Designing Security-First Symfony Apps

Secure Software Development Life-cycle

S-SDLC

S-SDLC

Security Policies	Risk Assessment	Secure Coding	Supply Chain Security	Incident Management
Data Protection	Threat Modeling	Risks as Code	Secure Code Review	Secure Secrets
Secure Architecture	Security Backlog	Security Testing	Security Monitoring	Secure Delivery
Secure Design	Security Training(s)	Static Analysis	Vulnerability Management	Psychology

Wait
What?
Why?
How ...

S-SDLC & Symfony

I just want to deliver a *safe* Symfony web app!

OWASP top 10

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Componentes
7. Identification and Authentication failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery

1 Broken Access Control

Access control enforces policy such that users cannot act outside of their intended permissions.

- Exposure of Sensitive Information to an Unauthorized Actor
- Cross-Site Request Forgery
- Insertion of Sensitive Information Into Sent Data

Symfony components:

- symfony/security-core
- symfony/security-guard
- symfony/security-csrf

Symfony Security Docs

1 Broken Access Control

Shoutout to symfony/security-core

- By default you get Role-Based Access Control (RBAC)
- Can be extended to ACL with **symfony/security-acl**
- Versatile to implement different Access Control strategies
    - Discretionary Access Control
    - Mandatory Access Control

2 Cryptographic Failures

Any data that needs protecting in transit and/or at rest is at risk of cryptographic failures, or lack thereof.

- Use of Hard-coded Password
- Broken or Risky Crypto Algorithm
- Insufficient Entropy

Symfony components:

- symfony/framework-bundle
- symfony/password-hasher

3 Injection

Injection is an attacker's attempt to send data to an application in a way that will change the meaning of commands sent to an interpreter.

- Cross-Site Scripting
- SQL Injection
- External Control of File Name or Path

Symfony & Friends:

- symfony/html-sanitizer
- symfony/validator
- symfony/routing[1]
- doctrine/DoctrineBundle[2]

3 Injection

[1] Symfony Routing can help filter routes and/or bind request parameters your application wants to handle, this does not mean that a **strong** Web-Server configuration is not necessary too!

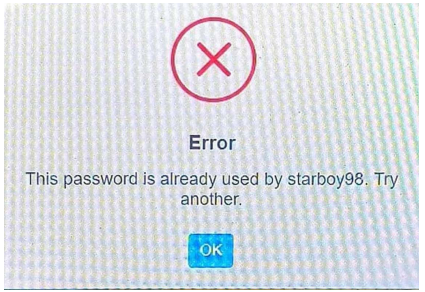
[2] Doctrine ORM and Doctrine DBAL help mitigate SQL injection via their QueryBuilder class API, allowing to bind parameters further down to PDO. It is also *recommended* to avoid raw SQL when Doctrine's API can be used to lower your source-code surface area.

```
<html>
  <body>
    <form action="https://example.com/update-email"method="POST">
      <input type="hidden" name="email" value="bad@attacker.co"/>
    </form>
    <script>
      document.forms[0].submit();
    </script>
    <!-- some content here to distract the user -->
  </body>
</html>
```

4 Insecure Design

Insecure Design specifically focuses on insecure design and not insecure implementation. Depends from project to project!

- Generation of Error Message Containing Sensitive Information
- Unprotected Storage of Credentials
- Trust Boundary Violations
- Insufficiently Protected Credentials



4 Insecure Design

Still, there are Symfony MVP components!

- symfony/messenger
  - Make unnecessary synchronous processes async
- symfony/error-handler
  - From Chaos to Control: Exception handling in Symfony
- symfony/rate-limiter
- symfony/*

5 Security Misconfiguration

Highly configurable software is vulnerable to Security Misconfiguration. This is not specific to your Symfony code, but there are still things to consider if you have a Symfony application!

- Apply static analysis to disallow usage of certain functions (`dump()`, `dd()`, …)
- Use *env* specific configuration as much as possible (`dev`, `test`, `prod`)
- Do not install `dev dependencies` in your production deployment
- Try to avoid using bundles outside of their default *env*
- Familiarise yourself with Configuring Symfony
- Familiarise yourself with Symfony Cache

6 Vulnerable and Outdated Components

Usage of vulnerable components makes you vulnerable to all known Common Vulnerabilities and Exploitations of said components.

Usage of outdated components, even when secure, you have an operational risk of disallowing a project to update other vulnerable components.

6 Vulnerable and Outdated Components

Things to consider:

- Familiarise yourself with Symfony's Release Process.
- Get a grip on your project's supply chain
  - Try to automate `composer audit` runs as a first step.

Packagist Security Monitoring

7 Identity and Authentication Failures

Confirmation of the user's identity, authentication, and session management is critical to protect against authentication-related attacks.

Symfony Components:

- symfony/security-guard
  - Form Login
  - JSON Login
  - HTTP Basic
  - Login Link
  - Access Token
  - Client Certificates
  - *Your Specific Case*

8 Software and Data Integrity Failures

Software updates, critical data, CI/CD pipelines can fail integrity checks.

- Inclusion of Functionality from Untrusted Control Sphere
- Download of Code Without Integrity Check
- Deserialization of Untrusted Data

Consider:

- Composer / (Private) Packagist
- Trusted CI/CD runners / suppliers
- Trusted Docker images
- Get a grip on your project's supply chain

9 Security Logging and Monitoring Failures

This category is to help detect, escalate, and respond to active breaches. Logging and monitoring can be challenging to test, often requiring audits and penetration tests.

- Insufficient Logging
- Improper Output Neutralization for Logs
- Omission of Security-relevant information
- Insertion of Sensitive Information into Log File

Symfony components:

- symfony/monolog-bundle
- symfony/event-dispatcher

10 Server-Side Request Forgery

SSRF flaws occur whenever a web application is fetching a remote resource without validating the user-supplied URL.

Symfony components:

- symfony/validator
  - `Url`
  - `Ip`
  - `NoSuspiciousCharacters`
- symfony/http-client
  - `NoPrivateNetworkHttpClient`

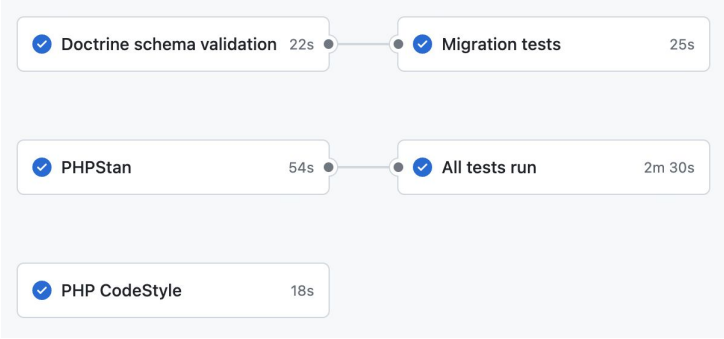
How I applied my own advice

Secure Coding Standards

- Always add a CONTRIBUTING guide
  - Inspired by NASA OpenMTC project's guide
  - Keywords from RFC 2119
- Describe Pull Request process
- Describe Design Standards
- Describe Coding Standards
  - i.e. non-nullable scalar variables
- Describe Test standards
- Describe Doctrine standards
  - Describe expected schema migration process

CI/CD: Static Analysis

- Apply **PHPStan** at level 8
  - Use `phpstan/phpstan-strict-rules` on top of L8
  - Use `phpstan/phpstan-symfony`
  - Use `phpstan/phpstan-doctrine`
  - Use `phpstan/phpstan-phpunit`
- Apply **PHP-CS-Fixer** with default Symfony ruleset
- Run Doctrine schema validation(s)



CI/CD: Supply Chain Checks

- Use **Dependabot** because we have Github enterprise
  - Renovate is a good alternative
  - Violinist is a good alternative
  - Custom job with composer is a good alternative
- Use **Blackduck** for license and operational checks
- Use **Trivy** for image scanning
- Try **Sonarqube** for potential attack vectors

Incident & Vulnerability Management

- Have a process that describes what is an Incident
  - Have steps how is an Incident resolved
- Have a process that describes what is a Vulnerability
  - Have steps how is a Vulnerability resolved
- After every Incident update our logging and monitoring
  - ProTip: The easier your logs are to read, the happier your team will be

Recap

1. Look up SSDLC
2. Start with OWASPT top 10
3. Create a team process

Thank you!

Symfony Online
JUNE 2024

JUNE 6 - 7, 2024