Laravel Security

> It's easy to make the assumption that a robust framework like Laravel is secure, but most of the time, it's the **little things** which expose vulnerabilities in your apps.

Recent News

1. Australian telecommunications company Optus was recently hacked due to allegedly leaving an unauthenticated API endpoint exposed "***with the assumption that the API would only be used by authorised company systems.***"

2. PortSwigger discovered a vulnerability in a Mastodon fork that allowed them to **steal user passwords**, caused by a weak Content Security Policy (CSP) and flexible limitations on user inputs: "***The form-action directive could prevent these sorts of attacks***".

3. Fortbridge discovered the REST API in Plesk was lacking adequate Cross-Site Request Forgery (CSRF) protection, which allowed them to craft custom attacks that affect "***all the POST requests and we could abuse most of the APIs with it***".

- Course Outline
- PHP Security Links

Day 1

- Introduction to Application Security
  - Importance and business impact

    #### Rising Cybersecurity Threats

    2024 has seen a variety of significant cybersecurity threats. One major trend is the increase in identity-based attacks, which have surged with the help of generative AI. Techniques like phishing, social engineering, and buying legitimate credentials from access brokers have become more prevalent. Cloud intrusions have also spiked, with adversaries using valid credentials to access and manipulate cloud environments, making it difficult to distinguish between normal activity and a breach (CrowdStrike).

Cost of Security Breaches

The financial impact of security breaches can be staggering. In 2023, the cost of an average data breach was $4.45 million. Data breaches are especially more common in businesses with over 25 employees. When a company employs its 25th employee, the chance of being affected by a data breach nearly triples, and the risk increases further for companies with more than 50 employees (Tech.co).

Reputation and Trust

The data breaches in 2024 have included high-profile incidents affecting a wide range of companies, from biotech firms like 23andMe to tech giants like SONY and service providers like Duolingo. These incidents highlight the significant impact on reputation and customer trust that a data breach can have. For instance, the breach of 23andMe's customer accounts led to the theft of sensitive genetic data, underlining the severe privacy implications of such breaches (Tech.co).

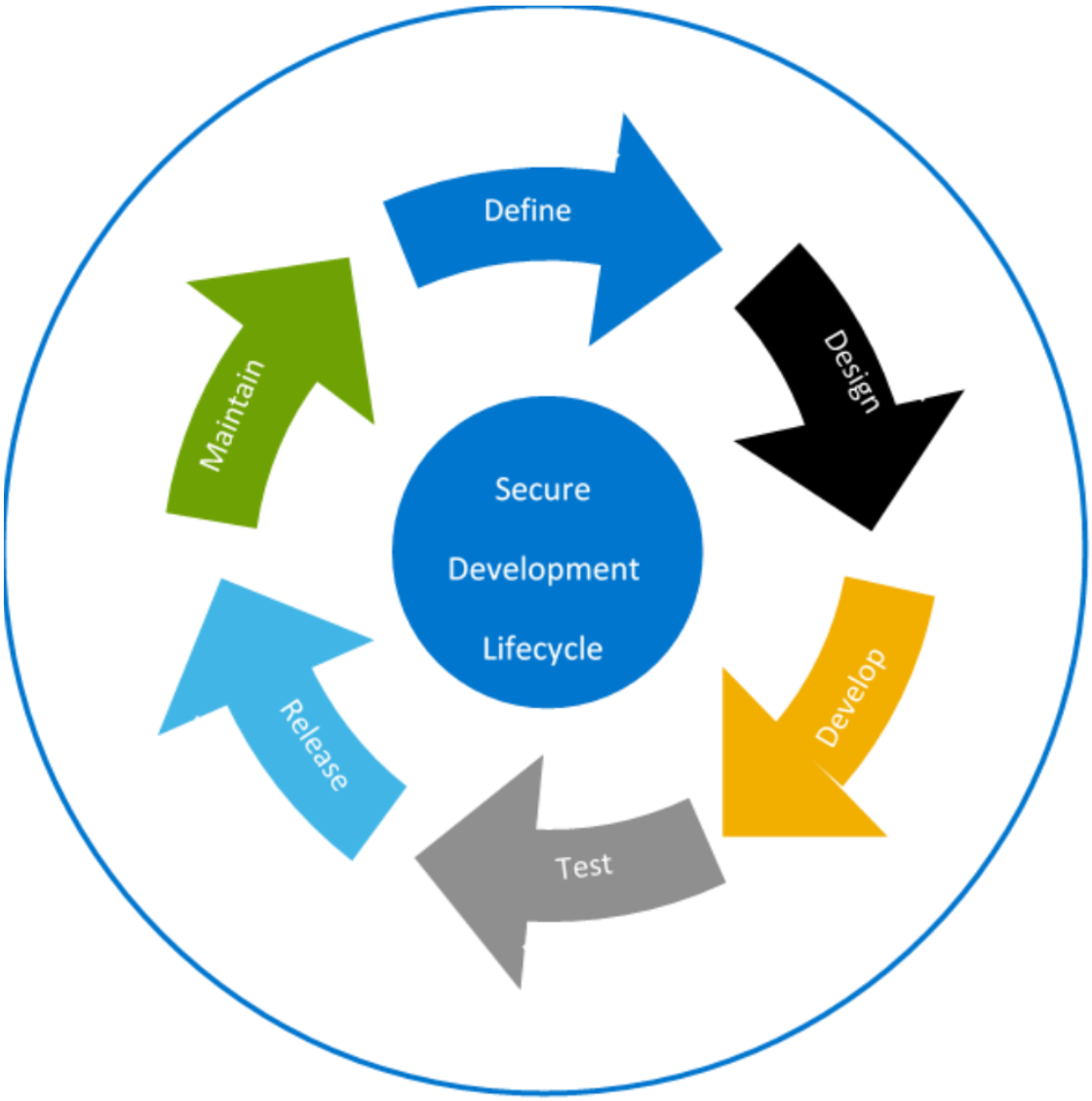
Regulatory and Compliance Requirements

While I don't have specific recent updates on regulatory changes, it's important to continuously emphasize the relevance of laws like GDPR and HIPAA. These regulations mandate stringent data protection and privacy measures, and non-compliance can result in hefty fines and legal repercussions. Keeping abreast of changes and interpretations in these laws is crucial for businesses to avoid legal risks.

These points can provide a contemporary and realistic context for your course, emphasizing the urgency and relevance of robust PHP security practices in today's digital landscape.

- Security Development Lifecycle
  The Security Development Lifecycle (SDL) is a process that helps developers build more secure software and address security compliance requirements while reducing development costs. The concept was pioneered by Microsoft in the early 2000s as a way of responding to increasing

concerns over software security. Here are the key aspects of the SDL:



1. Training

- **Purpose:** Educate and train developers and management in secure coding practices and the importance of security.
- **Implementation:** Regular training sessions, workshops, and access to security-related resources.

2. Requirements

- **Purpose:** Define and integrate security and privacy requirements into the development process.
- **Implementation:** Establishing security goals, documenting security controls, and considering legal/regulatory requirements.

3. Design

- **Purpose:** Assess and mitigate risks in software design.

- **Implementation**: Threat modeling (systematically identifying potential threats), architectural risk analysis, and design reviews.

4. Implementation

- **Purpose**: Write secure code and address common security flaws.
- **Implementation**: Using secure coding standards, static analysis tools to check code for vulnerabilities, and code reviews.

5. Verification

- **Purpose**: Ensure that security requirements are met and identify any security weaknesses.
- **Implementation**: Security testing (like fuzzing and penetration testing), dynamic analysis, and code reviews.

6. Release

- **Purpose**: Prepare for release with final security reviews and create response plans for potential security issues.
- **Implementation**: Final security review, creation of incident response plans, and final risk assessment.

7. Response

- **Purpose**: Respond to and manage security breaches in a timely and efficient manner.
- **Implementation**: Incident response plan execution, patch management, and ongoing security updates.

Benefits of SDL

1. **Reduces Vulnerabilities**: Identifies potential security issues early in the development process.
2. **Compliance**: Helps in meeting legal and regulatory compliance requirements for security.
3. **Cost Efficiency**: Preventing security issues in the development phase is cheaper than fixing them post-release.
4. **Reputation and Trust**: Enhances user trust by demonstrating commitment to security.

Implementing SDL

- **Customization**: The SDL process should be adapted to fit the specific needs and context of the organization and its projects.

- **Integration with Existing Processes:** It should be integrated with existing development workflows.
- **Continuous Improvement:** Regularly update the process based on new threats, technologies, and best practices.

SDL is an ongoing process; it's not just about fixing bugs. It's about creating a culture of security awareness and continuous improvement within the organization. By adopting SDL, developers and companies can significantly enhance the security and robustness of their software products.

- Threat Modelling and Risk Analysis

1. Identifying Potential Threats

- **Concepts:**
  - **Understanding Threats:** Discuss common threats in web applications, such as SQL injection, XSS, CSRF, session hijacking, and file inclusion vulnerabilities.
  - **PHP-Specific Threats:** Highlight threats particularly relevant to PHP applications, like remote code execution and PHP injection.
  - **Real-World Examples:** Use case studies or real incidents to illustrate how threats manifest in PHP environments.
- **Activity:**

▶ **Threat Identification Exercise**

Exercise Overview

Task Description

Participants will be divided into small groups, each focusing on a high-level component of a PHP application. Instead of delving into code, groups will identify potential security threats, link them to real-life incidents or news stories, and discuss broader implications and preventive strategies.

Objectives

- Connect theoretical vulnerabilities to real-world incidents to underscore their practical significance.
- Foster an understanding of the broader impact of these security threats on businesses and users.
- Encourage knowledge sharing from personal experiences and current events.

Component 1: User Authentication System

Prompt

- Reflect on notable security breaches or incidents involving user authentication failures in web applications. Consider issues like compromised passwords, two-factor authentication failures, or security lapses leading to data breaches.
- Discuss the broader consequences of these incidents, such as reputation damage, financial loss, or legal implications.
- Suggest high-level strategies for strengthening authentication systems, like adopting industry best practices or new technologies.

Component 2: Database Interactions

Prompt

- Examine major incidents reported in the news where database security was compromised in web applications, focusing on cases like data leaks due to SQL Injection or improper access controls.
- Analyze the impact of these incidents from a business and user trust perspective.
- Propose organizational measures and policies to enhance database security, such as regular audits, adopting a “least privilege” approach, or employee training programs.

Component 3: File Upload and Management

Prompt

- Investigate real-life examples where insecure file upload and management led to security breaches, such as the uploading of malicious files or unauthorized access to sensitive documents.
- Discuss how these breaches could have been prevented and the aftermath in terms of regulatory compliance, customer trust, and financial repercussions.
- Develop a high-level strategy for secure file handling, focusing on policy, education, and the use of advanced monitoring tools.

Deliverables for Each Group

- A summary of real-life incidents related to their component, with an analysis of the causes and effects.
- A discussion of the broader implications of these security threats.

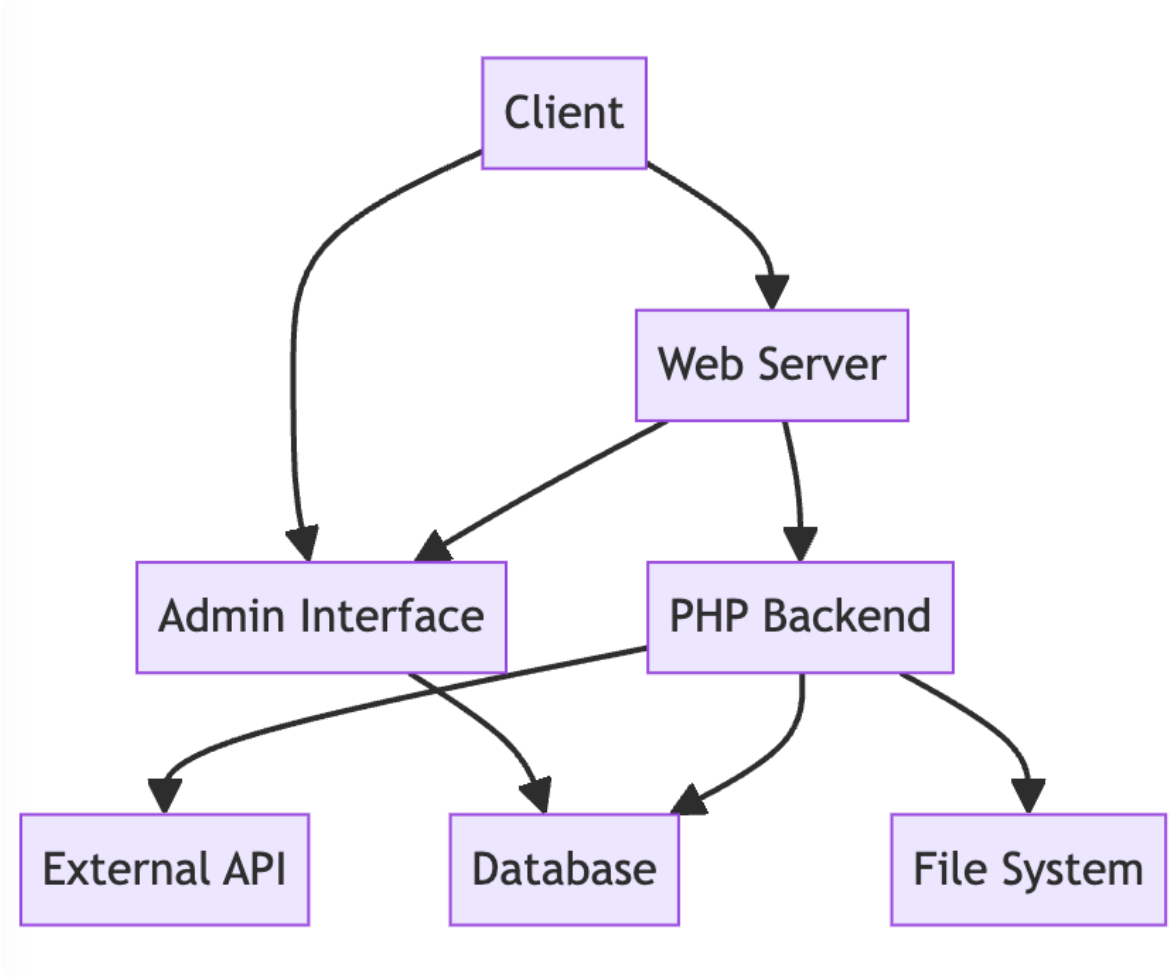
- High-level recommendations for preventing similar incidents.

Concluding the Exercise

- Groups present their findings, focusing on the linkage between real-world incidents and theoretical vulnerabilities.
- Facilitate a discussion on how awareness of these incidents can shape better security practices.
- Highlight the importance of staying informed about recent security breaches and trends as a proactive measure.

2. Attack Surface Analysis - **Concepts**: - **Defining Attack Surface**: Explain the attack surface as all the points where an attacker can try to enter data to or extract data from an environment. - **Minimizing Attack Surface**: Discuss practices to reduce the attack surface, such as minimizing code complexity, limiting user permissions, and disabling unnecessary services.

- **Activity**:
  - **Attack Surface Mapping**: Provide participants with a simplified architecture of a PHP application. Ask them to identify and mark potential attack vectors and discuss how to minimize these in a group discussion.



3. Analyzing Security and Privacy Risks

- Concepts:
  - **Risk Analysis Principles**: Introduce the concepts of likelihood and impact in the context of security risks.
  - **Privacy Concerns**: Address privacy issues in PHP applications, particularly concerning user data handling and GDPR compliance.
  - **Risk Mitigation Strategies**: Discuss strategies to mitigate risks, such as input validation, output encoding, and the use of security headers.
- Activity:
  - **Risk Analysis Workshop**: Give each group a list of potential vulnerabilities. Ask them to rate each vulnerability based on its likelihood and impact, then develop a mitigation plan.
  - ▶ List of vulnerabilities

1. **SQL Injection**: Exploiting vulnerabilities in SQL queries to manipulate databases.
2. **Cross-Site Scripting (XSS)**: Injecting malicious scripts into webpages viewed by other users.

3. **Cross-Site Request Forgery (CSRF)**: Tricking a user into submitting a malicious request.
4. **Remote Code Execution**: Enabling an attacker to execute arbitrary code on the server.
5. **File Upload Vulnerabilities**: Allowing the execution of malicious files uploaded by attackers.
6. **Insecure Direct Object References (IDOR)**: Accessing unauthorized data by modifying a parameter value (like changing the ID in the URL).
7. **Session Hijacking and Session Fixation**: Exploiting session management weaknesses to take over a user's session.
8. **Directory Traversal/Path Traversal**: Accessing files and directories outside the intended folder structure.
9. **Broken Authentication**: Implementing flawed authentication mechanisms leading to unauthorized access.
10. **Sensitive Data Exposure**: Inadequately protecting sensitive data like passwords, credit card numbers, or personal information.
11. **Security Misconfiguration**: Default configurations or incomplete setups leading to vulnerabilities.
12. **XML External Entities (XXE)**: Exploiting XML processors to perform hostile actions related to an external server.
13. **Buffer Overflow**: Writing data beyond the allocated memory space, potentially leading to code execution or crashing the system.
14. **Unvalidated Redirects and Forwards**: Redirecting users to phishing or malware sites without proper validation.
15. **Insecure Deserialization**: Exploiting the deserialization process to execute arbitrary code, bypass authentication, or perform other malicious activities.
16. **Use of Components with Known Vulnerabilities**: Utilizing libraries, frameworks, or other software modules that are known to be vulnerable.
17. **Insufficient Logging and Monitoring**: Failing to record events sufficiently, hindering the detection of malicious activity.
18. **Exposed APIs**: Unsecured APIs allowing unauthorized access to sensitive data or functionality.

For each of these vulnerabilities, participants can discuss the likelihood of occurrence and potential impact on the application or organization. They should then propose

mitigation strategies, which could include technical fixes, best practices, policy changes, or user education. This exercise will help participants understand the variety of risks involved in web application security and the importance of a comprehensive approach to mitigating these risks.

Additional Workshop Ideas

- **Interactive Quizzes**: Use tools like Kahoot! to create an interactive quiz about PHP security. This can be a fun way to test and reinforce the knowledge gained.
- **Case Study Analysis**: Present a case study of a PHP security breach and have participants analyze what went wrong and how it could have been prevented.
- **Live Coding/Code Review**: Perform a live coding session to demonstrate secure coding practices in PHP or conduct a code review of a sample PHP script to identify and fix security flaws.

General Tips for the Workshop

- **Engagement**: Encourage participation by asking direct questions, encouraging discussion, and providing real-world examples.
- **Breakout Rooms**: Utilize Zoom's breakout room feature for group activities, allowing for smaller group discussions and collaboration.
- **Resources**: Provide supplementary materials, like cheat sheets or reference guides, that participants can use during and after the workshop.

By integrating these concepts and activities, your workshop will not only be informative but also interactive and engaging, helping participants better understand the importance of security in PHP development.

- Input Validation

Objective

Learn to implement effective input validation in a Laravel application to enhance security and data integrity.

Prerequisites

- Basic PHP knowledge.
- Familiarity with Laravel framework.
- Understanding of HTML forms.

Exercise Breakdown

Handling Query Parameters and Form Fields

Objective: Learn to validate query parameters and form fields.

Create Routes and Controllers

```
// In routes/web.php
Route::get('/input-validation', 'ValidationController@showForm');
Route::post('/input-validation',
'ValidationController@processForm');
```

Implement Controller Methods

```
// In app/Http/Controllers/ValidationController.php
public function showForm()
{
    return view('input-validation-form');
}

public function processForm(Request $request)
{
    $validatedData = $request->validate([
        'name' => 'required|max:255',
        'age' => 'required|numeric',
    ]);

    // Process the data
    return redirect('home')->with('status', 'Form processed
successfully!');
}
```

Create a View

```
<!-- In resources/views/input-validation-form.blade.php -->
<form action="/input-validation" method="POST">
    @csrf
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>
    <label for="age">Age:</label>
    <input type="number" id="age" name="age" required>
    <button type="submit">Submit</button>
</form>
```

Data Existence Checks

Objective: Implement and discuss data presence validation techniques.

Using `isset()` and Null Coalescing Operator `??`

```
$name = isset($request->name) ? $request->name : 'Default Name';
// OR with ?? operator
$name = $request->name ?? 'Default Name';
```

Using `is_*()` Functions

```
if (is_string($request->name)) {
    // Process string
}
```

String Searching

Objective: Implement string searching functions.

Using `strpos()` and Related Functions

```
if (strpos($request->input('content'), 'example') !== false) {
    // 'example' is in 'content'
}
```

Searching Patterns with Regular Expressions

Objective: Learn to use regular expressions for input validation.

Regex in Laravel Validation

```
$validatedData = $request->validate([
    'email' => 'required|regex:/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.
([a-z\.]{2,6})$/'
]);
```

Using ctype Functions

Objective: Utilize ctype functions to validate data types.

Example Using `ctype_digit()`

```
if (ctype_digit($request->age)) {
    // Age is numeric
```

```
}
```

Filter Functions

Objective: Explore PHP's filter functions and compare with Laravel's techniques.

Using PHP Filter Functions

```
$age = filter_var($request->age, FILTER_VALIDATE_INT);
```

PHP 7+ Type Declarations

Objective: Understand and use strict type declarations.

Enforce Strict Types

```
<?php
declare(strict_types=1);

function addNumbers(int $a, int $b): int {
    return $a + $b;
}
```

Exercise

You have a choice of two:

A follow-up exercise after walking through the basic input validation techniques would ideally build on what the participants have learned, allowing them to apply these concepts in a more complex or nuanced scenario. Here are a few suggestions for effective follow-up exercises that could be both challenging and educational:

1. Create a New Form with Complex Validation Rules

Objective: Create a new form that collects more detailed information, requiring them to implement complex validation rules.

Details:

- **Form Requirements**: Create a form for event registration that includes fields like name, email, date of birth, event date selection, and dietary preferences.
- **Validation Challenges**:
  - Ensure that the email is in a proper format and unique (if storing in a database).
  - Validate that the date of birth indicates they are over a certain age (e.g., 18 years old).
  - Event date must be in the future.
  - Dietary preferences should be selected from a predefined list.

Learning Outcome: This exercise teaches how to validate various data types and handle more complex logic like dates and lists.

2. Enhance the Existing Form with Additional Features

Objective: Add new features to the existing form, such as file uploads, and implement relevant validations.

Details:
- **New Features**: Allow users to upload a profile picture and a resume.
- **Validation Challenges**:
  - Ensure the profile picture is in an image format (e.g., jpg, png) and does not exceed a size limit.
  - Validate that the resume is a PDF and also respects a size limit.

- XSS

Cross-site scripting

- Not CSS because that’s already in use
- Stealing the cookie
- POC
- The developer stored and ran the code
- Same-origin Policy
  - JS have only access to code from the same origin
  - Origin: Protocol/Domain/Port
- Consequences …
  - Cookie theft
  - DOM manipulation

        - Keylogger
        - Browser exploits
        - Anything JavaScript can do!
    - Types of XSS
        - Stored or Persistent XSS - it’s in the DB
        - Reflected XSS - uses query parameters
        - DOM-based XSS (rarest) - doesn’t use the server, all based in a SPA
    - Defend
        - Filtering
            - Might destroy original data
            - A black list mightn’t be extensive Enough
            - strip_tags() - doesn’t help with attributes
            - filter_var($s, FILTER_SANITIZE_STRING)
            - preg_replace()
        - Escaping the output!
            - Pretty easy in Laravel - default `{{}}` is escaping
            - PHP has a built in method `htmlspecialchars()` which will swap `<` to `$lt;`, etc.
            - You can also set `ENT_QUOTES` which will also handle other quotes
            - htmlentitites() also exists
            - Only works on the server … so for our dom xss

Day 2

- Security headers

Workshop Title:

Securing Web Applications: A Hands-On Workshop on HTTP Header Best Practices

Objective:

Participants will learn to implement secure HTTP headers to protect web applications from common vulnerabilities and attacks.

Workshop Outline:

Part 1: Introduction to HTTP Headers (30 minutes)

- **Presentation:** Overview of HTTP headers and their role in web security.
- **Discussion:** Common security vulnerabilities related to HTTP headers.
    - What is a header?
    - Auth
    -

Part 2: Deep Dive into Security Headers (1 hour)

For Part 2 of your workshop on HTTP headers focusing on a deep dive into security headers, here are detailed insights and structures that you might find useful:

Part 2: Deep Dive into Security Headers (1 hour)

Overview

This section intensifies the focus on specific security-related HTTP headers, leveraging the OWASP Security Headers Cheat Sheet. It's structured to enhance understanding through a mix of theoretical explanations, practical examples, and interactive discussions.

Topics and Structure

1. **Content Security Policy (CSP)**
    - **Definition:** Explain that CSP is used to specify which dynamic resources are allowed to load, thereby preventing a wide range of attacks, including Cross-Site Scripting (XSS) and data injection.
    - **Implementation:** Show how to properly configure CSP, including directives like `default-src`, `script-src`, `img-src`, etc. CSP directives
    - **Examples:** Compare correct versus incorrect CSP implementations and the potential vulnerabilities of weak policies.
    - Good:

```
Content-Security-Policy: default-src 'self'; script-src 'self'
https://api.example.com; img-src 'self'
https://images.example.com;
```

This configuration restricts all sources by default to the same origin (`self`), allows scripts only from the same origin and a trusted API, and permits images from the same origin and a specific trusted host.
- Bad:

```
Content-Security-Policy: default-src *; script-src *; img-src *;
```

This configuration essentially allows loading resources from any origin, which defeats the purpose of CSP and exposes the site to XSS and data injection attacks.

2. **X-Frame-Options**
   - **Definition:** This header helps protect against clickjacking attacks by instructing the browser whether the content can be displayed within frames. The `X-Frame-Options` HTTP header is a security feature that helps to protect websites against 'clickjacking' attacks. Clickjacking involves an attacker tricking a user into clicking a webpage element that is invisible or disguised as another element. This can result in revealing confidential information or taking control of their computer while clicking seemingly innocent web pages. The `X-Frame-Options` header can instruct the browser to prevent any site from framing the content (i.e., placing it inside an iframe or frame). Here are the values it can take:
   1. `DENY`
   - **Description:** The `DENY` value completely prohibits any domain from framing the content. This is the most secure setting, ensuring that no external or internal page can frame the content.
   - **When to Use:**
     - Use `DENY` when you want the maximum level of security and do not need any of your pages to be framed by any other page, even your own pages on the same domain.
   2. `SAMEORIGIN`
   - **Description:** The `SAMEORIGIN` value allows only the same origin as the website itself to frame the content. This means that the page can only be framed by other pages from the same domain.

- **When to Use:**
  - Use `SAMEORIGIN` when your site needs to frame its own pages (such as for layout purposes or internal widgets) but should not be framed by other sites. It strikes a balance between functionality and security, protecting against external misuse while allowing internal use.

3. `ALLOW-FROM` (Deprecated)
   - **Description:** The `ALLOW-FROM` value allows framing the content only by specified domains. This directive is now deprecated in favor of the `frame-ancestors` directive in Content Security Policy (CSP), which provides similar functionality but with better support and flexibility.
   - **When to Use:**
     - Traditionally, `ALLOW-FROM` was used to permit specific external sites to frame the content, useful in cases where integration with known and trusted partners or services was necessary. However, its support is inconsistent across browsers, and it's generally recommended to use CSP's `frame-ancestors` directive instead for more reliable and granular control.

Good Configuration:

`X-Frame-Options: SAMEORIGIN`

- **Explanation:** This header setting prevents the webpage from being displayed in a frame, iframe, or object from a different origin. It's a strong defense against clickjacking attacks.

Bad Configuration:

`X-Frame-Options: ALLOW-FROM https://example.com`

- **Explanation:** Using `ALLOW-FROM` can be risky if not implemented carefully. It's less secure because it allows framing from specified domains, which could be manipulated or spoofed.

3. **X-Content-Type-Options**
   - **Definition:** This header prevents the browser from interpreting files as a different MIME type to what is specified in the Content-Type HTTP header (MIME type sniffing).

- **Importance:** Explain how MIME type sniffing can be exploited to deliver attacks and why enforcing this header can mitigate such risks.
  Good Configuration:

`X-Content-Type-Options: nosniff`

- **Explanation:** This configuration prevents the browser from performing MIME type sniffing. It forces the browser to stick to the declared content-type delivered by the server.

Bad Configuration:

`X-Content-Type-Options: sniff`

- **Explanation:** There is actually no 'sniff' option for this header. Leaving it unset or misconfigured allows MIME type sniffing, which can lead to security vulnerabilities.

4. **Strict-Transport-Security (HSTS)**
   - **Definition:** HSTS enforces secure (HTTP over SSL/TLS) connections to the server.
   - **Configuration:** Discuss max-age and the inclusion of subdomains with `includeSubDomains`, and the effect of `preload`.
   - **Security Benefits:** Highlight the protection this offers against common attacks like man-in-the-middle (MITM).
   - **Examples:** Case studies of HSTS implementation and the consequences of not using it.
5. **Other Security Headers** (as time allows)
   - **Referrer-Policy:** Governs which referrer information sent along with requests.
   - **Feature-Policy:** Allows developers to selectively enable and disable use of various browser features and APIs.
   - **Expect-CT:** Enforces Certificate Transparency requirements.
   - **Permissions-Policy:** A newer header that controls access to certain APIs or features.

Part 3: Hands-On Exercises (1.5 hours)

- **Setup:** Participants set up their environments with access to a test web application.

- **Guided Activity**: Implementing various headers on their web applications.
  - Each participant will add headers to the web application, test the changes, and discuss the outcomes.
  - Use tools like web browsers' developer tools to analyze header effectiveness.

Part 4: Advanced Topics and Best Practices (30 minutes)

- **Lecture**: Discuss advanced security headers and upcoming standards.
- **Best Practices**: Summarize best practices for implementing and testing HTTP headers.

Part 5: Q&A and Wrap-Up (30 minutes)

- Open floor for questions.
- Discuss real-world scenarios where these headers prevented security breaches.
- Provide additional resources for further learning (links to OWASP documentation, blogs, etc.).

Part 6: Optional Follow-Up Activities

- Provide a follow-up email with additional resources and a feedback survey.
- Offer certificates or badges for completion.

Delivery Tips:

- **Interactive and Engaging**: Encourage questions and interactions among participants throughout the workshop.
- **Real-World Examples**: Use case studies and examples from recent security incidents to highlight the importance of secure HTTP header configurations.
- **Practical Implementation**: Ensure that the hands-on part of the workshop allows participants to practically implement headers and see real-time results on security scans or assessments.

Applying HTTP header security practices to a Laravel application is a vital step in enhancing the security posture of your web application. Laravel provides several ways to set and configure HTTP headers, either globally or on a per-response basis. Here's a step-by-step guide on how you can apply some of the key security headers using Laravel:

1. Content Security Policy (CSP)

A Content Security Policy helps prevent Cross-Site Scripting (XSS) and data injection attacks. You can add a CSP header in Laravel by modifying middleware.

Creating a Middleware for CSP:

```
php artisan make:middleware SecurityHeaders
```

In the created middleware (`app/Http/Middleware/SecurityHeaders.php`), you can set the CSP headers like so:

```
public function handle($request, Closure $next)
{
    $response = $next($request);

    $csp = "default-src 'self'; script-src 'self' 'unsafe-inline';
img-src 'self'; style-src 'self' 'unsafe-inline';";
    $response->headers->set('Content-Security-Policy', $csp);

    return $response;
}
```

2. X-Frame-Options

This header can prevent your site content from being embedded in other sites (clickjacking protection).

Adding X-Frame-Options:

You can add this header using the same middleware method shown above:

```
$response->headers->set('X-Frame-Options', 'DENY');
```

3. X-Content-Type-Options

This header prevents MIME type sniffing which can reduce exposure to drive-by download attacks.

Setting X-Content-Type-Options:

In the same middleware:

```
$response->headers->set('X-Content-Type-Options', 'nosniff');
```

4. Strict-Transport-Security (HSTS)

HSTS is crucial for HTTPS websites to ensure client interactions are conducted over secure connections.

Implementing HSTS:

```
        $response->headers->set('Strict-Transport-Security', 'max-age=31536000; includeSubDomains');
```

5. Register the Middleware

After setting up the middleware, you need to register it so it's executed with every request:

Global Middleware:

Add your new middleware to the global middleware stack in `app/Http/Kernel.php`:

```
protected $middleware =
    // Other middleware...
    \App\Http\Middleware\SecurityHeaders  class,

 ;
```

6. Testing Headers

After deploying these headers, it's important to test them:

- **Use Browser DevTools**: Check the network tab to see if the headers are applied correctly.
- **Online Tools**: Use tools like SecurityHeaders.com to analyze your website's headers and get a report on potential problems or missing headers.

7. Regularly Update Your Practices

Security standards evolve, so regularly review and update your security headers and practices. Follow Laravel updates and security patches to keep your application secure.

By implementing these headers in your Laravel application, you enhance its security against a variety of common web vulnerabilities. Always ensure to test your application thoroughly after making these changes to avoid inadvertently breaking functionality.

Security Headers Exercise

- Incident Response and Logging

Workshop Segment Title:

Essentials of Incident Response and Logging

Objective:

To provide participants with a focused overview of key logging practices and a basic framework for incident response.

Target Audience:

IT security teams, system administrators, and developers involved in maintaining system security.

Duration:

1-2 hours

Materials Needed:

- Computers with internet access
- Access to a basic logging tool or simulated logs
- Presentation equipment for slides and demonstrations

Workshop Segment Outline:

Part 1: Introduction to Incident Response and Logging (15 minutes)

- **Brief Overview**: Importance of logging and its role in incident response. Logging not only facilitates the tracking of what happens within an application or system but is also pivotal in the detection, diagnosis, and resolution of incidents. With a focus on IT security, understanding logging practices and frameworks, like those defined in the PHP Standards Recommendation (PSR), is crucial for professionals in this field.
- **Key Concepts**: Quick definitions and purposes of critical logging terms and incident response phases.

- Logging: The process of recording data about the operations of a system, network, or application. Logs serve as a factual reference that helps in troubleshooting, security monitoring, and forensic analysis.
- Log levels: Defined in the PSR-3 standard, log levels are critical in categorizing the severity of the data being logged. These levels help in filtering and managing logs more effectively. The levels include:
    1. **Emergency**: System is unusable.
    2. **Alert**: Action must be taken immediately.
    3. **Critical**: Critical conditions.
    4. **Error**: Error conditions.
    5. **Warning**: Warning conditions.
    6. **Notice**: Normal but significant condition.
    7. **Info**: Informational messages.
    8. **Debug**: Debug-level messages.
- Log stacks: In PHP, a log stack refers to a multi-layer logging system where logs can be sent to multiple destinations or handled by multiple loggers. This is particularly useful in complex environments where different systems or stakeholders might need access to specific types of logs (e.g., security logs, application performance logs, transaction logs).
- Incident response: A structured methodology for handling security breaches, attacks, or incidents. The aim is to manage the situation in a way that limits damage and reduces recovery time and costs.

Discussion Points:

- How do different log levels impact the way we handle and prioritize incident responses?
- What advantages do log stacks provide in managing logs across different systems or applications?

Part 2: What to Log and Logging Best Practices (20 minutes

Group activity:

Prompt: What should we log?

- Gather together a list of the key types of data that should be logged.
- To what level of granularity should each type be logged?

- How long should you retain each level?

Possible solution

1. **User Authentication Logs**
   - **What to Log**: Successful and failed login attempts, logout events, and password change requests.
   - **Granularity**: Include user identifiers, timestamps, source IP addresses, and outcomes (success or failure).
   - **Retention**: Typically, retain these logs for 6-12 months depending on legal or regulatory requirements.
2. **Access Logs**
   - **What to Log**: Requests to access sensitive data or administrative areas, changes to user permissions, and access to API endpoints.
   - **Granularity**: Record the user ID, timestamp, accessed URLs, HTTP method, status code, and any modifications made.
   - **Retention**: Retain for a minimum of one year for audit purposes, or longer if required by compliance regulations.
3. **System and Application Errors**
   - **What to Log**: Exceptions, error messages, stack traces, and context information about the state of the application when the error occurred.
   - **Granularity**: Include detailed error descriptions, affected components or services, user sessions, and relevant system metrics at the time of the error.
   - **Retention**: Keep error logs for at least 3-6 months for troubleshooting and root cause analysis.
4. **Performance Metrics**
   - **What to Log**: Response times, server CPU usage, memory usage, database query times, and page load times.
   - **Granularity**: Log detailed metrics that can help in identifying performance bottlenecks and patterns over time.
   - **Retention**: Short-term retention (about 1-3 months) is typically sufficient, unless historical performance data is needed for long-term analysis.
5. **Network Traffic**
   - **What to Log**: Incoming and outgoing traffic data, including the amount of data, connection times, and traffic sources/destinations.

- **Granularity**: Record timestamps, IP addresses, port numbers, protocols used, and traffic volume.
- **Retention**: Maintain these logs for at least 6 months to assist in network performance monitoring and troubleshooting.

6. **Changes to Application Configuration**
   - **What to Log**: Any changes made to the application settings or configurations, including updates to environment variables or deployment configurations.
   - **Granularity**: Include details about what was changed, who made the change, and when the change was made.
   - **Retention**: Retain these logs for the lifetime of the application to provide a historical record of changes.
7. **API Usage Logs**
   - **What to Log**: Details of API calls made, especially those involving data manipulation or sensitive data access.
   - **Granularity**: Log the API endpoint, caller identity, parameters passed, timestamp, and the outcome of the call.
   - **Retention**: Typically, these should be kept for one year to support usage analysis and audit requirements.

- **Best Practices Presentation**: Discuss concise best practices:
  1. **Data Minimization**: Emphasize the importance of logging only what is necessary to achieve the intended security and monitoring purposes. Discuss the implications of excessive logging, including performance impacts and privacy concerns.
  2. **Secure Storage and Access**: Talk about the need for secure log storage solutions to prevent unauthorized access and tampering. This includes using encryption for log data at rest and in transit, as well as proper access controls.
  3. **Log Retention Policies**: Discuss the importance of defining and adhering to log retention policies that balance operational and legal requirements with storage limitations.
  4. **Real-time Analysis and Alerts**: Highlight the benefits of real-time log analysis and setting up alerts for unusual activities that could indicate a security incident.

Part 3: Setting up logs in Laravel

- The stack
- The log file
- Logging things out

Part 4: Analysing logs

- In the ‘olden days’, we used the CLI to analyse logs and now we probably should keep a tail running on our larvel log while we are developing.
- `tail -f laravel.log`
- We used tail and grep to help with making sense of logs
- Now, we tend to use 3rd party tools - either specific logging platforms that can be configured to alert when something unusual happens. Also, Cloudwatch on AWS does the same thing.

Part 5: Quick Incident Response Strategy (20 minutes)

- **Response Overview**: Outline the basic steps of an incident response strategy:
- This part of the workshop focuses on developing a quick and effective incident response strategy tailored for web applications, particularly those built with Laravel. We'll go through each step of the incident response process, outlining key actions and considerations.

Overview of Incident Response Steps:

1. **Detection**
    - **Description**: The ability to detect an incident quickly is crucial. This involves continuously monitoring systems and logs for unusual activity that could indicate a security breach.
    - **Key Actions**: Set up automated alerting systems based on predefined thresholds or anomalies in log data, such as unexpected access patterns or spikes in traffic.
    - **Tools and Practices**: Use tools like Laravel's logging capabilities integrated with monitoring solutions like New Relic or Prometheus to collect and analyze data in real time.
2. **Containment**
    - **Description**: Once an incident is detected, the immediate goal is to contain it to prevent further damage.

- **Key Actions**: Isolate affected systems or components to stop the spread of the incident. This might involve taking certain servers offline, blocking network traffic, or disabling compromised user accounts.
- **Tools and Practices**: Implement network segmentation and have pre-defined security groups or rules that can be quickly applied to restrict access.

3. **Eradication**
   - **Description**: After containment, the focus shifts to removing the cause of the incident and any traces left by the attacker.
   - **Key Actions**: Identify and remove malware, close security loopholes, and update vulnerable software. Ensure all malicious changes are reversed.
   - **Tools and Practices**: Use malware scanners and vulnerability assessment tools to identify malicious software and weaknesses. Patch management strategies are crucial here.
4. **Recovery**
   - **Description**: The recovery phase involves restoring and validating system functionality for business operations to resume.
   - **Key Actions**: Gradually restore services with careful monitoring for any signs of issues. Validate the integrity of systems and data before going back online.
   - **Tools and Practices**: Use backup and restore solutions to recover clean versions of files and databases. Continuously monitor systems during the recovery phase to ensure no remnants of the incident remain.
5. **Post-Incident Analysis**
   - **Description**: After the incident is handled, conducting a thorough review to learn from the event is essential.
   - **Key Actions**: Analyze what happened, why it happened, how well the team responded, and what could be improved. Document every step taken during the incident and the lessons learned.
   - **Tools and Practices**: Hold a debriefing meeting with all involved parties. Use tools like root cause analysis frameworks to help understand underlying issues.

Group Exercise: Developing Incident Response Protocols for Your Application (10 minutes)
Exercise Objective:

To create tailored incident response protocols that align with the specific needs and configurations of the participants' web applications.

Activity Instructions:

1. **Scenario Planning**: Consider the incident type you've been given and work to develop an incident response plan. DDoS, Data breach, Ransomware
2. **Develop Response Steps**: Utilizing the five response steps outlined, groups develop an incident response strategy specific to their scenario.
    - Discuss potential detection mechanisms that could be used.
    - Decide on containment strategies appropriate for their systems.
    - Discuss what eradication steps would be most effective.
    - Plan for a recovery process, including any tools that might assist.
    - Outline a post-incident analysis approach to capture lessons learned.

Possible solution

For your workshop focusing on incident response for web applications, here are three top incident types to consider, each presenting unique challenges and requiring specific response strategies:

1. **DDoS (Distributed Denial of Service) Attack**
    - **Description**: This type of attack involves overwhelming a web application's resources (e.g., bandwidth, server capacity) by flooding it with excessive requests from multiple compromised computer systems. The goal is to render the website or service inoperable to legitimate users.
    - **Impact**: Can cause significant downtime, loss of customer trust, and financial losses due to interrupted service.

- **Response Strategy**: Implement rate limiting, use DDoS mitigation services like Cloudflare or AWS Shield, and configure network hardware to manage and absorb traffic spikes.

2. **Data Breach**
   - **Description**: Unauthorized access to confidential data stored by the application. This can occur through various means such as exploiting software vulnerabilities, phishing attacks, or insufficient access controls.
   - **Impact**: Leads to the exposure of sensitive user data (e.g., personal information, credit card numbers), potentially resulting in identity theft for users and legal repercussions for the company.
   - **Response Strategy**: Quickly isolate affected systems, assess the scope of the breach, notify affected users and legal authorities as required, and enhance security measures to prevent future incidents.
3. **Ransomware Attack**
   - **Description**: A type of malware that encrypts data on infected systems, making it inaccessible until a ransom is paid. Ransomware can be introduced through malicious downloads, email attachments, or compromised websites.
   - **Impact**: Causes data loss and significant operational disruption. Even if the ransom is paid, there's no guarantee that the data will be fully recovered.
   - **Response Strategy**: Maintain regular, secure backups of all critical data to restore systems with minimal downtime. Employ anti-malware solutions, educate users on phishing, and implement strict access controls.

Each scenario demands different detection techniques, preventative measures, and responses to mitigate damage effectively:

- For **DDoS attacks**, the focus might be on infrastructural resilience and traffic filtering.
- In **data breaches**, the emphasis would be on rapid detection, data protection practices, and compliance with data protection regulations.
- With **ransomware**, emphasis should be placed on preventative security practices, rapid isolation of affected systems,

and robust data recovery processes.

Part 6: Q&A and Wrap-Up (15 minutes)

- **Discussion**: Encourage final questions and discussions about implementing what they've learned.
- **Resource Sharing**: Provide a handout or digital resource list for further learning.

Key Learning Points:

- Understand essential logging data to improve incident detection and response.
- Learn practical skills for setting up and analyzing logs with common tools.
- Gain insight into forming a basic but effective incident response strategy.

- What to log
- What to look for

- ~~Dynamic/Fuzz testing~~
- Static Analysis

Workshop Segment Title:

Integrating Static Analysis with PHPStan and LaravelStan

Objective:

To educate participants on the benefits and methodologies of static code analysis, and to provide hands-on experience in configuring and using PHPStan/LaravelStan in Laravel applications.

Target Audience:

PHP developers, Laravel developers, software quality assurance professionals, and anyone involved in the development process of PHP applications.

Duration:

1-2 hours

Materials Needed:

- Computers with PHP and Laravel environment set up
- Internet access for downloading tools
- Projector and screen for demonstrations

Workshop Segment Outline:

Part 1: Introduction to Static Analysis (20 minutes)

- **Conceptual Overview**: Explain what static analysis is and how it compares to dynamic testing. Discuss its role in identifying bugs, security vulnerabilities, and code smells without executing the code.
- What is Static Analysis?
  - **Definition**: Static analysis involves examining the source code of an application without executing it. This process utilizes tools that can automatically review the code to detect a variety of issues.
  - **Tools**: Common tools for static analysis in the PHP context include PHPStan, Psalm, and Phan. These tools parse the code, build an abstract syntax tree (AST), and perform various checks to identify issues.
- Comparison with Dynamic Testing
  - **Execution**: Unlike dynamic testing, which requires running the code or simulating execution in a test environment, static analysis does not interact with the code at runtime.
  - **Scope of Detection**: Dynamic testing can uncover runtime errors and specific issues that only manifest when the application is running (e.g., memory leaks, concurrency issues). Static analysis, however, is limited to what can be deduced from the codebase itself.
  - **Early Detection**: Static analysis can be performed early in the development cycle, even before the code is runnable, making it a powerful tool for early bug detection and prevention.
  - **Integration**: Static analysis tools are often easier to integrate into continuous integration pipelines than some forms of dynamic testing because they do not require a running environment.
- Role of Static Analysis
  - **Bug Detection**: Identifies common coding errors like syntax mistakes, type mismatches, undefined variables, unreachable code, and more, before the application is run.
  - **Security Vulnerability Identification**: Finds patterns that may lead to security vulnerabilities such as SQL

injection, cross-site scripting (XSS), or insecure handling of user input.
    - **Code Smells**: Detects potential issues in code quality that do not necessarily cause bugs but may indicate poor coding practices (e.g., overly complex methods, duplicate code, magic numbers).
    - **Code Quality Improvement**: Helps maintain a high standard of code quality by enforcing coding standards and detecting deviations from best practices.
    - **Documentation**: Some tools generate documentation that helps developers understand complex codebases, detect architectural flaws, or visualize dependencies.
- Advantages of Static Analysis Over Dynamic Testing
    - **Comprehensive Coverage**: Can theoretically examine all paths through the code, providing more comprehensive coverage than dynamic tests, which are limited by the scenarios they execute.
    - **Time Efficiency**: Since it does not require code execution, static analysis can be faster than setting up and running dynamic tests, especially on large codebases.
    - **Preventive Approach**: Helps prevent bugs and vulnerabilities from being introduced into the codebase, rather than detecting them after the fact.
- Limitations of Static Analysis
    - **False Positives/Negatives**: May generate false positives (flagging issues that are not actual problems) and false negatives (failing to detect actual issues).
    - **Context Awareness**: Lacks awareness of the runtime environment, which can lead to missed issues that only appear under specific runtime conditions.

Part 2: Configuring PHPStan/LaravelStan (20 minutes)

- **Installation Guide**: Walk through the installation of PHPStan and LaravelStan using Composer. Show how to integrate these tools into a typical Laravel development workflow.
- **Configuration Basics**:
    - Creating and modifying `phpstan.neon` configuration files.

    - Setting up analysis levels, defining paths to analyze, and excluding files or directories.
    - Integrating with Laravel to recognize Laravel's magic methods and facades.
- **Rule Setup**: Explain how to write custom rules for specific project requirements or to enforce coding standards.

Part 3: Hands-On PHPStan/LaravelStan Exercise (40 minutes)

- **Group Activity**: Participants will apply PHPStan/LaravelStan to a pre-existing Laravel application (provided as part of workshop materials).
    - Run initial analysis and review the output.
    - Identify and discuss notable issues raised by the tool (e.g., type safety issues, possible null pointer exceptions).
    - Modify the code based on the tool's feedback to resolve issues.
- **Advanced Configuration**: Demonstrate how to tweak PHPStan/LaravelStan settings for deeper analysis or to accommodate specific project needs.

Part 4: Discussion and Q&A (20 minutes)

- **Discuss Experiences**: Participants share their findings and challenges faced during the hands-on exercise.
- **Best Practices**: Offer tips on integrating static analysis into continuous integration/continuous deployment (CI/CD) pipelines.
- **Q&A**: Address any specific questions or concerns participants may have about using static analysis tools.

Key Learning Points:

- Understand the importance and benefits of static code analysis in software development.
- Gain practical experience in setting up and configuring PHPStan/LaravelStan.
- Learn how to interpret static analysis reports and integrate findings into the development process.

Delivery Tips:

- **Practical Examples**: Use real-world code examples to show how static analysis can catch errors that might not be easily spotted through manual review or dynamic testing.
- **Interactive Engagement**: Encourage participants to interact with the tools during the exercise, fostering a practical

understanding of the tool's capabilities.
- **Follow-Up Resources**: Provide links to documentation, advanced configurations, and community forums for further learning.

- SQL Injection

SQL Injection, also known as **SQLi**, is a vulnerability that allows an attacker to modify the SQL queries that an application makes to its database. This can be used to bypass authentication, read sensitive data, modify data, or even delete data. SQLi is made possible when user controlled values are injected into SQL queries without being correctly parameterised or escaped.

SQLi is one of the most dangerous vulnerabilities, as it can be used to gain full access to the database, and therefore the entire application. Anything stored in the database is at risk, including user data, passwords, and more. It can also be used to access any other databases the application has access to, such as the database of another applications - which is an incredibly common attack that occurs against WordPress sites.

SQLi is incredibly common - it sits at #3 in the OWASP Top 10 as part of A03:2021 - Injection, and is a common cause of data breaches. It is also incredibly easily to exploit, with dedicated tools like sqlmap automating the process for you. *(Please don't use this on the challenges!)*

SQLi is also incredibly easy to prevent, by using parameterised queries, which come as standard in any decent database toolkit - especially Laravel's Eloquent ORM. In fact, the ease at which SQLi can be prevented in Laravel makes it easy to overlook and sneak it's way into your code.

Let's run through the main types of SQLi, and how they occur in Laravel.

Note that all of these examples and the challenges are focused on `SELECT` queries, where data is extracted or conditionals are bypassed. However, the same concepts apply to `UPDATE`, `DELETE`, etc, queries. If you can modify some part of the query, you can make it do what you want - including inserting, updating, and deleting data.

Classic SQLi

Login Form

One of the classic or clichéd SQLi attacks is when an attacker is able to modify the behaviour of a login form, to gain access to an account they shouldn't have access to. This is usually done by modifying the query so the password is not checked, allowing the attacker to log into the specified account without knowing the password.

Consider the following SQL query:

```
$sql = "SELECT * FROM users WHERE username = '{$request-
>username}' AND password = '{$request→password}'";
$user = Arr::first select($sql) ;
```

All the attacker needs to do is escape the quoted strings, and they can modify the query.

For example, if the username and password are:

```
username: admin
password: ' OR 1=1 #
```

The query becomes:

```
SELECT * FROM users WHERE username = 'admin' AND password = '' OR
1=1 #'"
```

In case you didn't know, the `#` is a comment marker in SQL - allowing the attacker to prevent anything after their modifications from being executed.

This is a very simplistic example, as you will often need to handle nested brackets and extra conditionals and fields, but the principle is the same, and a determined attacker can use trial and error to find the correct syntax.

Search Form

Another classic example is when an attacker is able to modify a search query to return all the data in the database, bypassing specific filters and limits.

Consider:

```
$sql = "SELECT * FROM members WHERE name LIKE '%{$request->search}%' AND public = 1 AND tenant_id = {$tenant->id}";
$members = DB::select($sql);
```

The query relies on the `tenant_id` check to prevent users from accessing data from other tenants, and `public = 1` to prevent private members from being returned. However, if the attacker can modify the query, they can bypass these checks and display both private members, **and** members from other tenants.

Union-Based SQLi

Union-based SQLi is when an attacker is able to modify a query to return data from another table, and is named after the `UNION` keyword in SQL. This keyword can be used to easily extract data from any other table within the database - including tables that contain sensitive data, such as user details, passwords, and more, through an injection point on an unrelated table.

`UNION` combines the results from multiple queries into a single result set, and simply requires the same number of columns in each query. For example, if we have a table called `users` with the columns `id`, `name`, and `email`, we can use the following query to extract the data:

```
SELECT id, name, email FROM users UNION SELECT date, event, notes FROM meetings
```

This query will return all the data from the `users` table, and all the data from the `meetings` table, as long as both tables have the same number of columns.

Error-Based SQLi

Error-based SQLi occurs when the raw SQL error is returned to the user, allowing them to see the query that was run, and the error that occurred during execution. This is incredibly common in early versions of applications, before the developers have had a chance to build user-friendly error messages, or in technical-focused applications where the developers assume technical errors won't confuse technical users.

Consider the following error message:

```
Invalid Database Query: SQLSTATE[42000]: Syntax error or access
violation:
1064 You have an error in your SQL syntax; check the manual that
corresponds to your
MySQL server version for the right syntax to use near '''' limit
1' at line 1
(Connection: mysql, SQL: select * from `codes` where `enabled` = 1
and code = ''' limit 1)
```

This error message contains the full query, including the injection point, and the error that occurred during execution.

With the right injection, we can convince the database to hand over sensitive information, like the database server version, or the database name.

In the below error, you can see the database server version is `8.0.33-0ubuntu0.20.04.2`:

```
Invalid Database Query: SQLSTATE[HY000]: General error:
1105 XPATH syntax error: ' 8.0.33-0ubuntu0.20.04.2'
(Connection: mysql, SQL: select * from `codes` where `enabled` = 1
and code = '<REDACTED>' limit 1)
```

(I redacted the SQL as that gives you the solution to challenge #4!)

This method can be used to extract any information from the database.

Blind SQLi

Blind SQLi occurs when the attacker is unable to see the results of their injection, but can still modify the query. While this may seem like a way to prevent SQLi, it is still possible to extract data from the database by using conditional statements (or make changes).

There are two approaches you can use when faced with blind SQLi:

1. Error-State
2. Timing-Based

Both work with the same concept - if you can trigger a specific response or behaviour, you can use that to identify and extract data within the database. It may be tedious and time-consuming, but it is possible, and it's easily scriptable!

For example, trying to extract a 4-character string, one character at a time. You can cut down the time by comparing each character in the target with a range, and slowly narrow down the options. Such as `a-m > a-f > a-c > a-b > a`. With enough time, you can extract any data you want - even the entire database. Rate limiting and monitoring can help identify these sorts of attacks to limit data loss.

Blind Error-State SQLi

As per the name, this method relies on the attacker being able to see the error state of the query. I.e. if the query has failed or not. This usually occurs when an error message is displayed upon query failure, while a different message is displayed upon a failed/successful request.

Consider this code:

```
try {
    $validCode = DB::table('codes')
        ->where('enabled', true)
        ->whereRaw("code = '{$request->code}'")
        ->first();

    return $validCode ? 'Valid Code!' : 'Invalid Code!';
} catch (QueryException $exception) {
    return 'An Error Occurred...';
}
```

The error state of the query is immediately obvious, making it easy to trigger errors to glean information.

Blind Timing-Based SQLi

The next logical step is when the error-state is not known, and the attacker must rely on timing to identify specific conditions. This can be difficult across a slow internet connection, and be a lot more time-consuming, so it depends on the application and the attacker's patience.

SQLi in Laravel

SQLi occurs in Laravel when you inject user-controlled values into queries without correctly parameterising or escaping the values.

For example, executing raw queries via the `DB` facade, or directly via PHP's database helpers:

```
$sql = "SELECT * FROM users WHERE email = '{$email}' AND password
= '{$password}'";
$results = DB::select($sql);
```

Or maybe using one of Eloquent's many `raw` methods: `whereRaw`, `havingRaw`, `orderByRaw`, `groupByRaw`, `DB::raw`, etc.

```
$validCode = DB::table('codes')
    ->where('enabled', true)
    ->whereRaw("code = '{$request->code}'")
    ->first();
```

These methods and approaches are not inherently bad, but they are dangerous if you don't properly handle the user input you inject into the queries. One approach to avoid this is to escape the user input - but this is not recommended, and depends on the type of input and query. A far better approach is to use parameterisation.

It's not just raw user input, but anything the user can control in some way - even hardcoded values - has the potential to affect the query. As a general rule of thumb, any time you're constructing an SQL query, and you need to inject a value, you should be using parameterisation.

Now that you know what SQLi is, and how it can be used, head over to the challenges to get some hands-on experience!

- SQLi attacks
- SQLi defences

Day 3:

- AWS security

Workshop Segment Title:

Securing Your AWS Environment: Key Concepts and Practices

Objective:

Equip participants with foundational knowledge and practical strategies for enhancing security within their AWS deployments.

Target Audience:

IT professionals, cloud administrators, developers, and anyone involved in managing AWS resources.

Duration:

1-2 hours

Materials Needed:

- Computers with internet access
- Access to AWS Management Console (if live demo is feasible)
- Presentation software for slides
- AWS documentation and resources for reference

Workshop Segment Outline:

Part 1: Overview of AWS Security (15 minutes)

- **Introduction to AWS Security:** Discuss the shared responsibility model that outlines what security AWS provides and what the user must handle.
- **Detailed Explanation of the Shared Responsibility Model**
  - **Security 'of' the Cloud:** Discuss what AWS is responsible for in terms of security. This typically includes the hardware, software, networking, and facilities that run AWS Cloud services.
  - **Security 'in' the Cloud:** Outline what the customer (user) is responsible for, which generally involves managing the guest operating system (including updates and security patches), other associated application software, and the configuration of the AWS-provided security group firewall.
  - https://aws.amazon.com/compliance/shared-responsibility-model/
- **Key Components:** Brief overview of AWS security tools and services (e.g., IAM, Security Groups, VPC, KMS).

Part 2: Identity and Access Management (IAM) (20 minutes)

- There are two types of identities you need to manage when approaching operating secure AWS workloads.
  - **Human identities:** The administrators, developers, operators, and consumers of your applications require an identity to access your AWS environments and applications. These can be members of your organization, or external users with whom you collaborate, and who interact with your AWS resources via a web browser, client application, mobile app, or interactive command-line tools.
  - **Machine identities:** Your workload applications, operational tools, and components require an identity to make requests to AWS services, for example, to read data. These identities include machines running in your AWS environment, such as Amazon EC2 instances or AWS Lambda functions. You can also manage machine identities for external parties who need access. Additionally, you might also have machines outside of AWS that need access to your AWS environment.
- Initial Group Activity: Exploring IAM Best Practices
  - **Task:** Each group is asked to brainstorm and list down what they believe are the best practices for managing identities and permissions in AWS.
    - Encourage them to consider both human and machine identities.
    - They should think about access controls, security policies, and any specific strategies they believe are important for maintaining secure and efficient access to AWS resources.
  - **Objective:** Each group will create a brief presentation of their proposed best practices.
- **IAM Best Practices:**
  - SEC02-BP01 Use strong sign-in mechanisms
  - SEC02-BP02 Use temporary credentials
  - SEC02-BP03 Store and use secrets securely
  - SEC02-BP04 Rely on a centralized identity provider
  - SEC02-BP05 Audit and rotate credentials periodically
  - SEC02-BP06 Leverage user groups and attributes
- Permissions management

    - There are a number of ways to grant access to different types of resources. One way is by using different policy types.
    - **AWS-managed policies** – Managed policies that are created and managed by AWS.
    - **Customer-managed policies** – Managed policies that you create and manage in your AWS account. Customer-managed policies provide more precise control over your policies than AWS-managed policies.
- Permissions best practices:
    - SEC03-BP01 Define access requirements
    - SEC03-BP02 Grant least privilege access
    - SEC03-BP03 Establish emergency access process
    - SEC03-BP04 Reduce permissions continuously
    - SEC03-BP05 Define permission guardrails for your organization
    - SEC03-BP06 Manage access based on lifecycle
    - SEC03-BP07 Analyze public and cross-account access
    - SEC03-BP08 Share resources securely within your organization
    - SEC03-BP09 Share resources securely with a third party
-

Part 3: Detection in AWS

- Detection consists of two parts: detection of unexpected or unwanted configuration changes, and the detection of unexpected behavior.
    - The first can take place at multiple places in an application delivery lifecycle. Using infrastructure as code (for example, a CloudFormation template), you can check for unwanted configuration before a workload is deployed by implementing checks in the CI/CD pipelines or source control. Then, as you deploy a workload into non-production and production environments, you can check configuration using native AWS, open source, or AWS Partner tools. These checks can be for configuration that does not meet security principles or best practices, or for changes that were made between a tested and deployed configuration. For a running application, you can check whether the configuration has been changed in an unexpected

fashion, including outside of a known deployment or automated scaling event.
    - For the second part of detection, unexpected behavior, you can use tools or by alerting on an increase in a particular type of API call. Using Amazon GuardDuty, you can be alerted when unexpected and potentially unauthorized or malicious activity occurs within your AWS accounts. You should also explicitly monitor for mutating API calls that you would not expect to be used in your workload, and API calls that change the security posture.
- Best practices:
    - SEC04-BP01 Configure service and application logging
    - SEC04-BP02 Analyze logs, findings, and metrics centrally
    - SEC04-BP03 Automate response to events
    - SEC04-BP04 Implement actionable security events

Part 4: Infrastructure protection

- Infrastructure protection encompasses control methodologies, such as defense in depth, that are necessary to meet best practices and organizational or regulatory obligations. Use of these methodologies is critical for successful, ongoing operations in the cloud.
- Protect networks: You need to pivot from traditional models of trusting anyone and anything that has access to your network. When you follow the principle of applying security at all layers, you employ a Zero Trust approach. Zero Trust security is a model where application components or microservices are considered discrete from each other and no component or microservice trusts any other.
- Best practices:
    - SEC05-BP01 Create network layers
    - SEC05-BP02 Control traffic at all layers
    - SEC05-BP03 Automate network protection
    - SEC05-BP04 Implement inspection and protection
- Protecting compute: Compute resources include EC2 instances, containers, AWS Lambda functions, database services, IoT devices, and more. Each of these compute resource types require different approaches to secure them. However, they do share common strategies that you need to consider: defense in depth, vulnerability

management, reduction in attack surface, automation of configuration and operation, and performing actions at a distance.

- Best practices:
    - SEC06-BP01 Perform vulnerability management
    - SEC06-BP02 Reduce attack surface
    - SEC06-BP03 Implement managed services
    - SEC06-BP04 Automate compute protection
    - SEC06-BP05 Enable people to perform actions at a distance
    - SEC06-BP06 Validate software integrity

Part 5: Data protection

In AWS, there are a number of different approaches you can use when addressing data protection. The following section describes how to use these approaches.

- Data Classification:
    - Data classification provides a way to categorize organizational data based on criticality and sensitivity in order to help you determine appropriate protection and retention controls.
    - Best practices:
        - SEC07-BP01 Identify the data within your workload
        - SEC07-BP02 Define data protection controls
        - SEC07-BP03 Automate identification and classification
        - SEC07-BP04 Define data lifecycle management
- Protecting data at rest:
    - *Data at rest* represents any data that you persist in non-volatile storage for any duration in your workload. This includes block storage, object storage, databases, archives, IoT devices, and any other storage medium on which data is persisted. Protecting your data at rest reduces the risk of unauthorized access, when encryption and appropriate access controls are implemented.
    - Encryption and tokenization are two important but distinct data protection schemes.
    - *Tokenization* is a process that allows you to define a token to represent an otherwise sensitive piece of information (for example, a token to represent a

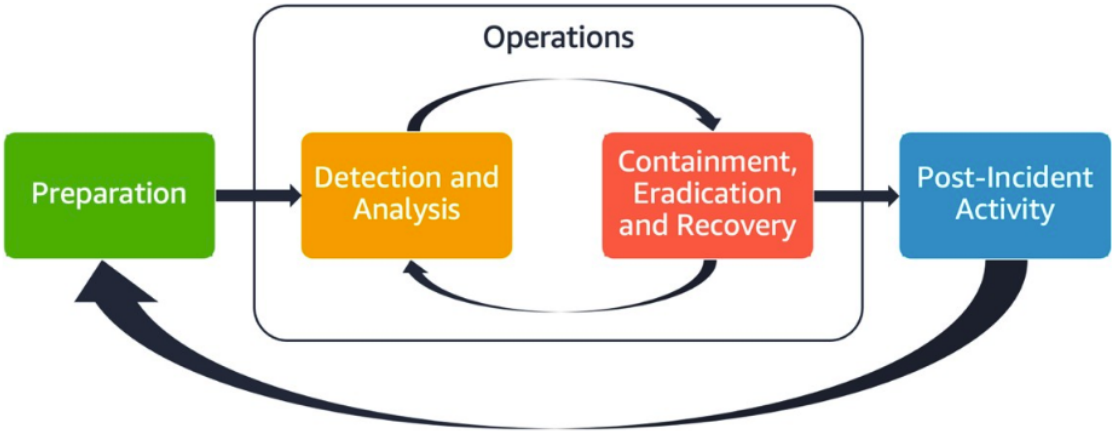
customer's credit card number). A token must be meaningless on its own, and must not be derived from the data it is tokenizing-therefore, a cryptographic digest is not usable as a token. By carefully planning your tokenization approach, you can provide additional protection for your content, and you can ensure that you meet your compliance requirements. For example, you can reduce the compliance scope of a credit card processing system if you leverage a token instead of a credit card number.

- *Encryption* is a way of transforming content in a manner that makes it unreadable without a secret key necessary to decrypt the content back into plaintext. Both tokenization and encryption can be used to secure and protect information as appropriate. Further, masking is a technique that allows part of a piece of data to be redacted to a point where the remaining data is not considered sensitive. For example, PCI-DSS allows the last four digits of a card number to be retained outside the compliance scope boundary for indexing.
- Best practices:
  - SEC08-BP01 Implement secure key management
  - SEC08-BP02 Enforce encryption at rest
  - SEC08-BP03 Automate data at rest protection
  - SEC08-BP04 Enforce access control
  - SEC08-BP05 Use mechanisms to keep people away from data

- Protecting data in transit:
  - *Data in transit* is any data that is sent from one system to another. This includes communication between resources within your workload as well as communication between other services and your end users. By providing the appropriate level of protection for your data in transit, you protect the confidentiality and integrity of your workload's data.
  - Best practices:
    - SEC09-BP01 Implement secure key and certificate management
    - SEC09-BP02 Enforce encryption in transit
    - SEC09-BP03 Automate detection of unintended data access

- SEC09-BP04 Authenticate network communications

Part 6: Incident response



We looked at this last week but it's important that you have an IR for your AWS services. Their diagram looks a lot like mine :)

Part 7: Application security

Application security (AppSec) describes the overall process of how you design, build, and test the security properties of the workloads you develop. You should have appropriately trained people in your organization, understand the security properties of your build and release infrastructure, and use automation to identify security issues.

That's what we're doing through this training.

Best practices:

- SEC11-BP01 Train for application security
- SEC11-BP02 Automate testing throughout the development and release lifecycle
- SEC11-BP03 Perform regular penetration testing
- SEC11-BP04 Manual code reviews
- SEC11-BP05 Centralize services for packages and dependencies
- SEC11-BP06 Deploy software programmatically
- SEC11-BP07 Regularly assess security properties of the pipelines
- SEC11-BP08 Build a program that embeds security ownership in workload teams

Key Learning Points:

- Understanding AWS’s shared responsibility model for security.
- Practical skills in using AWS security tools like IAM, VPC, KMS, and AWS CloudTrail.
- Awareness of the importance of monitoring and compliance in cloud security.

- GDPR and Data Protection

GDPR and Data Protection (Duration: 1 hour)

Introduction to GDPR (15 minutes)

- **What is GDPR?**
  - **Definition and Purpose**: Explain that the General Data Protection Regulation (GDPR) is a regulation in EU law on data protection and privacy in the European Union and the European Economic Area. It also addresses the transfer of personal data outside the EU and EEA.
    Objective: Emphasize GDPR's aim to give individuals control over their personal data and to simplify the regulatory environment for international business by unifying the regulation within the EU.
- **Key GDPR Requirements:**
  - **Data Protection by Design**: Introduce this principle as the requirement that data protection measures must be integrated into the development of business processes for products and services. For developers, this means incorporating data protection from the onset of the designing of systems, rather than an addition.
  - **Data Minimization**: Explain that organizations must ensure that only the data absolutely necessary for the completion of its business duties is collected and processed, both in terms of the amount of data and the extent of the processing.
  - **Rights of Data Subjects**: List and briefly explain rights such as the right to access, the right to be forgotten, the right to data portability, and rights related to automated decision making and profiling.
- **Implications for Web Applications:**
  - **Data Handling and Storage**: Discuss how web applications must be designed to secure personal data by default, highlighting the importance of encryption, secure data storage, and other protective measures.
  - **User Rights Management**: Cover how applications must provide mechanisms for users to exercise their rights,

such as deleting their account, accessing their data in a machine-readable format, and making corrections to their data.
    - **User Consent**: Stress the importance of obtaining clear, affirmative consent before collecting personal data, along with the necessity of making it as easy for users to withdraw consent as it is to give it.
- 10 Steps to help secure PII
    1. Identify the PII your company stores
    2. Find all the places PII is stored
    3. Classify PII in terms of sensitivity
    4. Delete old PII you no longer need
    5. Establish an acceptable usage policy
    6. Encrypt PII
    7. Eliminate any permission errors
    8. Develop an employee education policy around the importance of protecting PII
    9. Create a standardized procedure for departing employees
    10. Establish an accessible line of communication for employees to report suspicious behavior

GDPR Compliance Strategies for Laravel (30 minutes)

1. **Data Encryption (7 minutes)**
    - **Introduction to Encryption**: Briefly explain the importance of encryption in protecting personal data under GDPR.
    - **Laravel's Encryption Capabilities**: Introduce Laravel's built-in encryption facilities, which use OpenSSL to provide AES-256 and AES-128 encryption. Ensure participants understand the importance of never storing raw encrypted data alongside its encryption key.
    - **Practical Implementation**: Demonstrate how to use Laravel's `encrypt()` and `decrypt()` methods to securely encrypt and decrypt data. Show a simple example, such as encrypting user personal details before saving them to the database.
    - AWS vs Laravel encryption
2. **Secure Data Handling (7 minutes)**

- **Handling Sensitive Data**: Discuss the importance of sanitizing and validating all user inputs to prevent common vulnerabilities like SQL injection and XSS, which are also critical under GDPR for data integrity and security.
- **Validation Techniques**: Walk through Laravel's built-in validation rules that help ensure that only appropriately formatted data is stored. Show how to use Laravel's Validator facade or validation rules in a request class.
- **Sanitization Practices**: Briefly demonstrate how to sanitize data using Laravel's Eloquent ORM or mutators to automatically handle data formatting before it hits the database.

3. **User Consent and Data Access (8 minutes)**
   - **Managing Consent**: Explain the necessity of obtaining explicit consent from users before processing their data. Discuss the implementation of checkboxes or similar mechanisms that are unchecked by default for obtaining consent.
   - **Access, Correction, and Deletion**: Show how to create routes and controllers in Laravel to handle requests from users wanting to access, update, or delete their data. Emphasize the need for authentication and authorization checks to ensure that users can only affect their own data.
   - **Example Walkthrough**: Provide a brief code example of a form in Laravel that handles data access requests or consent management.
4. **Logging and Monitoring (8 minutes)**
   - **GDPR-Relevant Logging**: Link back to the importance of logging for security and compliance purposes discussed on previous days. Highlight how to configure Laravel's logging capabilities to ensure that they capture GDPR-relevant actions like data deletion requests and consent revocations without storing personal data in the logs.
   - **Monitoring Tools**: Suggest tools that can be integrated with Laravel to monitor and alert on suspicious activities or breaches. Discuss how to use these tools in compliance with GDPR.

    - **Practical Example**: Show a configuration example of Laravel logs that exclude personal identifiers but still capture the essence of the action for compliance purposes.

Wrap up by emphasizing the importance of an integrated approach to GDPR compliance, where encryption, secure data handling, user consent, and proactive monitoring form a cohesive strategy. Provide participants with additional resources or reading material for deeper exploration after the workshop.

3. **Hands-On: Making a Laravel Feature GDPR Compliant Exercise (45 minutes)**
    - **Preparation**: Have a pre-built Laravel application feature that handles user data, such as a user profile management system.
    - **Task**: Participants will enhance this feature to make it GDPR compliant. Tasks can include:
        - Implementing encryption for stored user data.
        - Adding functionality to log user consent and any data access requests.
        - Creating interfaces for users to request data deletion or correction.
    - **Tools and Libraries**: Introduce any specific packages or tools that facilitate GDPR compliance in Laravel, such as Laravel's built-in encryption libraries or third-party packages that help manage user consent.
4. **Review and Discussion (15 minutes)**
    - **Review Completed Work**: Allow groups or individuals to present their modified applications.
    - **Discuss Challenges**: What challenges did they face while implementing these changes?
    - **Best Practices Recap**: Highlight the best practices for maintaining GDPR compliance and discuss any ongoing maintenance requirements.

- Cross-Origin Resource Sharing (CORS)
  Any important part of the CSRF puzzle is Cross-Origin Resource Sharing (CORS) protection. CORS is a security feature built into web browsers that controls how different *cross-origin* sites can interact with each other. It comes with a set of defaults that set everything at a sane level of security, and you use different headers to loosen (or tighten) the security as needed.

It's such a big topic that we'll only be scratching the surface of it at the moment. Most Laravel apps won't need to worry about CORS at all - browsers defaults are designed for most use cases, and Laravel includes a set of sane defaults within `config/cors.php` for common scenarios. However, apps with complicated APIs or *cross origin* integration requirements will need to go down the CORS path, so it's worth being aware of what it is so you know where to go looking if you need to.

My aim for this section is to give you a basic understanding of CORS, so you're aware of what it does, how it helps with CSRF, and what to configure if you need to take it further.

Cross-Origin Resource Sharing (CORS)

Laravel can automatically respond to CORS `OPTIONS` HTTP requests with values that you configure. The `OPTIONS` requests will automatically be handled by the `HandleCors` middleware that is automatically included in your application's global middleware stack.

Sometimes, you may need to customize the CORS configuration values for your application. You may do so by publishing the `cors` configuration file using the `config:publish` Artisan command:

```
php artisan config:publish cors
```

This command will place a `cors.php` configuration file within your application's `config` directory.

CSRF Protections

Let's get started by looking at how CORS helps with CSRF protection.

By default, scripts on 3rd party domains can make requests to your app - **but cannot read the response**. This is crucial to blocking CSRF attacks, as it prevents malicious scripts from using GET requests to obtain the CSRF token before submitting forged requests with the token to impersonate the user.

It's important to note that CORS won't block requests being made, just how the response to that request is handled in the browser. So even with CORS in place (as it is by default),

it's still possible to send forged requests and impersonate users. **This is why CSRF attacks work.**

`Access-Control-Allow-Origin`

The main header used to control CORS is `Access-Control-Allow-Origin`. This header is sent by the server in response to a request, and tells the browser whether the response can be read by the requesting script.

This is a typical example of a developer trying to work around a security feature without taking the time to learn about it. The result being they simply disable it and leave their app wide open to attack.

```
Access-Control-Allow-Origin: https://<your-challenge-domain>
Access-Control-Allow-Credentials: true
```

With this disabled, the browser will allow the response to be read by the malicious script, exposing the CSRF token.

So why would we use this header?

A simple example would be if you have an API endpoint that needs to be accessed by the browser from a different *cross-origin* domain. In order to allow the browser to read the API response, you would need to disable CORS on that endpoint. This might be done between two (or more) of your own apps, or it might be done to allow 3rd party apps to access your API.

The risk is when you disable CORS for your API, but don't lock it down to encompass a specific path or resource, then other paths on your app may also be left open to attack.

`Access-Control-Allow-Credentials`

Linked with the `Access-Control-Allow-Origin` header is the `Access-Control-Allow-Credentials` header.

By default, the browser will not send any cookies (credentials) when making a cross-origin request. To get around this, you need to pass the `credentials: "include"` option to the `fetch()` request, which will tell the browser to include any cookies it is allowed to include. However, the browser won't blindly include cookies, it first makes what's called a *preflight* request to the server using the `OPTIONS` method to

check if the server allows credentials to be included. If the `Access-Control-Allow-Credentials` header is set to `true`, and the `Access-Control-Allow-Origin` header matches the origin making the request, then the browser will perform the request and include any cookies it is allowed to include.

So, these headers are defined, the browser acknowledges them, and the cookies are included in the request - allowing us to steal the CSRF token.

A quick note about form submissions

Form submissions are not subject to CORS restrictions, so you can submit forms to any domain you like. This includes forms that are submitted via JavaScript in the background. The browser will allow you to submit the form, **and** include any cookies (credentials) that are allowed to be included, **but** you cannot read the response to the form submission (without allowing it via CORS).

This is why we cannot rely on CORS alone to protect against CSRF attacks.

Defining CORS Policies in Laravel

As mentioned above, Laravel includes CORS support out-of-the-box through the `config/cors.php` configuration file, and the `\Illuminate\Http\Middleware\HandleCors` middleware.

The default configuration is as follows:

```php
<?php
return [
    'paths' => ['api/*', 'sanctum/csrf-cookie'],
    'allowed_methods' => ['*'],
    'allowed_origins' => ['*'],
    'allowed_origins_patterns' => [],
    'allowed_headers' => ['*'],
    'exposed_headers' => [],
    'max_age' => 0,
    'supports_credentials' => false,
];
```

This opens up the `api/*` (and `sanctum/csrf-cookie`) paths to allow any HTTP Method, from any origin, with any headers, while it specifically excludes support for credentials. This

allows the `api/*` endpoint to be used by any 3rd party app, but prevents the browser from including any cookies in the request - blocking CSRF attacks.

The `sanctum/csrf-cookie` is designed to be used for *same origin* requests and sets the CSRF token into a cookie. Since `supports_credentials` is set to `false`, this cookie is not included for *cross origin* requests and therefore cannot be read by a malicious script.

You can modify these settings to suit your needs, or replace the middleware with your own CORS implementation as needed.

CORS Headers

As a reference for the headers used by CORS, here's a quick summary:

- `Access-Control-Allow-Origin` - The origin(s) that are allowed to read the response. This can be a single origin, a comma-separated list of origins, or `*` to allow any origin.
- `Access-Control-Allow-Methods` - The HTTP methods that are allowed for accessing the resource. This can be a comma-separated list of HTTP method names, or `*` to allow any method.
- `Access-Control-Allow-Headers` - The allowed request headers for a resource. This can be a comma-separated list of header names, or `*` to allow any header.
- `Access-Control-Allow-Credentials` - Whether the response can be shared when the request's credentials mode is "include". This can be `true` or `false`.
- `Access-Control-Expose-Headers` - The additional headers that are accessible within the client-side Javascript making the request.
- `Access-Control-Max-Age` - The amount of time (in seconds) that the results of a preflight request can be cached.
- `Access-Control-Request-Method` - Used when issuing a preflight request to let the server know what HTTP method will be used when the actual request is made.
- `Access-Control-Request-Headers` - Used when issuing a preflight request to let the server know what HTTP headers will be used when the actual request is made.

Further Learning

- [MDN CORS Documentation]
- Not a security feature - doesn't make things more secure - doesn't stop random API calls, browser security - protecting user and browser - XSS protection.

- Cookies in Laravel
  - SameSite Cookies
    SameSite cookies are a newer security feature designed to combat [CSRF attacks]. They are a cookie attribute that is defined on each cookie when it is sent to the browser, much like the `HttpOnly` and `Secure` attributes. The browser honours the attribute and will only send the cookie if the request meets the criteria.

    If the `SameSite` attribute is not set, then the browser will default to `SameSite=Lax` on all cookies, providing a safe default for the cookie. In addition, Laravel will, by default, set `SameSite=Lax` it's cookies. This ensures that you're covered by SameSite cookies by default, and you only need to worry about them if you specifically change the value.

    It's important to note that SameSite cookies are not a replacement for CSRF tokens, but rather an additional layer of protection on top of other protections like [CSRF tokens]. They are designed to prevent CSRF attacks, but they don't prevent *all* CSRF attacks, and as we saw in the [CSRF Challenges], chaining vulnerabilities together can lead to a successful SameSite cookie bypass.

    ## Lax, Strict, and None

    There are three possible values for the SameSite attribute:

    - `SameSite=Lax` - the default value, which indicates that the cookie will be sent on all *safe* requests, and only on *unsafe* requests if the request is a *same site* request.
    - `SameSite=Strict` - more restrictive, and will only send the cookie on *same site* requests.
    - `SameSite=None` - will send the cookie on all requests, regardless of the request type.

As mentioned above, `Lax` is the default value and is generally what you'll want to use. It provides a nice balance between security and usability - allowing users to stay logged in as they navigate to and from your site, while preventing CSRF attacks and other malicious activities.

`Strict` may be useful if you have a site that is entirely self-contained, and you need strict controls over when cookies are sent. You'll typically have very short user sessions that require frequent activity to keep the session alive. Some financial systems or ticketing systems may require this level of control.

`None` is the most permissive option, and should only be used if you have a specific need for it. It disables the SameSite protection entirely, and will send the cookie on all requests, regardless of the request type. It also requires the `Secure` attribute to be set, so it will only work over HTTPS. This is useful if your site is integrated closely with other sites, such as being embedded in an `<iframe>` or even if you need to support cross-site `POST` submissions.

Safe vs Unsafe Requests

Requests are considered **safe** when they use the `GET`, `HEAD`, `OPTIONS`, or `TRACE` methods. These methods are considered safe because, by convention, they should not change the state of the server in any way. They should only be used to retrieve data from the server, and should not have any side effects.

Requests are considered **unsafe** when they use the `POST`, `PUT`, `PATCH`, or `DELETE` methods. These methods do change the state of the server, and can have side effects.

In addition, safe requests cannot be within an `<iframe>` or `<img>` tag, but must be a top level navigation event. This is to prevent click-jacking attacks, where a malicious site embeds your site in an `<iframe>` and tricks the user into clicking on something they didn't intend to. *This is why some of*

the CSRF challenges won't work within the challenge frame and need to be loaded in a new tab.

Same Site vs Cross Site Requests

So now you know what the different options are and what safe and unsafe requests are, but what is a *same site* request?

The definition of *same site* is a little complicated, but it's basically any request that is on the same root domain as the site that set the cookie. So if you have a cookie set on `https://one.example.com`, then any request to `https://two.example.com`, or even `https://example.com` is considered the *same site* and the cookie will be included. However, a request from `https://example.net` is considered a *cross site* request because the root domain is different.

This loose definition of *same site* is demonstrated in CSRF challenge #5, where `SameSite=Lax` is set on the cookie but a subdomain within the same root domain is able to make a CSRF request. This is why the `SameSite` attribute is not a replacement for CSRF tokens, but is a useful additional layer of protection. If your only protection against CSRF attacks is SameSite cookies, all an attacker needs to do is find a vulnerable subdomain, and they can bypass the SameSite protection.

It's also important to be aware of the Public Suffix List, which is a list of domains that are considered public suffixes, where anyone can register a subdomain. `github.io` is a good example of this. Requests between different subdomains on this list are **always** considered *cross site* requests, keeping each site separate and protected.

SameSite Cookies in Laravel

Laravel's default SameSite value comes from the `session.same_site` configuration value, which can be found in `config/session.php`:

```
'same_site' => 'lax',
```

> This value is used when setting the `SameSite` attribute on any of the session cookies Laravel sets.
>
> When setting your own cookies on a response, you can set the attribute like any of the other cookie options.

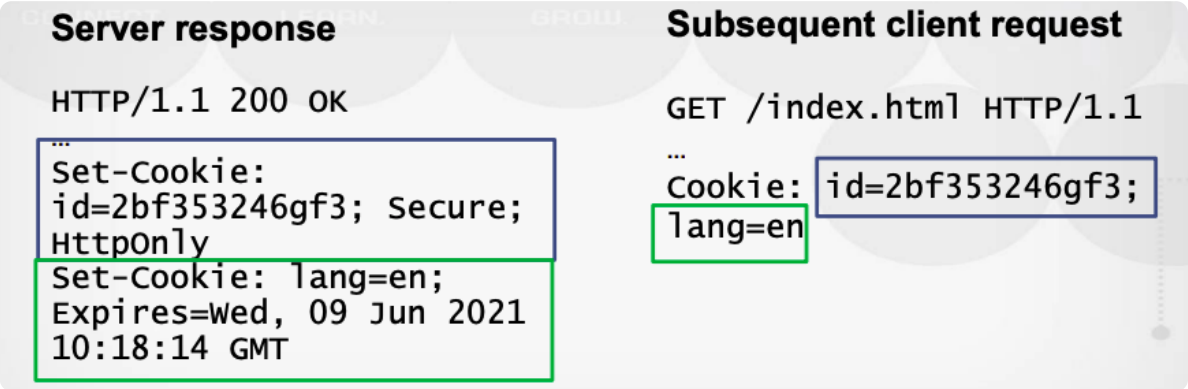
```
return response()
    ->view('template')
    ->withCookie(cookie(name: $name, value: $value,
sameSite: 'Strict'));
```

Cookie Security and Session Management¶

Cookies are based on an old recipe:

- 1994 -Netscape draft
- 1997 - RFC 2109
- 2000 - RFC 2965
- 2002 - HttpOnly - MicroSoft - IE - tried to stop the cookie being accessible by JS in browser
- 2011 - RFC 6265 - rather than say what should happen, let's instead accept they are in use and document that
- 2017 - RFC 6265bis (draft) (still seems to be in draft) https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis-13

How are cookies set?

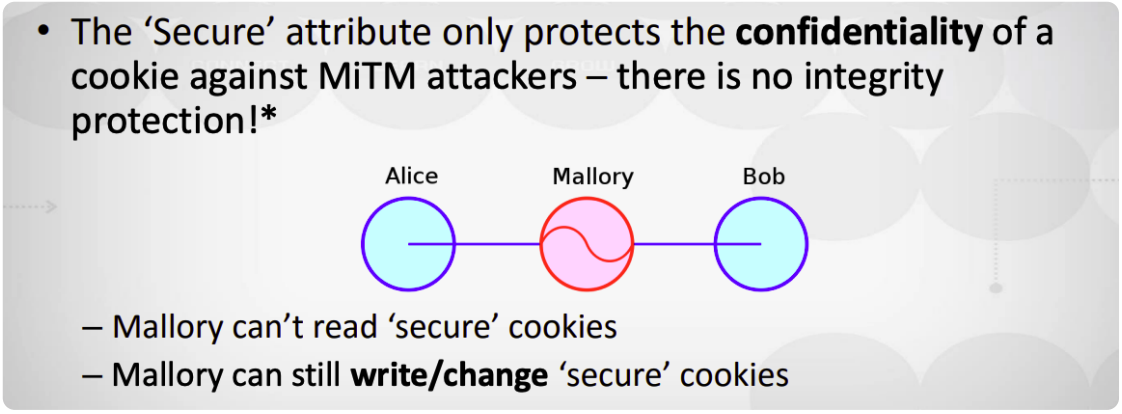


- Set by header
- When sent back to the server, the attributes aren't sent back just the value

`Secure`

> True or false? "Cookies marked with the 'Secure' attribute are only sent over encrypted HTTPS connections and are therefore safe from manin-the-middle attacks."

- True for confidentiality
- False for integrity



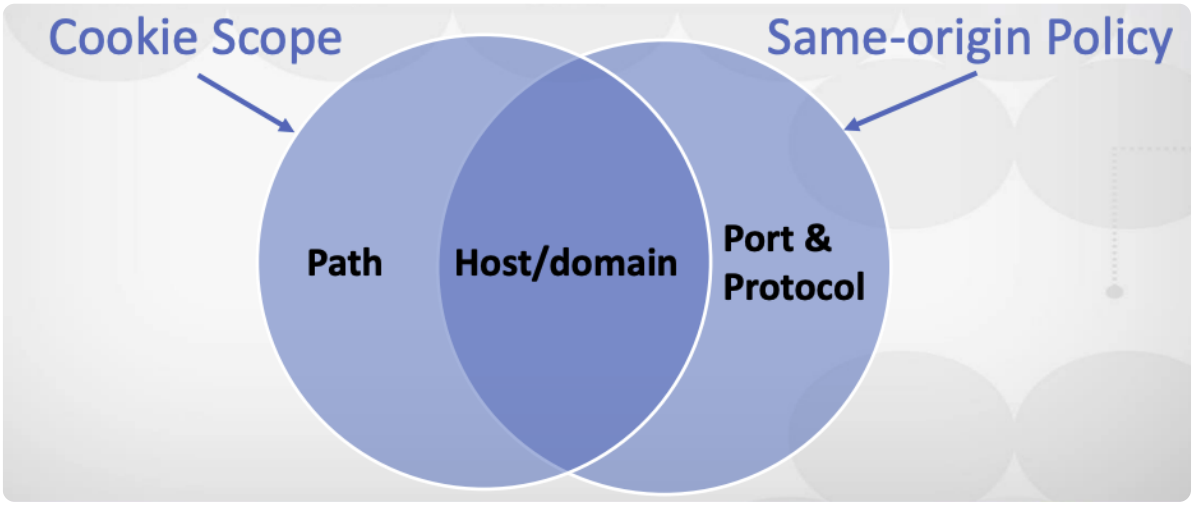
HTTPONLY

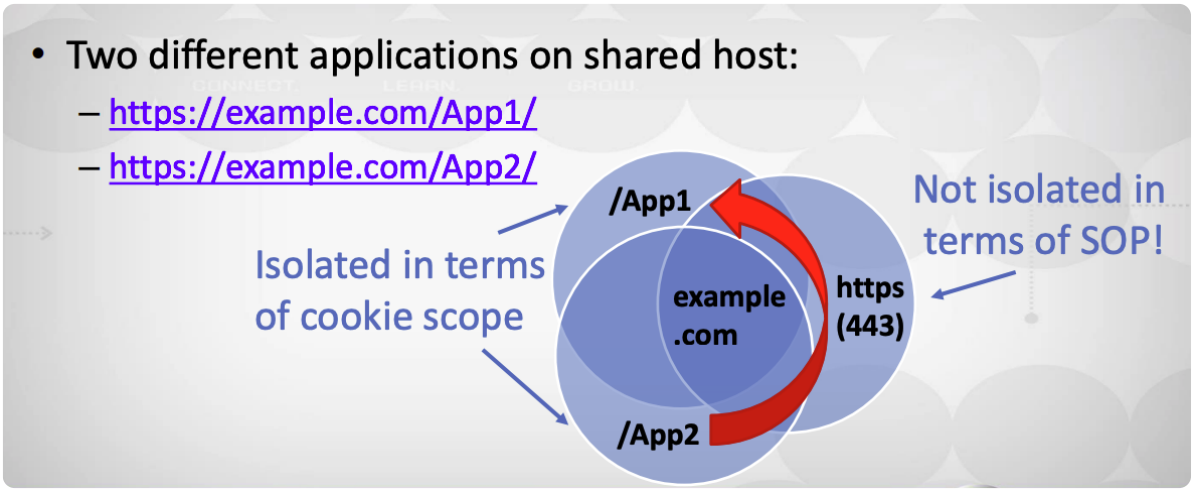
> True or false? "Cookies marked with the 'HttpOnly' attribute are not accessible from JavaScript and therefore unaffected by cross-site scripting (XSS) attacks."

- Invented by MS to keep JS away from the cookie jar!
- Only confidentiality protected in practice
- HttpOnly-cookies can be replaced by overflowing the cookie jar from JavaScript

Path

> True or false? "The 'Path' attribute limits the scope of a cookie to a specific path on the server and can therefore be used to prevent unauthorized access to it from other applications on the same host."





`Domain`

> True or false? “The ‘Domain’ attribute should be set to the origin host to limit the scope to that particular server. For example if the application resides on server app.mysite.com, then it should be set to domain=app.mysite.com”

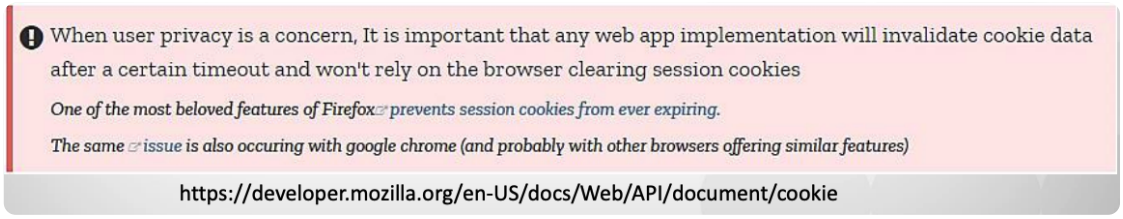
- With domain set, cookies will be sent to that domain and all its subdomains
- The risk with subdomains is lower than when scoped to parent domain, but still relevant
- Remove domain attribute to limit cookie to origin host only

Cookie lifetime

> True or false? “A session cookie, also known as an in-memory cookie or transient cookie, exists only in temporary memory while the user navigates the website.”

- It’s up to the browser to decide when the session ends

- ‘Non-persistent’ session cookies may actually be persisted to survive browser restart



Modern Cookie protections

Strict Secure cookies

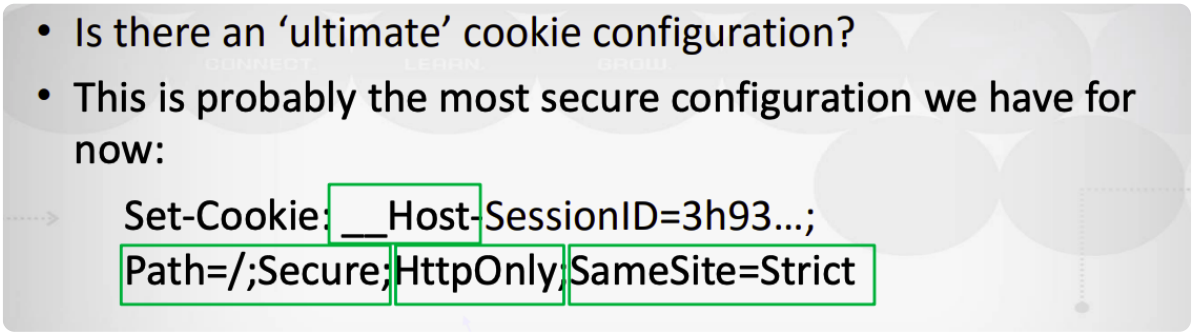
- Makes ‘secure’ cookies a little more secure by adding integrity protection
- Prevents plain-text HTTP responses from setting or overwriting ‘secure’ cookies
- Attackers still have a window of opportunity to “pre-empt” secure cookies with their own

Cookie Prefixes

- Problem:
  - Server only sees cookie name and value in HTTP request, no information about its attributes
  - Impossible for server to know if a cookie it receives was set securely
- Solution:
  - ‘Smuggle’ information to server in cookie name
  - `__Secure-` prefix
  - `__Host-` prefix

The `SameSite` attribute

- Problem:
  - Cookies are sent with all requests to a server, regardless of request origin
  - Attackers can abuse this by initiating authenticated cross-origin requests, e.g., CSRF, XSSI, etc.
- Solution:
  - New cookie attribute SameSite=[Strict|Lax]
  - Prevents cookies from being attached to cross-origin requests



By default, Laravel is configured in a secure manner. However, if you change your cookie or session configurations, make sure of the following:

- Enable the cookie encryption middleware if you use the `cookie` session store or if you store any kind of data that should not be readable or tampered with by clients. In general, this should be enabled unless your application has a very specific use case that requires disabling this. To enable this middleware, simply add the `EncryptCookies` middleware to the `web` middleware group in your `App\Http\Kernel` class:

```
/**
* The application's route middleware groups.
*
* @var array
*/
protected $middlewareGroups = [
  'web' ⇒
          \App\Http\Middleware\EncryptCookies  class,          .

..

 ,
  ...
];
```

Since Laravel encrypts all cookies, including your application's session cookie, essentially every request to a Laravel application relies on encryption. However, because of this, rotating your application's encryption key would log all users out of your application. In addition, decrypting data that was encrypted by the previous encryption key becomes impossible.

If you change your application's encryption key, all authenticated user sessions will be logged out of your application. This is because every cookie, including session cookies, are encrypted by Laravel. In addition, it will no longer be possible to decrypt any data that was encrypted with your previous encryption key.

To mitigate this issue, Laravel allows you to list your previous encryption keys in your application's `APP_PREVIOUS_KEYS` environment variable. This variable may contain a comma-delimited list of all of your previous encryption keys:

```
APP_KEY="base64:J63qRTDLub5NuZvP+kb8YIorGS6qFYHKVo6u7179stY="
APP_PREVIOUS_KEYS="base64:2nLsGFGzyoae2ax3EF2Lyq/hH6QghBGLIq5uL+Gp8/w="
```

When you set this environment variable, Laravel will always use the "current" encryption key when encrypting values. However, when decrypting values, Laravel will first try the current key, and if decryption fails using the current key, Laravel will try all previous keys until one of the keys is able to decrypt the value.

This approach to graceful decryption allows users to keep using your application uninterrupted even if your encryption key is rotated.

- Enable the `HttpOnly` attribute on your session cookies via your `config/session.php` file, so that your session cookies are inaccessible from Javascript:

`'http_only' ⇒ true,`

- Unless you are using sub-domain route registrations in your Laravel application, it is recommended to set the cookie `domain` attribute to null so that only the same origin (excluding subdomains) can set the cookie. This can be configured in your `config/session.php` file:

`'domain' ⇒ null,`

- Set your `SameSite` cookie attribute to `lax` or `strict` in your `config/session.php` file to restrict your cookies to a first-party or same-site context:

`'same_site' => 'lax',`

- If your application is HTTPS only, it is recommended to set the `secure` configuration option in your `config/session.php` file to `true` to protect against man-in-the-middle attacks. If your application has a combination of HTTP and HTTPS, then it is recommended to set this value to `null` so that the secure attribute is set automatically when serving HTTPS requests:

`'secure' => null,`

- Ensure that you have a low session idle timeout value. OWASP recommends a 2-5 minutes idle timeout for high value applications and 15-30 minutes for low risk applications. This can be configured in your `config/session.php` file:

`'lifetime' => 15,`

You may also refer the Cookie Security Guide to learn more about cookie security and the cookie attributes mentioned above.

- Using Cypress to Test for Security Vulnerabilities

Introduction to Cypress for Security Testing

Duration: 1 hour

1. Introduction to Cypress (10 minutes)

- **What is Cypress?**
  - Explain that Cypress is a front-end testing tool built for the modern web. It is both a powerful and versatile tool that allows you to write various types of tests such as end-to-end tests, integration tests, and unit tests.
- **Importance in Security Testing:**
  - Discuss how automated testing tools like Cypress can be used to simulate attacks and detect vulnerabilities in a web application, emphasizing its role in a proactive security strategy.
  - Highlight how Cypress can interact with HTML forms, handle sessions, and mimic user behavior on a web application, which is crucial for testing security measures like CSRF and XSS protections.

2. Setting Up Security Tests (20 minutes)

- **CSRF Testing Setup:**
  - Explain CSRF and why it's important to test for vulnerabilities. Show how to simulate a form submission using Cypress to test if Laravel's CSRF token checks are enforced.
  - Example Cypress Test:

```
describe('CSRF Protection', () => {
  it('Rejects form submission without CSRF token', () => {
    cy.visit('/login');

cy.get('input[name="email"]').type('user@example.com');
    cy.get('input[name="password"]').type('password');
    // Simulate form submission without a CSRF token
    cy.request({
      method: 'POST',
      url: '/login',
      failOnStatusCode: false,
      body: {
        email: 'user@example.com',
        password: 'password'
      }
    }).then((resp) => {
      expect(resp.status).to.eq(419); // Assuming 419
status code for CSRF token mismatch
    });
  });
});
```

- **XSS Testing Setup:**
  - Describe XSS and the importance of ensuring that user input is sanitized. Demonstrate using Cypress to verify if script tags entered into form inputs are properly escaped.
  - Example Cypress Test:

```
describe('XSS Protection', () => {
  it('Escapes script tags in user input', () => {
    cy.visit('/profile/edit');
    cy.get('input[name="bio"]').type('<script>alert("XSS")
</script>');
    cy.get('form').submit();
    cy.get('div.bio').should('not.contain', '<script>');
  });
});
```

3. Hands-On: Writing and Running Cypress Tests (30 minutes)

- Exercise Setup:
  - Provide participants with a sample Laravel application configured for testing.
  - Ensure that Cypress is installed and configured to run against this application.
- Task:
  - Participants write and execute Cypress tests to check for CSRF and XSS vulnerabilities in the provided application.
  - Guide them to use the Cypress Dashboard and visual test runner to view the results of their tests.
- Review and Discuss:
  - After completing the tests, review the outcomes with the participants.
  - Discuss common pitfalls in security testing and how automated tests can help mitigate them.

Additional Tips

- **Preparation**: Ensure all participants have access to the application repository and that Cypress is set up and ready to run before the workshop begins.
- **Documentation**: Provide links to further reading on Cypress and security testing practices, and how to integrate Cypress into a CI/CD pipeline for continuous security assessment.

- CD with a Security Focus

Integrating Cypress security tests into a CI/CD pipeline is an effective way to ensure continuous security assessment throughout the development and deployment lifecycle. GitHub Actions and Jenkins are popular tools that can be used to automate these tests. Here's how you can structure this part of your workshop to cover both theoretical concepts and practical implementation.

Integrating Security into CI/CD

Duration: 1.5 hours

1. Introduction to CI/CD and Security Integration (20 minutes)

- **Overview of CI/CD:** Briefly explain Continuous Integration and Continuous Deployment/Delivery, emphasizing the importance of automating tests in the CI/CD pipeline.
- **Role of Security in CI/CD:** Discuss why it's crucial to integrate security tests like those from Cypress into CI/CD pipelines to catch vulnerabilities early and often.
- **Examples of Tools:** Highlight GitHub Actions and Jenkins as tools that facilitate this integration.

2. Setting Up GitHub Actions for Cypress Tests (40 minutes)

- **Introduction to GitHub Actions:**
  - Explain what GitHub Actions are and how they can be used to automate workflows directly within GitHub repositories.
  - Show how to set up a basic workflow file.
- **Creating a Workflow for Cypress Tests:**
  - Guide participants through the process of creating a `.github/workflows/cypress.yml` file in their repository.
  - Explain and write the YAML configuration for a workflow that installs dependencies, runs a build, and executes Cypress tests.

```yaml
name: Cypress Tests

on: [push, pull_request]

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Node
      uses: actions/setup-node@v2
      with:
        node-version: '14'
    - name: Cache Node modules
      uses: actions/cache@v2
      with:
        path: ~/.npm
        key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
        restore-keys: |
          ${{ runner.os }}-node-
    - name: Install Dependencies
```

```
      run: npm install
    - name: Run Cypress Tests
      uses: cypress-io/github-action@v2
      with:
        start: npm start
        wait-on: 'http://localhost:3000'
        wait-on-timeout: 300
```

- **Explanation:**
  - Break down each step of the workflow, explaining how the Cypress GitHub Action is configured to run tests after the application is built and the server is started.

3. Hands-On: Configure GitHub Actions (30 minutes)

- **Task:**
  - Participants will fork a provided GitHub repository containing a sample Laravel application with Cypress tests.
  - They will create the `.github/workflows/cypress.yml` file based on the instructions and commit it to their fork.
  - Trigger the workflow by making a pull request or push to their repository to see the action in play.
- **Review:**
  - Participants share their screens to show the action results.
  - Discuss any errors that occurred and troubleshoot common issues.

4. Jenkins Alternative Discussion (10 minutes)

- **Overview of Jenkins:**
  - Briefly discuss how Jenkins could be used for a similar purpose, particularly in environments where self-hosted solutions are preferred.
  - Mention plugins like the Cypress Jenkins plugin which can facilitate similar workflows.
- **Comparative Discussion:**
  - Encourage a discussion on the pros and cons of using GitHub Actions versus Jenkins for running Cypress tests in different organizational contexts.

Additional Resources

- Provide detailed guides and links to documentation for GitHub Actions and Jenkins.

- Offer examples of more complex workflows and configurations for different environments.

By the end of this workshop section, participants should be comfortable setting up automated Cypress tests in a CI/CD pipeline using GitHub Actions and have a basic understanding of how to do it with Jenkins. This ensures they can not only write security tests but also integrate them effectively into their development workflows.

- Snyk, Socket (https://socket.dev/pricing)
- https://www.laravel-enlightn.com/

- CSRF in Laravel

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery, also known as **CSRF**, is a vulnerability that allows an attacker to forge requests from the victim's browser, impersonating the victim and performing actions on their behalf that they are not aware of. It can be used to perform any action available on a website, such as changing passwords, making purchases, modifying settings, or even creating or promoting admin users. When an app is vulnerable to CSRF, the possibilities are limited only by what functionality is exposed in the UI.

CSRF takes advantage of the fact that browsers cannot tell the difference between a request made by the user and a request made by a malicious script. When a user creates a session on a website (usually through logging in), the browser receives a cookie that identifies the user and their session. The browser will try to send this session cookie with every subsequent request, to ensure that the user is recognised and their session is used for the request. However, the browser does not know **what** initiated each request, only that a request was made, so malicious scripts on 3rd party sites can also send requests with the user's session cookie.

It's important to note that Laravel apps are well defended against CSRF attacks by default. The protections Laravel comes with need to be disabled, but that isn't an uncommon occurrence with complicated web apps. This is why it's so important to understand CSRF attacks, so if you do need to disable or weaken the defences, you know how to compensate and keep your app protected.

CSRF Attacks

CSRF attacks require 4 things:

1. The target app must be vulnerable to CSRF attacks.
2. The victim must possess some form of *session cookie* stored in their browser, which identifies them uniquely to the target app. This is usually an authentication cookie after the user has logged in.
3. The victim must be tricked into visiting a 3rd party site that contains a malicious script.
4. The malicious script must be able to send a request to the target app, and include the victim's *session cookie*.

When all these are met, a common attack would look like this:

1. Victim logs into the target app (`target-site.com`), and receives a session cookie (`cookie=ABCD1234`).

```
GET https://target-site.com/change-password
Cookie: session=ABCD1234

HTML response...
```

2. Victim visits a 3rd party site (`malicious.com`) that contains a malicious script.

```
GET https://malicious.com
Cookie: none

HTML response, including:
<script src="/malicious.js"></script>
```

3. The malicious script sends a request to the target app (`target-site.com`), and the browser includes the victim's session cookie (`cookie=ABCD1234`) with the request.

   The script hidden within `/malicious.js`:

```
fetch("https://target-site.com/change-password", {
    method: "POST",
    credentials: "include",
    headers: {"Content-Type": "application/x-www-form-
urlencoded"},
```

```
    body: "password=qwerty&password_confirmation=qwerty"
});
```

Sends request in the background:

```
POST https://target-site.com/change-password
Cookie: session=ABCD1234
Content-Type: application/x-www-form-urlencoded

password=qwerty&password_confirmation=qwerty
```

4. The target app receives the request, and since the session cookie is valid, it assumes the request is legitimate.

```
cookie→session ⟸ user→session
```

5. The target app performs the action requested by the malicious script, such as changing the victim's password.

There are a number of moving parts, but it's relatively easy when you think about it in terms of a Form Submission. Normally you'd visit the form on `target-site.com` via `GET` and submit the form to `target-site.com` via `POST/PUT`. The only difference during the attack is that the form is submitted from `malicious.com` instead of `target-site.com`.

CSRF attacks don't have to be form-encoded requests via `fetch()` - you can use whatever method you can trigger in the victim's browser. For example, they can be actual `<form>` elements on the page that submit a full page `POST`, or maybe `<iframe>`s and `<img>`s that send `GET` requests. The only requirements are that you send data in the format the target is expecting to receive it in.

Bypassing CSRF Protections

There isn't one single solution to CSRF, but rather a couple of different security mechanisms that work together to prevent it. This gives us multiple failure points, and different bypass methods depending on where the weaknesses are.

We'll cover each of these methods in their own **Defend** modules, so we'll focus on the general concepts and bypass techniques.

CSRF Tokens

CSRF Tokens are the first defence against CSRF attacks. They are a random string that is generated by the server and stored in the user's session. When the user makes a request to the server, the token is sent along with the request. The server then checks that the token matches the one stored in the session, and if it does, the request is allowed. If the token doesn't match, the request is rejected.

CSRF tokens are enabled by default in Laravel, all you need to do is use the `@csrf` directive or `csrf_field()` helper in your forms to include the CSRF token field.

However, CSRF tokens are not enough to prevent CSRF attacks on their own... If CORS protection has been disabled (see below), then the CSRF token will be available to any scripts that CORS allows to make requests. Also, Laravel makes it really easy to disable CSRF protection on specific routes, so you need to be careful that you don't disable it on routes that are vulnerable to CSRF attacks.

Learn more in the CSRF Tokens Defend module.

CORS Protection

CORS stands for Cross-Origin Resource Sharing, and it is a security feature built into web browsers that limits how scripts on different domains can interact with your app.

By default, scripts on 3rd party domains can make requests to your app - **but cannot read the response**. This is crucial to blocking CSRF attacks, as it prevents malicious scripts from first making `GET` requests to obtain the CSRF token before submitting forged requests with the token to impersonate the user.

It's important to note that CORS won't block requests being made, just how the response is handled. So even with CORS in place (as it is by default), it's still possible to send forged requests and impersonate users

- this is why CSRF attacks work.

As with CSRF tokens, the risk with CORS is when you specifically disable them. It's harder to do, but there are

legitimate reasons to do so, such as when you need to make requests from a 3rd party domain to your app. However, if you open your site up too much, you can expose your CSRF tokens and make your app vulnerable.

CORS is the reason you need to include `credentials: "include"` in your `fetch()` requests. By default, the browser won't include any cookies (credentials), but adding that option instructs the browser to include any cookies it is allowed to include.

SameSite Cookies

SameSite cookies are a newer security feature designed to combat CSRF attacks, which add an extra layer of protection on your app. They work by telling the browser which cookies are supported on which requests, and which are not.

The default value is `SameSite=Lax`, which indicates that the cookie will be sent on all *safe* requests, and only on *unsafe* requests if the request is a *same site* request. `SameSite=Strict` is more restrictive, and will only send the cookie on *same site* requests, while `SameSite=None` will send the cookie on all requests, regardless of the request type.

Safe requests are `GET`, `HEAD`, `OPTIONS`, and `TRACE`, while *unsafe* requests are `POST`, `PUT`, `PATCH`, and `DELETE`. Safe requests also cannot be within a `<iframe>` or `<img>`, but must be a top level navigation event.

There are two risks with SameSite cookies:

1. Enabling `SameSite=None`. There are legitimate use cases for setting `SameSite=None`, such as when you need to make requests from a 3rd party domain to your app. However, you need to be careful your other protections (such as CSRF tokens) are still in place.
2. The definition of *same site* is quite broad, and includes any domain that shares the same base domain. For example, `one.target-site.com` and `two.target-site.com` are considered the same site, because they are both under `target-site.com`. The risk here, if you haven't spotted it yet, is that a malicious script running

on `one.target-site.com` can make requests to `two.target-site.com` and include the victim's session cookie.

Note, there is a list called the public suffix list, which lists domains that shouldn't be considered the same site across subdomains. This is for services like `github.io` that provide subdomains for different users.

Summary

CSRF attacks are a relatively simple attack, with the complexity coming from the different layers of protection and methods of bypassing them. Luckily for us as Laravel developers, we have all the tools we need to defend our sites.

Most Laravel apps won't need to worry about CSRF beyond ensuring that all routes are covered by the CSRF middleware, and that the CSRF token is included in all forms. However, it's still important to understand how CSRF attacks work and what defences are in place.

As is the theme with Practical Laravel Security, you should head over now to the challenges. Working through those and inspecting each of the pieces of code, checking cookies, following the requests, etc, will explain CSRF in a much more practical way.

Attacks

Defenses

- CSRF Tokens
- SameSite Cookies
- Cross-Origin Resource Sharing (CORS)

- DDOS Protection and Rate Limiting
  To prepare for a workshop session on DDOS mitigation techniques focusing on Laravel configurations and integrating services like Cloudflare, you will want to cover both theoretical aspects and practical hands-on activities. Here's a structured plan on how to do this effectively:

DDOS Mitigation Techniques

Duration: 1.5 hours

1. Introduction to DDOS Attacks and Mitigation Techniques (20 minutes)

- **What is a DDOS Attack?**
  - Explain Distributed Denial of Service (DDOS) attacks, emphasizing their goal to exhaust the resources of a target.
- **General Mitigation Strategies:**
  - Brief overview of common mitigation techniques including rate limiting, Web Application Firewalls (WAFs), and geographic blocking.

2. Integrating Cloudflare for DDOS Protection (20 minutes)

- **Overview of Cloudflare:**
  - Explain how Cloudflare can protect web applications from a variety of attacks, including DDOS, by acting as a reverse proxy.
- **Features Relevant to DDOS Mitigation:**
  - Discuss Cloudflare's rate limiting, I'm Under Attack Mode, and other relevant features like CDN caching that help mitigate DDOS attacks.
- **Setting Up Cloudflare:**
  - Demonstrate how to add a Laravel application to Cloudflare.
  - Show how to configure basic security settings in Cloudflare's dashboard.

3. Rate Limiting in Laravel (30 minutes)

- **Laravel's Rate Limiting Features:**
  - Describe Laravel's built-in rate limiting capabilities, focusing on the `ThrottleRequests` middleware.
  - Explain how to customize rate limiting rules according to the application's routes, guarding against excessive requests that could lead to a service being overwhelmed.
- **Implementing Advanced Rate Limiting:**
  - Discuss the use of dynamic rate limiting based on application state or user behavior.
  - Code Example:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;
```

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000);
});

RateLimiter::for('login', function (Request $request) {
    return $request->user()->isAdmin()
            ? Limit::perMinute(50)
            : Limit::perMinute(5)->by($request->ip());
});
```

4. Hands-On: Setup and Configure Rate Limiting and Cloudflare (30 minutes)

- **Practical Exercise Setup:**
  - Provide participants with access to a basic Laravel application and a Cloudflare account.
- **Tasks:**
  - Participants configure rate limiting in the Laravel application as per different use cases (e.g., API vs. web front-end).
  - Set up Cloudflare for the application and configure basic DDOS protection features.
- **Testing:**
  - Instruct participants on how to simulate high traffic to test the rate limiting and see Cloudflare in action.

5. Discussion and Q&A (10 minutes)

- **Review:**
  - Participants share their configurations and describe the choices they made.
- **Q&A:**
  - Address any questions regarding DDOS mitigation, Cloudflare settings, or Laravel configurations.

Additional Resources

- Provide links to Cloudflare documentation, Laravel's official documentation on rate limiting, and best practices guides for DDOS mitigation.

By the end of this session, participants should have a practical understanding of how to protect a Laravel application from DDOS attacks using both application-level and network-level mitigation strategies. They will also gain experience in configuring real-time protection with

Cloudflare, equipping them with the knowledge to implement these techniques in their own projects.

- Final Security Thoughts
  - LocalStorage - any scripts that you load in the page can access the localStorage - don’t use user data in local storage
  - 3rd party scripts - what is running on the key/vulnerable points in the user journey? Audit.
    - script that on key press was sending to the data
    - Ticketmaster - checkout page - help widget - pulling all data from page - including the credit card details - the 3rd party was then hacked
  - https://cheatsheetseries.owasp.org/cheatsheets/Laravel_Cheat_Sheet.html
  -