Architecture(s) et application(s) Web

Télécom SudParis

13/11/2023

CSC4101 - Sécurité, gestion des erreurs

Table des matières

Objectifs de cette séquence

Cette section présente les enjeux généraux de sécurité qui s'imposent aux concepteurs d'applications Web, avec un catalogue rapide de quelques problèmes courants.

1 Sécurité

1.1 Objectifs

— Éviter que les données ne soient corrompues
— Garantir le bon fonctionnement du service
— Garantir que les permissions accordées aux utilisateurs soient respectées

Danger : toute entrée de donnée

1.2 Exemple d'attaques

1.2.1 Injection SQL

— **Objectif** : garantir que les requêtes SQL sont sans effet de bord
— Nettoyer les données servant à constituer ces requêtes SQL

Cf. `http://php.net/manual/fr/security.database.sql-injection.php`

— code PHP

```php
<?php

  $id = $_GET['id'];
  $sql = "SELECT username
      FROM users
      WHERE id = $id";
...
```

— requête envoyée à :
`http://localhost/?id=-1%20UNION%20SELECT%20password%20FROM%20users%20where%20id=1`
— argument de la requête GET :
`id : -1 UNION SELECT password FROM users where id=1`

Exploits of a Mom (xkcd)

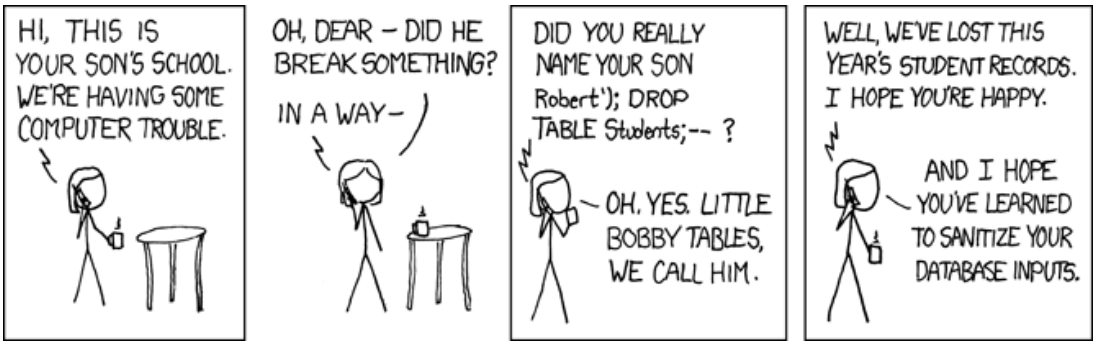
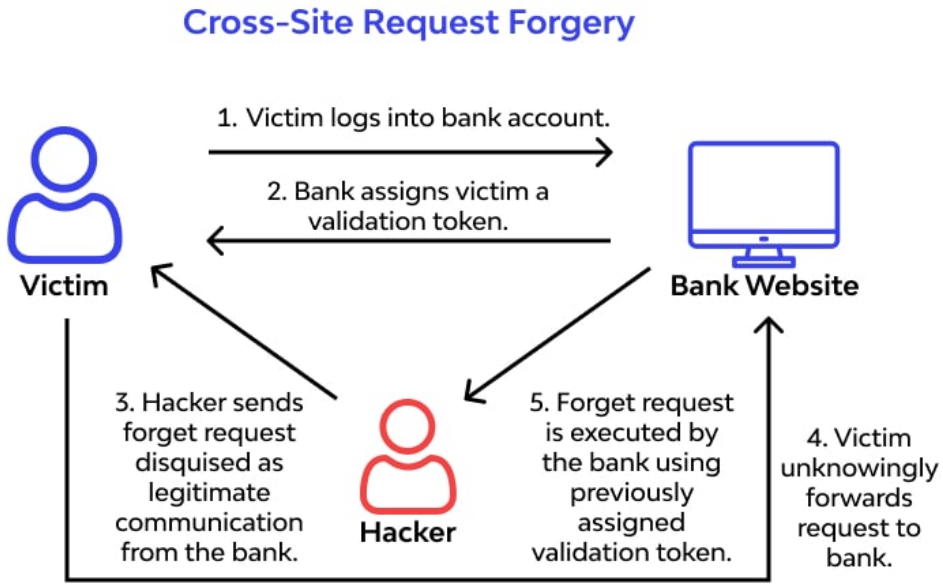


Figure 1 – *Exploits of a Mom (xkcd)*

Danger : le PHP codé comme montré au début de la séance 5 où on manipule directement les données reçues, type `_GET[]` ou `_POST[]`.
Si ces données sont injectées dans la construction de requêtes en base, on peut amener le serveur à intégrer du contenu malicieux dans la page, ou dans la base.
Ne surtout pas passer les données telles-quelles à la base. Doctrine nous évite celà.

1.2.2 *Cross Site Request Forgery* (CSRF)

Induire l'exécution de transition dans le graphe d'états de l'application en jouant des requêtes à l'insu de l'utilisateur.
`https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`



Source : `https://www.wallarm.com/what/what-is-cross-site-request-forgery` Exemple de transaction sur site banque : virement de 100 euros à destination d'*olivier* :

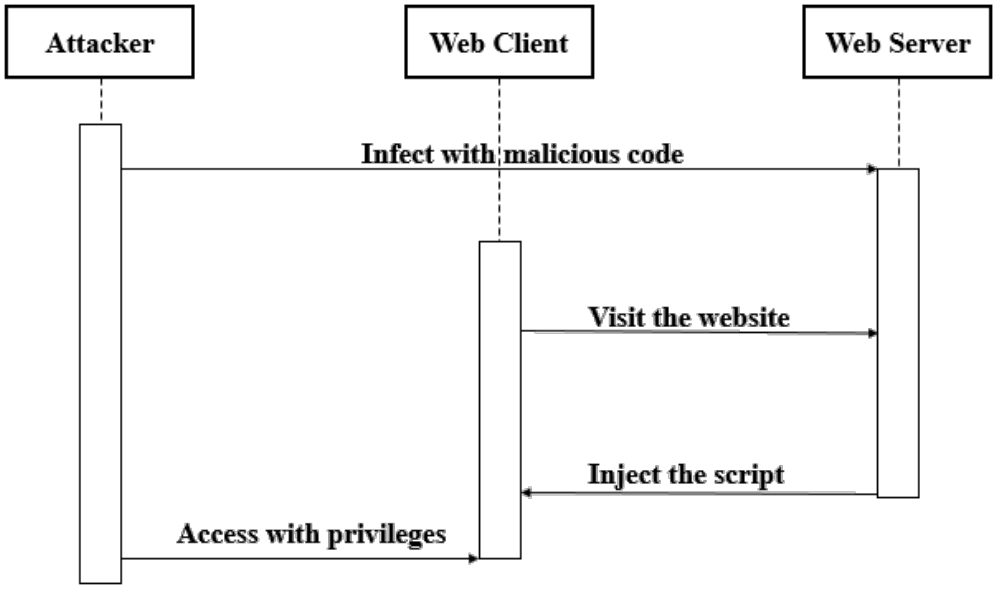
```
POST http://bank.com/transfer.do HTTP/1.1

acct=olivier&amount=100
```

Objectif : faire exécuter ce formulaire à l'insu de l'utilisateur :

```
<form action="http://bank.com/transfer.do" method="POST">
  <input type="hidden" name="acct" value="philippe"/>
  <input type="hidden" name="amount" value="100000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

1.2.3 *Cross-site scripting* (XSS)



Source : Michel Bakni, CC BY-SA 4.0, via Wikimedia Commons

— Donne accès aux données de l'application à un script tiers

— cookies
— jetons session

https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

— Exemple de page :

```php
<?php

  echo '<html>';
  // ...
  echo $_POST["name"];
```

— Requête POST : injection de `name` qui vaut
`name=%3Cscript%3Ealert%2842%29;%3C/script%3E`
— Résultat :

```
<script>alert(42);</script>
```

Tout dépend où se trouve le `echo` …
Solution possible :

```php
<?php

   echo '<html>';
   // ...
   echo htmlentities($_POST["name"]);
```

Résultat :

```
<html>
   ...
   &lt;script&gt;alert(42);&lt;/script&gt;
```

Normalement, c'est plus sûr…

Deuxième danger, les données sont injectées dans la construction du contenu HTML des pages visualisées par le navigateur, qui peut ainsi être amené, par exemple, à exécuter un script JS dans le contexte courant de l'utilisateur.
Ici, le système de gabarits va nous aider. On en construit pas le HTML à la main, et les données sont nettoyées.

1.3 Avantage du framework

Le *framework* permet de gérer automatiquement certaines précautions

1.3.1 *Sanitization*

— **Objectif** : ne pas transmettre au Modèle des données avec des caractères indésirés
— Supprime de manière automatique les caractères indésirés,
— Champs des formulaires associés à un type de donnée pour restreindre les plages de valeur

En plus, le développeur peut intégrer des règles de validation des données pour éviter d'avoir des données incohérentes par rapport aux règles de gestion de l'application
Cf. `https://symfony.com/doc/current/validation.html`

1.3.2 Doctrine

On ne manipule pas SQL directement.

1.3.3 Gabarits

On ne manipule pas HTML directement

1.3.4 Sécurité des formulaires

Protection CSRF
Objectif : éviter d'arriver sur la soumission d'un formulaire directement, contrôler les transitions dans le graphe HATEOS

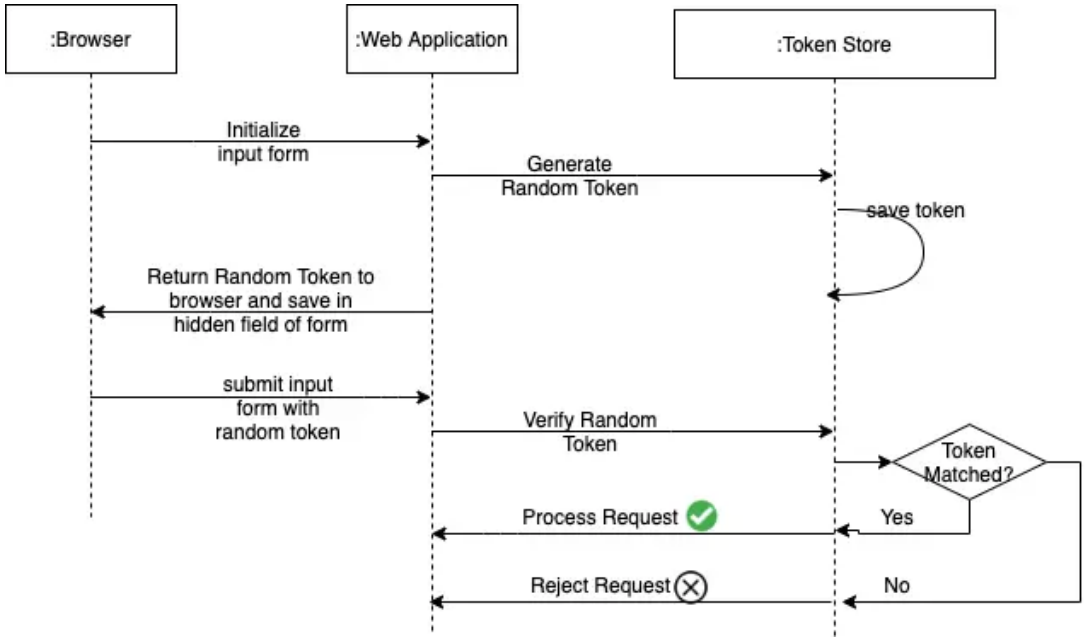


Figure 2 – Identifying Legitimate Requests with an CSRF Token

Source : `https://reflectoring.io/complete-guide-to-csrf/`

- Contrôle que la gestion de la soumission d'un formulaire suit immédiatement l'affichage du formulaire
- Paramètre caché généré de manière automatique (`_token` en Symfony)
- Associé à une mémorisation du paramètre côté serveur (dans la session)

Construction du formulaire :

1. stockage dans la session

```
Session Attributes

_csrf/post : "RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I"
```

2. génération du formulaire envoyé au client

```
<form>
  ...
  <input type="hidden" id="post__token"
         name="post[_token]"
         value="RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I" />
</form>
```

Réception de la requête POST :

- données du POST

```
[
  "title" => "Lorem Ipsum"
  "summary" => ""
  "content" => "Lorem Ipsum"
  "publishedAt" => "2017-11-26T20:07:34+01:00"
  "tags" => ""
  "_token" => "RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I"
]
```

— code du contrôleur (dans `isValid()`) :

```php
if ($this->isCsrfTokenValid('token_id', $submittedToken)) {
        // ...
}
```

Attention, Symfony intègre du CSRF dans certains formulaires, mais on doit le gérer manuellement lors de certains opérations avancées.
Par exemple, c'est le cas dans le code qui est généré dans les contrôleurs et gabarits obtenus avec `make:crud`. On y gère des formulaires particuliers pour la suppression des entités, via de fausses méthodes « DELETE » simulées par des POSTS. On doit y gérer manuellement l'ajout d'un jeton CSRF dans le gabarit (type `_delete_form.html.twig`) et le vérifier ensuite dans les méthodes `delete()` des contrôleurs.

1.4 Inconvénient du framework

Sécurité des dépendances ?
S'abonner à des listes de notification de vulnérabilités, pour être notifié des alertes (CVE).
- pour PHP : `https://wiki.php.net/cve`
- pour le reste : `https://github.com/FriendsOfPHP/security-advisories`

1.5 Sécurité des utilisateurs

- Protéger les données des usagers :
  - argent
  - infos personnelles (RGPD)
  - vie privée
  - réputation
- Lutter contre surveillance
  - chiffrement TLS (HTTPS)
  - accessibilité via TOR (domaine en `.onion`)

Exemple : le profil d'un usager ne doit pas être accessibles par d'autres.
RGPD : Réglement Général (Européen) sur la Protection des Données
Contraintes : garder des traces du consentement au traitement des données personnelles.

1.6 Contrôle d'accès

déjà vu

1.7 HTTPS *everywhere*

- BD : `https://howhttps.works/`
- Cf. LetsEncrypt, pour générer des certificats TLS.

1.8 Sécurité des applis Web

Approfondir :
- `https://phptherightway.com/#security`
- Guidelines Web Security Mozilla
- OWASP Web Security Testing Guide

— Recommandations pour la sécurisation des sites web ANSSI

Ressource vraiment intéressante : *Open Web Application Security Project* (OWASP)

2 Gestion des erreurs

Cette section aborde la gestion des erreurs, notamment à travers l'utilisation des exceptions.

2.1 Code de réponse

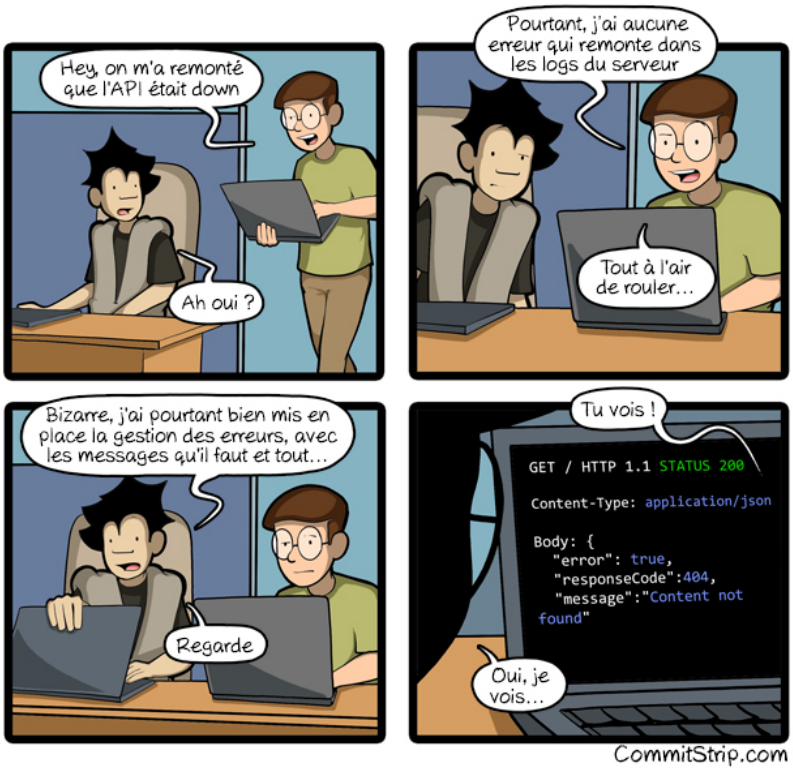


Figure 3 – *HTTP Headers FTW (CommitStrip)*

HTTP Headers FTW par CommitStrip

En cas de problème, ne pas renvoyer 200 dans la réponse HTTP, contrairement à ce qui est fait dans cet exemple humoristique.

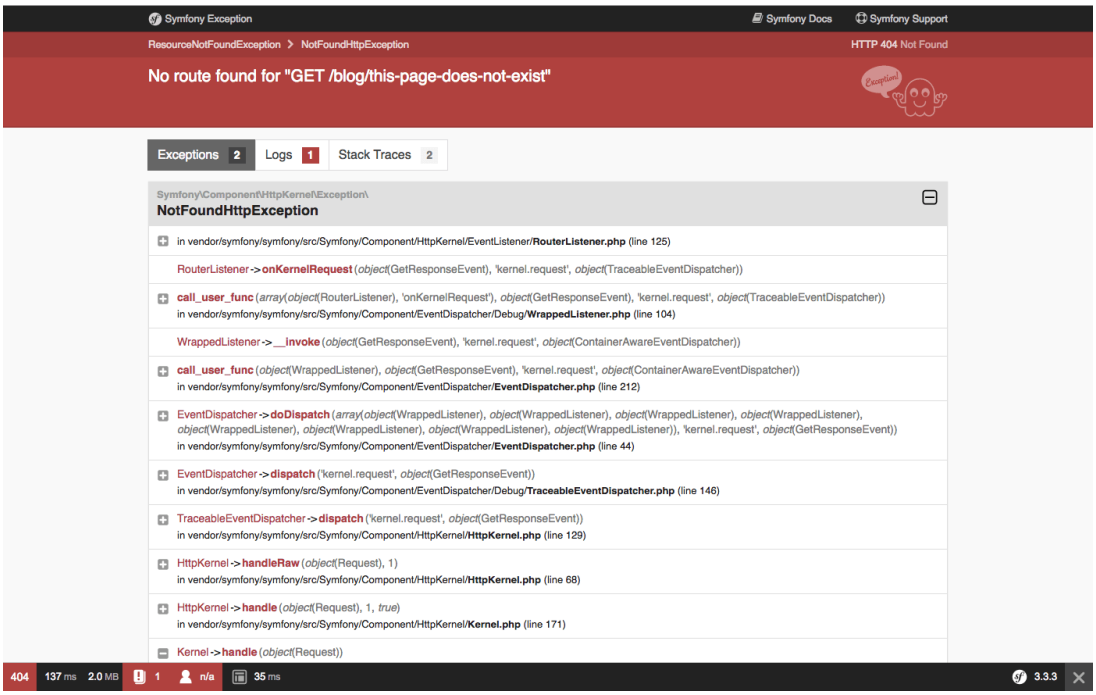
2.2 Exceptions

```php
throw new OutOfBoundsException("not good");
```

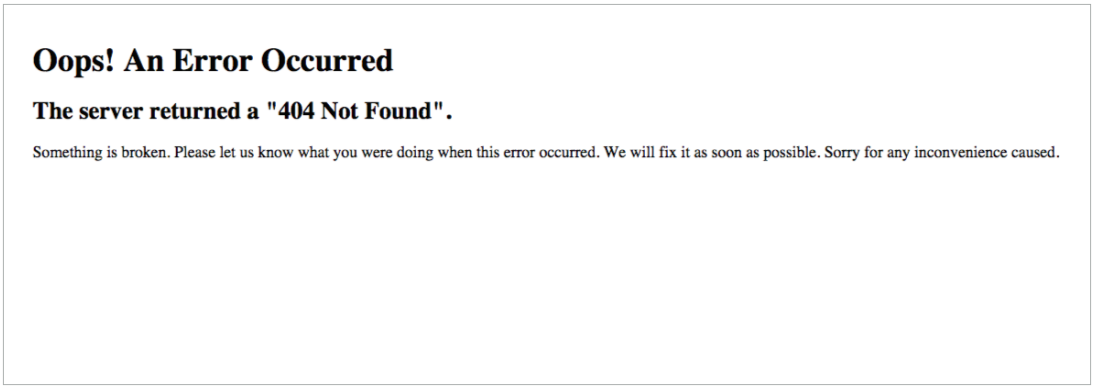
```php
try {
    ...
} catch (RequestException $e) {
    ...
}
```

Certains d'entre-vous approfondiront la programmation des exceptions dans le module CSC4102.

2.3 Environnement de dév



2.4 Environnement de prod ?



2.5 Pages d'affichage des erreurs dans les gabarits

Cf. How to Customize Error Pages.

2.6 Déployer

Checklists : `https://symfony.com/doc/current/deployment.html`

3 Bugs, Qualité

Cette section évoque l'enjeu de la qualité du code, qui permet de garantir que l'application aura le fonctionnement attendu, et qu'ainsi, on évitera des problèmes de corruptions des données du modèle, ou de fuite d'infos, par exemple.

3.1 Éviter les bugs

— Coder
— **Tester**

PHPUnit

3.2 Tests manuels ?

Vraiment ?

3.3 Tests unitaires

— Tests des méthodes des objets du modèle, en dehors du contexte Web.
— PHPUnit (très similaire à JUnit)

3.4 Tests « système »

Tester de bout en bout, systématiquement :

1. Fabriquer une « requête » semblable à celle entrant depuis un client HTTP : accès à une route avec ses arguments
2. Traiter
3. Vérifier la réponse :
   — code de retour HTTP
   — présence de données dans le contenu (HTML)

Cf. `https://symfony.com/doc/current/testing.html`

```php
public function testPageIsSuccessful($url)
{
        $client = self::createClient();
        $client->request('GET', $url);
        $this->assertTrue($client->getResponse()->isSuccessful());
}
```

Take Away

- sécurité de l’application :
  - données entrées ou fournies
- sécurité des utilisateurs
- qualité du code
- le framework fait les choses « bien », et c’est tant mieux !
- ne plus programmer en PHP « basique » comme autrefois

Postface

Crédits illustrations et vidéos

— Exploits of a Mom by xkcd - Creative Commons Attribution-NonCommercial 2.5 License.
— HTTP Headers FTW par CommitStrip - utilisation non-commerciale autorisée
— Cross-site scripting attack sequence diagram par Michel Bakni, CC BY-SA 4.0, via Wikimedia Commons
— WHat is cross-site request forgery - Wallarm
— Identifying Legitimate Requests with an CSRF Token - Reflectoring