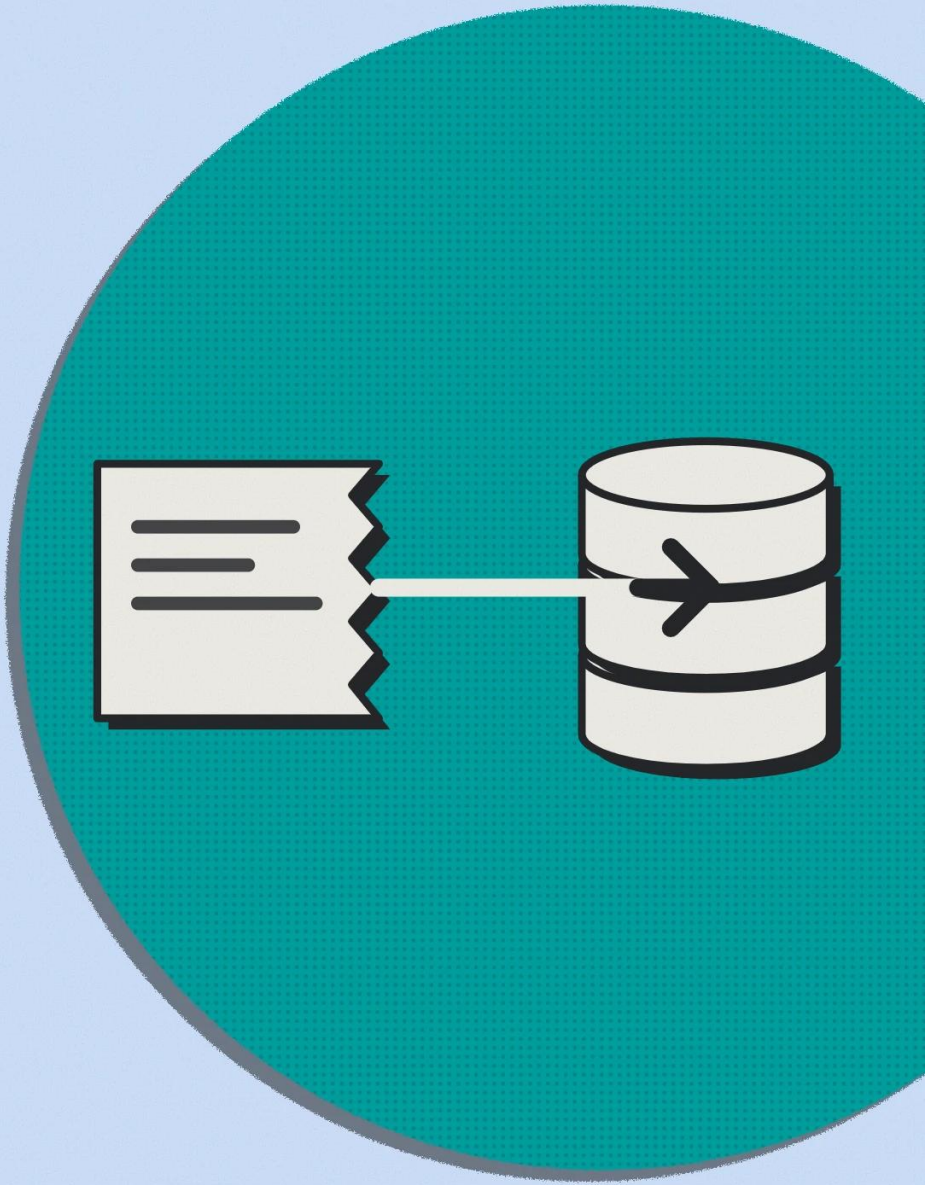
2.06

TEMPS 2 · LE TAMAGOTCHI TROUÉ · SÉANCE 1

L'injection SQL

Thibault Six · IFOSUP Wavre · 2026-2027



? En octobre 2015, des pirates lisent les données des clients de TalkTalk. Par où sont-ils entrés ?

Une injection SQL dans trois pages oubliées a livré les données de 156 959 clients de TalkTalk.

octobre 2015

trois pages web oubliées de TalkTalk

héritées d'un rachat, sur une base de données ancienne



une injection SQL → **la base de données** →

156 959 clients

dont les coordonnées bancaires de 15 656 d'entre eux

400 000 £ d'amende, un record à l'époque

la même faille avait déjà servi deux fois plus tôt en 2015

Une injection SQL glisse du code dans une requête de base de données à la place d'une simple donnée.

Le magasin de votre Tamagotchi a la même : sa recherche d'articles lit toute la base.

Suivez la flèche : de vieilles pages, une seule faille, toute la base. L'amende vient du régulateur britannique.

? Mallory tape un mot dans « Chercher un article ». Que devient ce mot dans la requête SQL ?

`whereRaw()` colle le mot tapé dans la requête SQL : `os` y reste une donnée entre apostrophes.

Chercher un article

os

le texte tapé dans le champ

app/Http/Controllers/ShopController.php

```
if ($search !== '') {
    $items->whereRaw("name LIKE '%{$search}%'");
}
```

extrait

`whereRaw()` écrit un bout de SQL brut, tel quel, sans qu'Eloquent le relise

la requête lancée

```
select * from "items" where name LIKE '%os%' order by "name" asc
```

la table des articles

le mot cherché reste une donnée, entre les apostrophes de la chaîne

`LIKE '%os%'` : un nom qui contient `os` ; les deux `%` valent n'importe quels caractères avant et après le mot cherché

Le danger vient d'une saisie qui sort de ces apostrophes.

Suivez le mot `os` : du champ au contrôleur, puis dans la requête. Il y arrive collé au texte SQL, sans séparation.

? Mallory cherche L'os, un article qui existe. Pourquoi une erreur s'affiche-t-elle à la place ?

L'apostrophe de `L'os` ferme la chaîne SQL trop tôt : la base lit la suite comme du code.

```
select * from "items" where name LIKE '%L'os%' order by "name" asc
```

la requête lancée pour L'os

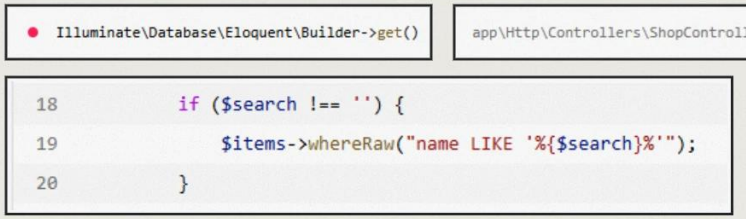
l'apostrophe de L'os ferme la chaîne — **la base lit os%' comme du code, ne le comprend pas, et s'arrête**

la fenêtre d'erreur d'Inertia s'ouvre sur la page de débogage de Laravel




```
SQLSTATE[HY000]: General error: 1 near "os": syntax error
```

la pile pointe `ShopController.php:23`, l'appel `get()` ; l'extrait montre la ligne 19, celle du `whereRaw`



la même requête, affichée en toutes lettres par la page

SQL: select * from "items" where name LIKE '%L'os%' order by "name" asc)

Cette page de débogage est déjà une fuite : elle donne la requête exacte et le fichier.

Lisez la requête entre les apostrophes : la chaîne s'arrête après L. La suite n'est plus une donnée.

? Aucun article ne s'appelle zzz. Pourquoi cette saisie rend-elle tout le catalogue ?

La saisie ajoute une condition toujours vraie et efface la fin : tout le catalogue revient.

`zzz%' OR '1'='1' --` ce que Mallory tape

```
select * from "items" where name LIKE '%zzz%' OR '1'='1' --%' order by "name" asc
```

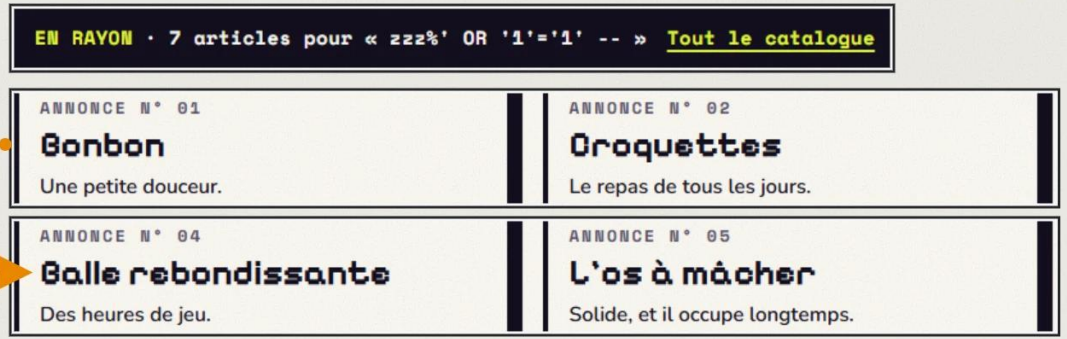
la requête lancée

l'apostrophe ferme la chaîne

une condition toujours vraie : chaque ligne passe

deux tirets : un commentaire SQL, qui efface la fin de la ligne, le order by compris

le magasin de Mallory

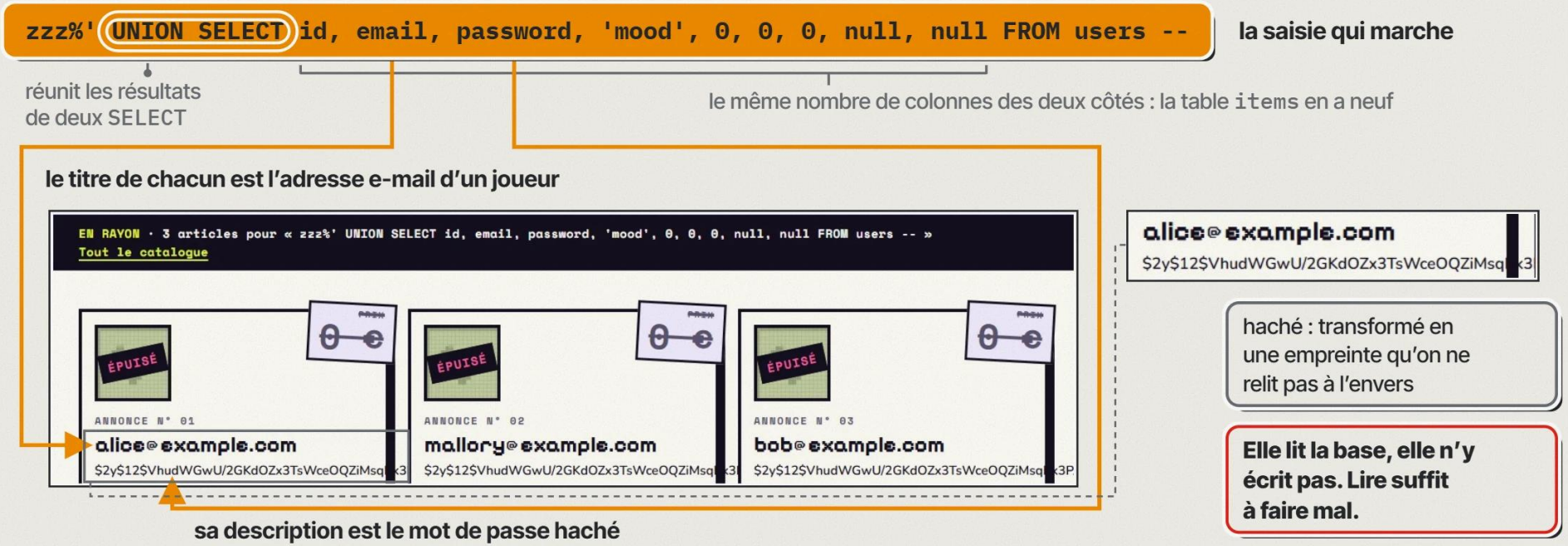


un indice le trahit : la liste n'est plus triée par nom

Lisez la requête en trois morceaux : la chaîne fermée, la condition ajoutée, la fin effacée. La page montre le résultat.

? Comment Mallory fait-elle afficher les comptes des joueurs par la recherche des articles ?

Un `UNION` colle la table `users` sous les articles : chaque e-mail devient le titre d'une annonce.



Suivez les deux flèches : la colonne `email` devient le titre, la colonne `password` devient la description. La saisie est devenue du code.

? Avant de corriger, quels contrôles du projet voient déjà le trou du ShopController ?

PHPStan refuse ce `whereRaw()` et les deux tests de la faille sont rouges avant votre correction.

PHPSTAN

```
$ vendor/bin/phpstan analyse
  19     Parameter #1 $sql of method Illuminate\Database\Eloquent\Builder
         <App\Models\Item>::whereRaw() expects …|literal-string,
         non-falsy-string given.

 [ERROR] Found 1 error
```

PHPStan lit le code sans le lancer et signale les erreurs de type

`literal-string` : une chaîne écrite par le développeur, ou composée seulement de telles chaînes

LE TEST DE LA FAILLE

```
$ php artisan test --filter=Flaw06
   FAIL  Tests\Feature\Flaws\Flaw06SqlInjectionTest
  × a search with an apostrophe works
  × a search cannot rewrite the query
  Tests:    2 failed (10 assertions)
```

le premier échoue sur l'erreur serveur (500) de l'apostrophe, le second parce que l'injection OR rend des articles au lieu d'aucun

Cette erreur arrête aussi `composer test`, le contrôle complet : Pint, qui vérifie la mise en forme du code, PHPStan, puis les tests.

En haut, PHPStan refuse une saisie là où il attend du texte écrit par le développeur. En bas, les deux tests rejouent l'attaque.

? Comment écrire la recherche pour que la saisie ne soit jamais lue comme du SQL ?

Deux écritures sûres : `where()` d'Eloquent, ou `whereRaw()` avec un ? et un tableau.

Dans le texte de la requête, la valeur cherchée est remplacée par un ?. Les valeurs de remplacement sont données à part, dans un tableau.
C'est la **requête préparée**, vue au cours de PHP. Ce ? s'appelle un **paramètre lié**.

```php
$items->whereRaw("name LIKE '%{$search}%'");
```

la ligne trouée : colle la saisie au milieu du texte SQL

1. where() d'Eloquent, la forme par défaut

```php
$items->where('name', 'like', "%{$search}%");
```

Laravel écrit lui-même le ?

2. whereRaw() écrit comme une requête préparée

```php
$items->whereRaw('name like ?', ["%{$search}%"]);
```

le ? dans le texte — la valeur dans un tableau

pour le SQL qu'Eloquent ne sait pas écrire

Pour la recherche L'os, Laravel envoie le texte `where "name" like ?` **, et la valeur** `%L'os%` **à part.**

Les deux écritures protègent de la même façon. Se prémunir contre l'injection SQL est facile si vous n'oubliez pas ces deux formes.

? Qu'est-ce qui doit être vrai à la fin ?

À vous : rejouez les trois saisies, corrigez la ligne, puis montrez PHPStan et les tests verts.

À FAIRE

1. Connecté comme Mallory, tapez les trois saisies dans « Chercher un article », à `tamagotchi.test/magasin`.
2. Lancez `vendor/bin/phpstan analyse` et `php artisan test --filter=Flaw06` : les deux sont rouges. Sous PowerShell, écrivez `.\vendor\bin\phpstan analyse`.
3. Dans `ShopController`, remplacez la ligne trouée par la première écriture, puis relancez les deux contrôles.

À VÉRIFIER

```
[OK] No errors
```

```
 PASS  Tests\Feature\Flaws\Flaw06SqlInjectionTest
✓ a search with an apostrophe works
✓ a search cannot rewrite the query
Tests:    2 passed (20 assertions)
```

`php artisan test` passe de 16 tests rouges à 14

la même saisie UNION ne rend plus que « 0 article »

EN RAYON · 0 article pour « zzz%' UNION SELECT

RAYON VIDE

Aucun article ne correspond

L'os retrouve son article

EN RAYON · 1 article pour « L'os »

ANNONCE N° 01

L'os à mâcher

DANS VOTRE CARNET DES FAILLES

Une requête SQL ne se construit jamais en collant du texte venant de l'utilisateur. La saisie passe par un paramètre lié.

Trois étapes à gauche, les preuves à droite : l'analyse, les tests, puis les deux recherches rejouées après la correction.