS 1.3, BCP 195 guidance, NIST 800-57 key management, FIPS 140-3 modules, and mTLS for cryptography; CIS Benchmarks, NSA and CISA Kubernetes Hardening, Pod Security Standards, and OPA or Gatekeeper for infrastructure; SLSA, NIST 800-161, SBOM standards such as SPDX or CycloneDX, VEX, SCA with service-level agreements, and automated dependency updates for supply chain; and finally SAST, DAST, fuzzing, secrets scanning, OpenTelemetry, and NIST guidance for logging and incident response.

Slide 22 — References  
 This final slide lists the condensed APA references grouped by topic—governance, application security, identity, crypto, infrastructure, supply chain, testing and secrets, and incident response. These sources back each policy and tool we’ve adopted.