Secure SDLC Integration (PCI DSS 4.0 Alignment)

Step 1: Reviewing Our Development Lifecycle

To begin, I sat down with our DevOps and engineering teams to understand how our software is built from design to deployment.

Here's what I observed:

- Our teams follow Agile with 2-week sprints and use Jenkins for CI/CD.
- Secure design principles are not consistently applied, threat modeling is absent.
- Peer code reviews are performed, but security isn't a formal checkpoint.
- Logging, error handling, and credential management practices vary by team.
- No structured process for verifying PCI DSS security requirements during development.

I documented these observations to map them against PCI DSS 4.0 controls, particularly Sections 6 and 8.

Step 2: Integrating PCI DSS Secure Coding Practices

Next, I began integrating secure development practices based on PCI DSS:

Input Validation

I enforced strong server-side input validation and output encoding:

- Implemented a shared validation utility across services.
- Integrated OWASP ESAPI principles into new backend APIs.

Authentication & Session Management

We updated all login flows to align with PCI DSS Req. 8:

- Passwords are now hashed with bcrypt and salted.
- Session tokens expire after 15 minutes of inactivity.
- MFA is enforced for admin and developer portals.

Secure Configuration

I worked with DevOps to ensure:

- TLS 1.2+ is enforced across all environments.
- Default passwords are removed from test builds.
- Secrets are no longer hardcoded, we now use HashiCorp Vault.

Security Testing in CI/CD

- Added SAST using GitHub Advanced Security for real-time scanning.
- Required DAST runs in QA environments for all releases.
- Created “security gates” in Jenkins, no build passes with high-severity CVEs.

Step 3: Conducting a PCI DSS-Oriented Code Review

I led a deep dive into our most sensitive module: the payment processing service.

Issues I discovered:

- SQL queries constructed with string concatenation (SQL risk).
- Logging exposed full PAN during failed transactions (violates PCI Req. 3.4).
- Token expiration logic missing from password reset flow.

What I did:

- Refactored queries using parameterized statements (Java PreparedStatement).
- Implemented a maskCardNumber utility for all logs.
- Updated the password reset system to use short-lived, single-use JWTs.

I also documented secure code patterns and added them to our internal code review checklist.

Step 4: Developer Training Program Rollout

To build long-term sustainability, I developed a tailored PCI Secure Coding Training for our developers.

Training Format:

- 5-part video series (hosted internally)
- Monthly live secure coding labs
- GitHub "bad vs. good code" examples

Core Modules:

PCI DSS for Developers – Why it matters to us

Input Validation & Injection Defense – Practical OWASP coverage

Secure Auth & Access Control – Passwords, sessions, and MFA

Data Protection & Secure Logging – Masking PANs, avoiding sensitive debug info

Code Review Walkthrough – Practice spotting flaws in real PRs

Attendance is now required for all engineers working on cardholder-related functionality.

Step 5: Presenting the New Secure SDLC to the Team

I hosted a 45-minute session titled:

"Baking Security into Our Code: PCI DSS and You"

During the session, I shared:

- A revised SDLC diagram, with security added to design, code, test, and deploy stages.
- Real breach examples (Target, British Airways) and how simple dev mistakes led to major fines.

Our new dev checklist:

- No PAN logging
- Use prepared statements only
- Validate every input
- Rotate and vault secrets
- Review with security in mind

I ended with:

> “Security isn’t just a final checkbox — it’s a team responsibility. What you build today protects our customers tomorrow.”

Results and Next Steps

Since implementing these changes:

- Vulnerability rates in production code have dropped by 60%.
- Developers proactively raise security questions during code reviews.
- We've scheduled our first internal PCI DSS mock audit for Q3 2025.