Summary of PCI-DSS Requirement 6: Develop Securely

Goal: Ensure that both **custom-developed** and **third-party software** used in the environment are secure and do not introduce vulnerabilities. This requirement encompasses all phases of the **Software Development Life Cycle (SDLC)**, including identifying vulnerabilities, installing patches, and developer training.

Key Sub-Requirements of Requirement 6

1. Vulnerability and Risk Ranking (6.1)

- **Description**: Every identified vulnerability must be ranked based on its risk to the environment. Risk ranking determines how quickly it needs to be addressed.
- **Action Steps**:
    - Regularly assess vulnerabilities published by third-party vendors, InfoSec communities, and other sources.
    - Classify vulnerabilities (e.g., "critical," "high," "medium," "low") based on their potential impact on systems handling cardholder data (CHD).

Example:

A vulnerability affecting a payment application is rated "critical" because it allows unauthorized access to customer payment data. It must be patched immediately to mitigate risk.

---

2. Timely Patch Installation (6.2)

- **Description**: Install critical security patches as soon as possible, preferably within **30 days** of release. Non-critical patches should be installed within **2-3 months**.
- **Action Steps**:
    - Implement a process for applying security patches promptly to all systems handling CHD.
    - Document the timeline for installing patches based on risk classification (critical vs. low-risk).

Example:

A new version of a payment server software is released with a critical security fix. The IT team schedules the update within the first week of the patch release.

---

3. Secure SDLC (6.3)

- **Description**: Integrate security practices throughout the SDLC, ensuring that security requirements are considered during each stage of development.
- **Action Steps**:
    - Train developers to write secure code and follow best practices.
    - Conduct **code reviews** and **penetration testing** before deploying software into production.

        - Ensure **security testing** is part of the CI/CD pipeline for Agile or DevOps teams.

Example:

Before releasing a new e-commerce feature, the development team performs code reviews focused on common vulnerabilities like SQL injection, ensuring the code is secure.

---

4. Development Change Controls (6.4)

- **Description**: Manage changes in the development and production environments to prevent vulnerabilities.
- **Action Steps**:
    - Separate **development** and **production environments** to avoid cross-contamination.
    - Remove **test credentials** and **developer accounts** from production.
    - Reassess PCI-DSS compliance after significant changes.

Example:

After adding new payment processing functionality, the development team removes any test accounts and retests for PCI-DSS compliance to ensure the environment remains secure.

---

5. Secure Development Training (6.5)

- **Description**: Provide regular training for developers to address common vulnerabilities such as **SQL injection**, **cross-site scripting (XSS)**, and **buffer overflows**.
- **Action Steps**:
    - Ensure all developers are trained on secure coding practices and common vulnerabilities.
    - Implement an annual training schedule to keep developers updated on the latest threats.

Example:

The development team completes annual secure coding training that focuses on preventing SQL injection attacks in web applications.

---

6. Prevent Public Vulnerabilities (6.6)

- **Description**: Public-facing applications must be reviewed for vulnerabilities at least **once per year** or after any significant changes.
- **Action Steps**:
    - Regularly review public applications using **automated vulnerability scanning tools**.
    - Perform manual reviews after every significant change in the application code.

Example:

A vulnerability scanner is scheduled to review an online payment gateway for vulnerabilities every quarter to identify any security gaps.

---

7. Document and Enforce Policies and Procedures (6.7)

- **Description**: All policies and procedures for secure development, vulnerability ranking, and patching must be documented and enforced.
- **Action Steps**:
  - Maintain a comprehensive set of documentation that covers the SDLC, patch management, developer training, and vulnerability management.
  - Ensure developers follow documented procedures and undergo regular security audits.

Example:

The company's development team follows a documented **change management policy** that details how code is reviewed, tested, and deployed to production.

Roles and Responsibilities

1. IT Security Team

- **Responsibilities**:
  - Ensure vulnerability management processes are followed, including risk ranking and timely patching.
  - Work closely with development teams to integrate security into the SDLC.
- **Tools to Use**:
  - **Vulnerability Management Systems** (e.g., Tenable, Qualys).
  - **Patch Management Tools** (e.g., WSUS, SCCM).

2. Developers

- **Responsibilities**:
  - Write secure code following best practices and undergo regular security training.
  - Conduct code reviews, unit testing, and vulnerability assessments.
- **Tools to Use**:
  - **Static Code Analysis Tools** (e.g., SonarQube, Checkmarx).
  - **CI/CD Pipeline Integration** for security testing.

3. Project Managers

- **Responsibilities**:
  - Ensure that security requirements are considered in project planning.
  - Track compliance with security best practices throughout the SDLC.
- **Tools to Use**:
  - **Project management software** (e.g., Jira) for tracking security tasks.
  - **Compliance checklists** to ensure policies are followed.

4. System Administrators

- **Responsibilities**:

    - Implement and manage patch management for third-party and internally developed software.
    - Monitor and log updates to ensure compliance with patching timelines.
- **Tools to Use**:
    - **Patch management software**.
    - **Logging and monitoring tools** for patch installations.

Sample Policy Documents

1. Secure Development Policy

- **Purpose**: Define secure development practices and ensure security is part of the SDLC.
- **Key Elements**:
    - Security in design, coding, and testing phases.
    - Regular code reviews and developer training.
    - Secure handling of development accounts and credentials.

2. Vulnerability Management and Patch Installation Policy

- **Purpose**: Outline the process for identifying, classifying, and addressing vulnerabilities through timely patch installation.
- **Key Elements**:
    - Vulnerability risk ranking and patch prioritization.
    - Timelines for critical and non-critical patches.
    - Regular monitoring for new vulnerabilities.

Conclusion and Best Practices for the Company

1. **Integrate Security into the SDLC**: Security should be considered at every stage of the SDLC, from design to deployment.
2. **Timely Patch Installation**: Ensure all critical patches are applied within **30 days** to minimize risk exposure.
3. **Regular Developer Training**: Keep developers trained on common vulnerabilities and secure coding practices.
4. **Document and Enforce Policies**: Maintain and enforce security policies and procedures for all stages of development and vulnerability management.

These policies can help ensure the secure development of software and protect the company’s sensitive data, including cardholder information. If you need downloadable files, please let me know how you'd like me to proceed!

***

Here’s a tailored approach to **PCI-DSS Requirement 6** for a development team working with **Laravel API**. I’ll give examples and sample policies with specific tools and processes that would apply to this environment.

Security in Design, Coding, and Testing Phases (Laravel API)

For a development team using **Laravel API**, security must be integrated into each phase of the Software Development Life Cycle (SDLC), ensuring that sensitive information, like **Cardholder Data (CHD)**, is protected throughout the process.

---

1. Security in the Design Phase

In the design phase, it's essential to incorporate secure architecture and ensure that the Laravel API aligns with security best practices from the start.

Example:

- **Designing API Authentication**: Use **OAuth 2.0** or **JWT (JSON Web Tokens)** for secure API authentication. Plan to use **Laravel Passport** or **Laravel Sanctum** to manage tokens securely.
- **Input Validation**: Plan for **data sanitization and validation** in every API endpoint to prevent injection attacks, such as **SQL injection** and **cross-site scripting (XSS)**.

Documentation:

- Include API documentation for secure endpoints (using tools like **Swagger**).
- Design security layers, such as encryption for stored data and HTTPS for all API communication.

---

2. Security in the Coding Phase

During the coding phase, developers must follow secure coding practices and avoid common vulnerabilities, especially when handling CHD.

Example:

- **Using Eloquent ORM**: Leverage Laravel's **Eloquent ORM** to prevent SQL injection. Laravel automatically escapes parameters in SQL queries to reduce risk.
- **Secure Input Handling**: Use Laravel's **Request validation** to sanitize and validate user inputs in all API requests.
- **File Uploads**: Implement file validation in Laravel for any uploaded files to avoid arbitrary file uploads that could harm the system (e.g., malicious file uploads).

Tools:

- **Static Code Analysis Tools** like **SonarQube** or **PHPStan** can be used to review Laravel code for potential vulnerabilities, such as unvalidated inputs or unsafe function usage.
- **Laravel Security Packages**:
  - **Laravel Passport** or **Laravel Sanctum**: Secure API authentication.
  - **Laravel's built-in CSRF protection**: Ensure that cross-site request forgery is prevented in forms and state-changing API actions.

---

3. Security in the Testing Phase

In the testing phase, developers need to perform both automated and manual security testing before deploying the Laravel API.

Example:

- **Testing for SQL Injection**: Write unit tests to ensure that all user input is validated and sanitized before interacting with the database.
- **Penetration Testing**: Perform penetration testing on your Laravel API endpoints, focusing on authentication and sensitive data (e.g., CHD).
- **Automated Security Scans**: Run security tools like **OWASP ZAP** or **Burp Suite** against the API to detect vulnerabilities in authentication, session management, and input validation.

Tools:

- **PHPUnit**: For writing tests to ensure that all API routes are secure and handle inputs correctly.
- **Laravel Dusk**: Useful for testing browser-based applications and ensuring the front-end securely interacts with your API.
- **OWASP ZAP**: Automated tool to scan for vulnerabilities in your API.

---

4. Example of Policies and Procedures

A. Secure Development Policy (Laravel API)

Purpose:
Define secure development practices for all stages of the SDLC, ensuring that the Laravel API follows PCI-DSS guidelines for secure data handling.

Key Elements:

1. **Security Requirements in Design**:
    - Define secure authentication methods (e.g., **OAuth 2.0** using Laravel Passport).
    - Use HTTPS for all API endpoints, ensuring secure data transmission.
2. **Secure Coding Standards**:
    - All inputs must be validated and sanitized using Laravel's **validation rules**.
    - Implement role-based access control (RBAC) for API endpoints.
3. **Testing and Code Review**:
    - Use **PHPUnit** for security testing on all Laravel API routes.
    - Ensure regular code reviews to detect potential vulnerabilities (e.g., SQL injection).

B. Vulnerability Management and Patch Installation Policy (Laravel API)

Purpose:
Outline the process for identifying, classifying, and addressing vulnerabilities and patching Laravel-based applications.

Key Elements:

1. **Vulnerability Risk Ranking**:
    - Use a risk ranking system (e.g., **critical**, **high**, **medium**, **low**) to prioritize patches. For example, a vulnerability in Laravel's **authentication system** is critical.
    - Ensure that vulnerabilities like unprotected API routes are patched within 30 days of discovery.
2. **Patch Installation Timeline**:
    - Use **Composer** to manage and update Laravel packages.
    - Apply patches for Laravel, PHP, and any third-party dependencies within the specified timeline (e.g., critical patches within 30 days).
3. **Logging and Monitoring**:
    - Implement logging using **Laravel's Monolog** to track patch updates and application errors.
    - Ensure that all patch-related activities are logged and stored for at least 90 days for auditing.

Real-World Example of Laravel API Implementation

Scenario:

An e-commerce platform built on Laravel uses a custom API to process customer payments. The development team is responsible for ensuring the API is secure and compliant with PCI-DSS.

1. **Design Phase**:
    - The team designs the API with **JWT authentication** using **Laravel Sanctum**, ensuring that all sensitive API routes are protected.
    - HTTPS is enforced on all routes to protect CHD during transmission.
2. **Coding Phase**:
    - Developers use **Laravel validation** to ensure all input fields are sanitized.
    - **CSRF protection** is enabled for all state-changing requests to protect against CSRF attacks.
3. **Testing Phase**:
    - The team writes unit tests using **PHPUnit** to validate all input fields and ensure that no SQL injection vulnerabilities exist.
    - The API undergoes regular scans using **OWASP ZAP** to detect potential vulnerabilities.
4. **Patch Management**:
    - When Laravel releases a critical security patch, the team uses **Composer** to update the API and apply the patch within **48 hours**.

Conclusion

To ensure the security of a **Laravel API**, developers need to integrate security best practices at every stage, from design to deployment. Using Laravel's built-in security features, such as **Eloquent ORM**, **Sanctum**, and **CSRF protection**, can significantly reduce the risk of vulnerabilities. Additionally, regular training, patching, and testing are essential to maintaining a secure environment for CHD.

***

Comprehensive Security Overview for Laravel API Development Team

Incorporating security in a Laravel API is essential to protecting sensitive information, including **Cardholder Data (CHD)**, and ensuring compliance with industry standards like **PCI-DSS**. Here's a detailed guide based on the **Laravel Enlightn Security documentation** and best practices to ensure that nothing related to security is overlooked.

---

1. Authentication and Authorization

Laravel offers built-in tools to implement robust authentication and authorization, making it easier to secure API endpoints.

Authentication

- **OAuth 2.0** and **JWT (JSON Web Tokens)** can be implemented using **Laravel Passport** or **Laravel Sanctum**. These solutions ensure that tokens securely identify and authorize users.

  Example: Use Laravel **Sanctum** to issue API tokens to users and limit them to specific scopes (e.g., read-only access to certain data).

Authorization

- Implement **role-based access control (RBAC)** using **Laravel Gates** and **Policies**. This allows fine-grained control over user actions based on roles and permissions.

  Example: Define policies that restrict access to payment processing endpoints to users with specific roles (e.g., admin).

---

2. Input Validation and Data Sanitization

Proper validation and sanitization of inputs are critical to avoid security vulnerabilities like **SQL injection** and **XSS (Cross-site scripting)**.

Input Validation

- Use **Laravel's Request Validation** feature to ensure that all incoming data is properly validated before interacting with the database.

Example: Validate the format of customer names, emails, and card information in API requests using Laravel's built-in validation rules.

Data Sanitization

- Use **Laravel's filtering** to remove unwanted characters from inputs, ensuring that all data passed to the database is clean and secure.

  Example: Strip out harmful scripts from input fields to prevent XSS attacks.

---

3. Secure Password Storage

Laravel uses **bcrypt** for password hashing, but it also supports more advanced algorithms like **Argon2**.

- Use **bcrypt** or **Argon2** for hashing passwords and never store passwords in plain text.

  Example: When a user creates or updates their account, hash their password using `Hash::make($password)` to ensure secure storage.

---

4. Protecting Against SQL Injection

Laravel's **Eloquent ORM** and **Query Builder** automatically protect against SQL injection by binding query parameters.

- Always use Eloquent or Query Builder for database interactions to prevent manually constructing SQL queries with unsanitized inputs.

  Example: When querying user data, always use the ORM: `User::where('email', $email)->first();`, which ensures the input is properly escaped.

---

5. Cross-Site Request Forgery (CSRF) Protection

CSRF attacks occur when malicious websites trick users into submitting requests to your Laravel application. Laravel provides built-in **CSRF protection** for all forms and state-changing requests.

- Enable **CSRF protection** for all forms and ensure API routes handling state changes are protected.

  Example: Use the `csrf_token()` in forms or API routes where user state is altered (e.g., updating profile data).

---

6. HTTPS (SSL/TLS) Encryption

Ensure that all communications between the client and the Laravel API are encrypted by forcing **HTTPS** on all routes.

- Enforce **HTTPS** using the `AppServiceProvider` by setting `URL::forceScheme('https');` in production environments.

  Example: Ensure that sensitive routes such as login, registration, and payment processing are only accessible over HTTPS.

---

7. Preventing Cross-Site Scripting (XSS)

XSS attacks occur when malicious scripts are injected into web pages viewed by other users. Laravel automatically escapes output to prevent XSS.

- Ensure that all dynamic content is properly escaped using Laravel's built-in `{{ }}` syntax for output.

  Example: When displaying user-generated content, such as comments or messages, use `{{ $comment->body }}` to ensure it is properly escaped.

---

8. Secure API Rate Limiting

To protect against brute-force attacks, Laravel provides **API rate limiting** out of the box.

- Use **Laravel's built-in rate limiting** to restrict the number of requests users can make to sensitive routes (e.g., login, password reset).

  Example: Implement rate limiting in your API routes:

  ```
  php
  Copy code
  Route::middleware('throttle:60,1')->group(function () {
      // API routes
  });
  ```

---

9. Logging and Monitoring

To detect security incidents, logging and monitoring API activity is crucial. Laravel provides extensive logging support.

- Log all authentication attempts, failed requests, and unusual activity using **Laravel's Log facade** and ensure logs are rotated and stored securely.

  Example: Use `Log::info('User login attempt', ['user' => $user->id]);` to log authentication attempts and monitor for potential attacks.

---

10. Secure Headers (CORS)

Control which domains are allowed to interact with your API by configuring **CORS (Cross-Origin Resource Sharing)** headers.

- Set up **Laravel CORS configuration** to restrict access to the API from only trusted domains.

  Example: Configure the CORS settings in `config/cors.php` to limit the allowed domains and headers for API interactions.

  Set strict CORS policies in `config/cors.php` to allow only trusted origins and restrict headers, methods, and credentials to what's necessary.

  ```
  'paths' => ['api/*'],
  'allowed_methods' => ['GET', 'POST', 'PUT'],
  'allowed_origins' => ['https://your-frontend.com'],
  'allowed_headers' => ['Content-Type', 'Authorization'],
  ```

---

11. Secure Session Handling

Laravel automatically secures session data, but additional measures can be taken, such as using **secure cookies** and limiting session time.

- Use **Laravel's secure session cookies** to ensure session data is encrypted in transit.

  Example: Set the `SESSION_SECURE_COOKIE=true` in the `.env` file to ensure session cookies are only transmitted over HTTPS.

---

12. Prevent Directory Traversal Attacks

Limit file upload vulnerabilities and prevent directory traversal attacks by validating file paths and ensuring they remain within the correct directory.

- Ensure uploaded files are stored securely and validated using **Laravel's file validation**.

  Example:

  ```
  php
  Copy code
  $request->validate([
      'file' => 'required|mimes:jpeg,png,pdf|max:1024',
  ]);
  ```

---

13. Security Alerts and Patching

Stay updated on security patches and Laravel's security advisories. Ensure **Composer** dependencies are regularly updated and patched.

- Use the `composer audit` command to check for known vulnerabilities in third-party packages.

Example:

- Run `composer audit` regularly to ensure that dependencies, including **Laravel packages**, are free from vulnerabilities.

Documentation: Monitor Laravel's security advisories.

---

14. Laravel Encryption

Laravel offers easy-to-use encryption services for securing sensitive data in your application.

- Use Laravel's `Crypt` facade to encrypt and decrypt sensitive data, such as **payment tokens** or **CHD**.

  Example:

  ```
  php
  Copy code
  $encrypted = Crypt::encryptString('sensitive_data');
  $decrypted = Crypt::decryptString($encrypted);
  ```

---

15. Security-Related Laravel Packages

Use the **Enlightn** package for Laravel, which automatically scans your application for security vulnerabilities and best practice violations.

- Install **Enlightn** and run regular security checks to maintain compliance with Laravel's best security practices.

Example:
Run Enlightn security checks in production:

```
bash
Copy code
php artisan enlightn
```

---

16. Preventing File Tampering

Laravel provides **file integrity checks** to detect tampering. Implement regular hash checks on files that are critical to the system's security.

- Ensure that system-critical files (such as `.env` and configuration files) are monitored for unauthorized changes.

Example: Use a monitoring tool to track changes in key files and send alerts if they are modified unexpectedly.

---

17. Prevent Insecure Redirects

When handling redirects in your Laravel API, avoid **open redirects**, which could be exploited to redirect users to malicious websites.

- **Example**: Validate redirect URLs before issuing them to prevent attackers from injecting malicious URLs.

```
// Validate the redirect URL
$url = $request->input('redirect_url');
if (!filter_var($url, FILTER_VALIDATE_URL)) {
    abort(400, 'Invalid URL');
}
```

---

18. Preventing Timing Attacks

Laravel uses constant-time algorithms for comparing sensitive values, such as **passwords** and **tokens**, to prevent **timing attacks**.

- **Example**: Use **hash_equals()** when comparing sensitive values (e.g., API tokens or session tokens) to avoid revealing information about the length of these values via timing differences.

---

19. Prevent Mass Assignment Vulnerabilities

Mass assignment vulnerabilities occur when malicious users submit unexpected fields in forms to update data they should not be able to change.

- **Example**: Use **Laravel's guarded or fillable** properties to define which fields can be mass-assigned.

  ```
  php
  Copy code
  protected $fillable = ['name', 'email'];
  ```

---

20. Secure Deserialization

Improper handling of serialized data can lead to remote code execution attacks. Laravel provides **secure serialization mechanisms** that avoid this.

- **Example**: When using Laravel's serialization tools (e.g., Eloquent models), make sure only trusted sources can interact with serialized data.

---

21. Database Encryption

Laravel provides built-in **encryption utilities** that allow you to encrypt sensitive data in the database. It's useful for ensuring that data such as CHD remains protected.

- **Example**: Use Laravel's `Crypt` class to encrypt sensitive data before storing it in the database.

  ```
  php
  Copy code
  $encrypted = Crypt::encryptString($value);
  ```

---

22. Protecting Against Session Fixation

Laravel handles session fixation by regenerating session IDs after authentication. Make sure to call `session()->regenerate()` after a user logs in to prevent session fixation attacks.

---

23. Limiting Resource Access

Use **Laravel Policies** or **Gates** to ensure fine-grained access control on resources based on user roles or specific conditions.

- **Example**: Define policies to restrict access to sensitive API actions like payment processing or administrative tasks.

---

24. Security Headers Configuration

Laravel allows you to set **security headers** to further protect against attacks like **clickjacking** and **XSS**.

- **Example**: Use middleware or server configuration to add headers like `X-Frame-Options`, `X-Content-Type-Options`, and `Content-Security-Policy` to your API responses.

---

25. Preventing JSON Vulnerabilities

When returning JSON responses from the API, ensure they are properly escaped to prevent JSON vulnerabilities.

- **Example**: Use **Laravel's `response()->json()`** method to return JSON responses securely.

---

26. Periodic Security Audits

It is important to perform **regular security audits** using Laravel **Enlightn** and other security tools to assess vulnerabilities and ensure continuous compliance with best practices.

---

27. Environment Variable Protection

Ensure that sensitive environment variables, such as database credentials and API keys, are stored securely and not exposed to unauthorized access.

- **Example**: Never commit the `.env` file to version control, and restrict access to it on production servers.

Final Recommendations for the Laravel API Team

1. **Regular Audits**: Run **Laravel Enlightn** and **composer audit** regularly to identify vulnerabilities.
2. **Use HTTPS**: Force HTTPS on all sensitive API routes.
3. **Implement Rate Limiting**: Prevent brute force and denial-of-service attacks using Laravel’s rate-limiting features.
4. **Secure Logs**: Ensure logs are stored securely and rotated regularly.
5. **Train the Development Team**: Regularly train developers on secure coding practices, focusing on Laravel’s security tools and features.
6. **Stay Updated**: Subscribe to **Laravel security advisories** and install patches promptly.

https://www.laravel-enlightn.com/docs/security