Software Assurance: CSS321

A Pet Nose Best: Security in Nontraditional Development Models

Antoine Gaton

September 22, 2024

Table of Contents

**Project Outline3**

**Security in the Development Life Cycle4**

**Software Assurance Techniques6**

Analysis of Software Applications**6**

Guidelines for Software Assurance Techniques**8**

**Security in Nontraditional Development Models10**

Major Steps and Potential Security Threats**10**

Policies and Processes to Minimize Security Threats**11**

**Security Static Analysis13**

TBA**14**

TBA**15**

**Software Assurance Policies and Processes16**

TBA**17**

TBA**18**

# Project Outline

1. Organization
   1. *A Pet Nose Best*
      1. This organization provides both desktop and web-based software solutions for pet care service management, focusing on both private-sector pet care companies and government contracts related to pet licensing and animal welfare.
         1. *Note: This is my fiancé’s business which I am using as inspiration for this individual project.*
   2. Company Size
      1. Medium-sized: Approximately 150 employees
   3. Locations
      1. Miami
      2. New York
      3. San Francisco
      4. Austin
   4. Application
      1. Pet Care Management System (PCMS)
         1. Desktop application
      2. Government Pet Licensing Portal
         1. Web/Mobile application
   5. Database
      1. SQL Server
         1. Used for the PCMS desktop application.
      2. Firebase
         1. Used for the government per licensing portal.
2. Software Development Organization
   1. The software development team includes developers, QA engineers, DevOps, and security engineers, all reporting through the CTO. The company follows standard software development practices, including using Git for source control, CI/CD pipelines with Jenkins, and Jira for project management.

# Security in the Development Life Cycle

1. **Using Agile Methodology**
   1. **Sprint Planning**
      1. **Security Component**: Incorporate security requirements as part of the user stories. This includes prioritizing security tasks and defining security acceptance criteria for each sprint.
      2. **Tasks**: Conduct a threat model, identify potential risks, and ensure security requirements are added to the product backlog.
   2. **Design and Architecture in Sprint**
      1. **Security Component**: Continuous security architecture review. With each new feature or update, the design is evaluated for security risks.
      2. **Tasks**: Perform lightweight design reviews with a focus on security in each sprint and ensure any new security implications are documented.
   3. **Development (Coding) in Sprint**
      1. **Security Component**: Secure coding practices are integrated into each sprint. Developers adhere to secure coding standards and follow best practices for preventing security vulnerabilities.
      2. **Tasks**: Conduct peer reviews and code analysis focused on security vulnerabilities during the development process.
   4. **Testing (Quality Assurance) in Sprint**
      1. **Security Component**: Continuous security testing. Both manual and automated tests are conducted to identify potential security vulnerabilities.
      2. **Tasks**: Perform ongoing security testing, such as static analysis, dynamic testing, and penetration testing, as part of the sprint's quality assurance activities.
   5. **Sprint Review and Retrospective**
      1. **Security Component**: Evaluate security outcomes from the sprint. Identify what security issues were addressed and what vulnerabilities remain.
      2. **Tasks**: Review the security acceptance criteria and discuss any security concerns that arose during the sprint. Update the backlog with any new security tasks or issues.
   6. **Ongoing Monitoring and Maintenance**
      1. **Security Component**: Post-release, continuously monitor for security incidents. Security is not a one-time effort but rather a constant process of patching and updating.
      2. **Tasks**: Implement security patches, monitor for vulnerabilities, and respond to incidents in real-time. Conduct regular security reviews even after product deployment.

# Software Assurance Techniques

**Analysis of Software Applications**

**1. Pet Care Management System (PCMS) - Desktop Application**

* **Description**: The PCMS is a desktop-based software application designed for private pet care companies. It allows for managing client and pet data, appointments, services, and invoicing. The application stores sensitive data in a local SQL Server database.
* **Security Risks:**
  + **User Authentication and Authorization**
    - ***Threat***: Unauthorized access to client and pet data due to weak authentication mechanisms.
    - ***Implications***: Compromised sensitive data, legal liabilities, and loss of client trust.
    - ***Mitigation***: Implement multi-factor authentication (MFA) and role-based access control (RBAC). Using MFA ensures that even if login credentials are stolen, an additional layer of verification prevents unauthorized access. RBAC limits users to only the information and actions necessary for their role.
  + **Database Security**:
    - ***Threat***: SQL injection attacks that exploit vulnerabilities in the application’s database queries.
    - ***Implications***: Data breaches, data loss, and potential financial losses.
    - ***Mitigation***: Employ parameterized queries or stored procedures to prevent SQL injection. Regular vulnerability scans should also be conducted to ensure that the application is not susceptible to this attack.

**2. Government Pet Licensing Portal - Web/Mobile Application**

* **Description**: This web-based application serves as a portal for local governments to manage pet licensing and related information. It is accessible through both browsers and mobile devices, using Firebase for data storage.
* **Security Risks**:
  + **Cross-Site Scripting (XSS)**
    - ***Threat***: Malicious scripts injected into the website, compromising user data and sessions.
    - ***Implications***: Loss of sensitive government data, user accounts being hijacked, and exposure to future attacks.
    - ***Mitigation***: Implement content security policies (CSP) and sanitize all input fields. CSP prevents unauthorized scripts from running, while input sanitization ensures user inputs are safely processed.
  + **Data Transmission Security**:
    - ***Threat***: Man-in-the-middle attacks during data transmission, leading to the interception of sensitive information such as pet registration and user credentials.
    - ***Implications***: Breaches in the confidentiality of government information, data leaks, and public distrust.
    - ***Mitigation***: Ensure all data transmission uses secure communication protocols like HTTPS with Transport Layer Security (TLS). Also, enforce the use of strong encryption methods to protect data in transit.

**3. Pet Tracking Mobile App**

* **Description**: A mobile application that allows users to track their pets using GPS. The app store’s location data and user preferences in the cloud, likely integrating with Firebase for cloud services.
* **Security Risks**:
  + **Location Data Leakage**
    - ***Threat***: Unauthorized access to real-time GPS data, allowing tracking of users and pets.
    - ***Implications***: Physical safety risks for users and pets, breaches of privacy laws, and reputational damage.
    - ***Mitigation***: Implement strong encryption for stored and transmitted location data. Use anonymization techniques to protect users' identities when sharing location information.
  + **Mobile Device Security**
    - ***Threat***: Compromised mobile devices accessing the app, exposing sensitive data and user accounts.
    - ***Implications***: Data breaches, unauthorized control over the app, and exposure to malware.
    - ***Mitigation***: Use device-specific encryption and enforce biometric authentication (e.g., fingerprint or facial recognition). Additionally, regular updates should patch any discovered vulnerabilities in mobile platforms.

**Guidelines for Software Assurance Techniques**

1. **Desktop Applications**

* **User Authentication and Authorization**:
  + Implement Multi-Factor Authentication (MFA) and Role-Based Access Control (RBAC).
  + Secure login mechanisms, including password complexity requirements and lockout policies after repeated failed login attempts.
  + Regularly review and audit access control logs to detect any unauthorized attempts.
* **Database Security**:
  + Use parameterized queries or stored procedures to protect against SQL injection attacks.
  + Conduct regular database vulnerability assessments and implement real-time monitoring for suspicious activities.
  + Apply database encryption techniques for sensitive data storage.

1. **Web Applications**

* **Cross-Site Scripting (XSS) Prevention**:
  + Sanitize all user inputs using whitelisting approaches, and escape output to prevent malicious code execution.
  + Implement Content Security Policy (CSP) to limit the execution of unauthorized scripts.
  + Regularly perform dynamic and static code analysis to detect XSS vulnerabilities early.
* **Data Transmission Security**:
  + All data between the client and the server should be transmitted using TLS with strong encryption ciphers.
  + Implement HTTP Strict Transport Security (HSTS) to ensure that the application can only be accessed over secure connections.
  + Use encryption for sensitive data stored in the cloud (e.g., government licensing data).

1. **Mobile Applications**

* **Location Data Protection**:
  + Encrypt GPS and personal information both at rest and during transmission to cloud services.
  + Use anonymization for location data before sharing or storing it to avoid exposing users' exact locations.
* **Mobile Device Security**:
  + Enforce strong device authentication methods, such as biometric authentication or device-specific encryption.
  + Ensure that the mobile app is frequently updated to patch known vulnerabilities.
  + Use secure APIs to communicate between mobile apps and cloud services.

# Security in Nontraditional Development Models

**Non-Traditional Development Model: DevSecOps**

For A Pet Nose Best, I recommend implementing a DevSecOps model. This non-traditional approach integrates security practices within the DevOps process, ensuring that security is a shared responsibility throughout the entire software development lifecycle.

**Major Steps and Potential Security Threats**

**1. Plan**

**Description**: In this phase, project requirements are gathered, and the development strategy is outlined.

**Potential Security Threats**:

* Inadequate security requirements specification
* Overlooking potential security risks in the planning stage

**2. Code**

**Description**: Developers write code for the application features.

**Potential Security Threats**:

* Introduction of security vulnerabilities through insecure coding practices
* Use of outdated or vulnerable third-party libraries

**3. Build**

**Description**: The application is compiled and built into an executable form.

**Potential Security Threats**:

* Inclusion of sensitive information (e.g., API keys, passwords) in the build artifacts
* Use of insecure build tools or processes

**4. Test**

**Description**: The application undergoes various testing phases, including unit testing, integration testing, and user acceptance testing.

**Potential Security Threats**:

* Insufficient security testing coverage
* Failure to identify critical security vulnerabilities

**5. Release**

**Description**: The application is prepared for deployment to production environments.

**Potential Security Threats**:

* Unauthorized changes to release artifacts
* Insecure handling of production credentials

**6. Deploy**

**Description**: The application is deployed to production environments.

**Potential Security Threats**:

* Misconfiguration of production environments
* Exposure of sensitive data during deployment

**7. Operate**

**Description**: The application runs in production, serving end-users.

**Potential Security Threats**:

* Runtime vulnerabilities
* Inadequate monitoring for security incidents

**8. Monitor**

**Description**: The application and infrastructure are continuously monitored for performance and security issues.

**Potential Security Threats**:

* Delayed detection of security breaches
* Insufficient logging of security-relevant events

**Policies and Processes to Minimize Security Threats**

**1. Plan**

**Policy**: Security Requirements Integration

* All project plans must include explicit security requirements.
* Threat modeling must be conducted for each new feature or major change.

**Process**:

1. Include a security expert in all planning meetings.
2. Use a standardized security requirements checklist for each project.
3. Conduct and document threat modeling sessions for each new feature.

**2. Code**

**Policy**: Secure Coding Standards

* All developers must adhere to the company's secure coding guidelines.
* Regular code reviews with a focus on security must be conducted.

**Process**:

1. Implement automated code analysis tools in the development environment.
2. Conduct peer code reviews with a security checklist before any merge.
3. Maintain an up-to-date inventory of approved and vetted third-party libraries.

**3. Build**

**Policy**: Secure Build Process

* All build processes must be automated and version-controlled.
* Sensitive information must never be hard-coded in the source.

**Process**:

1. Use a secure, centralized secret management system for handling sensitive data.
2. Implement automated checks to detect hard-coded secrets in the codebase.
3. Ensure all build tools are regularly updated and securely configured.

**4. Test**

**Policy**: Comprehensive Security Testing

* Automated security testing must be integrated into the CI/CD pipeline.
* Penetration testing must be conducted before major releases.

**Process**:

1. Integrate SAST (Static Application Security Testing) and DAST (Dynamic Application Security Testing) tools into the CI/CD pipeline.
2. Conduct regular security-focused test scenarios as part of QA processes.
3. Engage third-party penetration testers for annual security assessments.

**5. Release**

**Policy**: Secure Release Management

* All releases must go through a security sign-off process.
* Release artifacts must be cryptographically signed.

**Process**:

1. Implement a release checklist that includes security verification steps.
2. Use a secure, audited process for managing and rotating production credentials.
3. Implement artifact signing in the release process and verify signatures before deployment.

**6. Deploy**

**Policy**: Secure Deployment Practices

* All production deployments must use infrastructure-as-code principles.
* Deployment processes must be fully automated with minimal human intervention.

**Process**:

1. Use version-controlled infrastructure-as-code tools (e.g., Terraform) for all deployments.
2. Implement automated configuration checks post-deployment to verify security settings.
3. Use blue-green deployment strategies to minimize risk and enable quick rollbacks.

**7. Operate**

**Policy**: Runtime Security Monitoring

* All production systems must have real-time security monitoring enabled.
* Incident response plans must be in place and regularly tested.

**Process**:

1. Implement a SIEM (Security Information and Event Management) system for centralized logging and alerting.
2. Conduct regular security drills to test incident response procedures.
3. Implement runtime application self-protection (RASP) tools where applicable.

**8. Monitor**

**Policy**: Proactive Security Monitoring and Response

* Security-relevant events must be logged and retained for at least one year.
* Regular security reports must be generated and reviewed by the security team.

**Process**:

1. Implement automated alerting for suspicious activities or potential security breaches.
2. Conduct weekly security review meetings to discuss monitoring reports and address any concerns.
3. Regularly update monitoring rules and alert thresholds based on emerging threats and past incidents.

# Security Static Analysis

# Software Assurance Policies and Processes