**Rapport : Simulation d'une injection SQL avec détection Suricata**

**1. Création de la base de données et de la table users**

**a) Ouverture de phpMyAdmin**

* Depuis un navigateur sur la machine Windows (où XAMPP est installé), accéder à l’interface phpMyAdmin via l’URL :

arduino

Copier le code

http://localhost/phpmyadmin

* phpMyAdmin permet d’administrer les bases de données MySQL/MariaDB via une interface web simple.

**b) Création de la base de données vulnapp**

* Cliquer sur l’onglet **Nouvelle base de données** en haut à gauche.
* Saisir le nom :

nginx

Copier le code

vulnapp

* Choisir l’interclassement par défaut (utf8mb4\_general\_ci) qui supporte les caractères Unicode.
* Cliquer sur **Créer**.

**c) Création de la table users**

* Sélectionner la base vulnapp dans la liste à gauche.
* Cliquer sur **Nouvelle table**.
* Saisir le nom : users
* Nombre de colonnes : 3
* Cliquer sur **Exécuter**.

**d) Définition des colonnes**

Dans la fenêtre suivante, remplir les champs pour chaque colonne :

| **Nom** | **Type** | **Longueur/Valeur** | **Null** | **Valeur par défaut** | **Extra** | **Description** |
| --- | --- | --- | --- | --- | --- | --- |
| id | INT | 11 | Non |  | AUTO\_INCREMENT | Identifiant unique, clé primaire |
| username | VARCHAR | 50 | Oui | NULL |  | Nom d’utilisateur |
| password | VARCHAR | 50 | Oui | NULL |  | Mot de passe (en clair ici) |

* Cliquer sur **Enregistrer** pour créer la table.

**e) Insertion des données**

* Cliquer sur l’onglet **SQL**.
* Copier-coller la requête suivante pour insérer deux utilisateurs de test :

sql

Copier le code

INSERT INTO users (username, password) VALUES

('alice', 'pass123'),

('bob', 'qwerty');

* Cliquer sur **Exécuter**.

**2. Base de données utilisée**

La table users ainsi créée a la structure suivante :

| **#** | **Nom** | **Type** | **Null** | **Valeur par défaut** | **Extra / Description** |
| --- | --- | --- | --- | --- | --- |
| 1 | id | int(11) | Non | Aucun | AUTO\_INCREMENT - Clé primaire |
| 2 | username | varchar(50) | Oui | NULL | Nom d’utilisateur |
| 3 | password | varchar(50) | Oui | NULL | Mot de passe |

Cette table contient deux utilisateurs test alice et bob.

**3. Application PHP vulnérable**

Le fichier login.php hébergé sur la machine Windows contient le code suivant, volontairement vulnérable pour permettre les injections SQL :

php

Copier le code

<?php

$conn = new mysqli("localhost", "root", "", "vulnapp");

if ($conn->connect\_error) {

die("Connexion échouée : " . $conn->connect\_error);

}

if (isset($\_GET['username'])) {

$username = $\_GET['username'];

// Requête SQL vulnérable sans échappement ni guillemets autour de la variable

$sql = "SELECT \* FROM users WHERE username = $username";

$result = $conn->query($sql);

if ($result && $result->num\_rows > 0) {

echo "Utilisateur trouvé !";

} else {

echo "Aucun utilisateur trouvé.";

}

} else {

echo "Utilisez /login.php?username=...";

}

$conn->close();

?>

**Explications :**

* Le paramètre username est pris directement depuis l’URL GET sans aucun filtrage ni validation.
* La requête SQL est construite en injectant la variable brute sans guillemets, ce qui permet d’injecter des commandes SQL complètes.
* Cette configuration est uniquement à des fins pédagogiques pour démontrer une injection SQL.

**4. Injection SQL et test**

Depuis la machine Kali Linux, les commandes suivantes ont été utilisées :

* **Requête normale** (recherche utilisateur "alice") :

bash

Copier le code

curl "http://192.168.60.131/login.php?username='alice'"

Résultat affiché :

nginx

Copier le code

Utilisateur trouvé !

* **Injection SQL** (tautologie OR 1=1) pour contourner la validation :

bash

Copier le code

curl "http://192.168.60.131/login.php?username='alice'%20OR%201=1--"

Résultat affiché :

nginx

Copier le code

Utilisateur trouvé !

**Explications :**

* Le paramètre username injecté correspond à la chaîne SQL :

lua

Copier le code

'alice' OR 1=1--

* La requête finale envoyée à la base devient :

sql

Copier le code

SELECT \* FROM users WHERE username = 'alice' OR 1=1--

* L’expression OR 1=1 est toujours vraie, ce qui fait que la requête retourne tous les utilisateurs, contournant ainsi la logique d’authentification.

**5. Règles Suricata pour la détection**

Pour détecter ces attaques, trois règles Suricata ont été écrites et ajoutées dans le fichier local.rules sur la machine Ubuntu Server :

text

Copier le code

alert http any any -> any any (msg:"SQLi attempt - tautology OR 1=1"; flow:to\_server,established; http\_uri; content:" OR 1=1"; nocase; classtype:web-application-attack; sid:1000001; rev:1;)

alert http any any -> any any (msg:"SQLi attempt - UNION SELECT"; flow:to\_server,established; http\_uri; content:"UNION SELECT"; nocase; classtype:web-application-attack; sid:1000002; rev:1;)

alert http any any -> any any (msg:"SQLi attempt - SQL comment"; flow:to\_server,established; http\_uri; content:"--"; nocase; classtype:web-application-attack; sid:1000003; rev:1;)

**6. Fonctionnement de la détection**

* Suricata inspecte en temps réel le trafic HTTP transitant sur le réseau.
* Lorsque l’URI HTTP contient l’une des chaînes suspectes ( OR 1=1, UNION SELECT, ou --), Suricata génère une alerte avec le message correspondant.
* Ces alertes peuvent être consultées en direct via la commande :

bash

Copier le code

sudo tail -f /var/log/suricata/fast.log

* Ou dans un format JSON plus détaillé via :

bash

Copier le code

sudo tail -f /var/log/suricata/eve.json | jq '. | select(.alert)'

**7. Conclusion**

Ce test a permis de démontrer :

* La mise en place d’une application web volontairement vulnérable à l’injection SQL.
* Le passage d’une injection SQL fonctionnelle depuis Kali Linux.
* La détection en temps réel par Suricata grâce à des règles personnalisées ciblant des signatures d’attaques SQL.

Cette démonstration illustre bien l’importance d’un IDS/IPS comme Suricata pour sécuriser les systèmes contre les attaques web courantes.

# Rapport sur la Simulation d’Injection SQL et Détection avec Suricata

## 1. Contexte et Objectif

L’objectif est de simuler des attaques d’injection SQL dans un environnement contrôlé, pour ensuite détecter ces attaques à l’aide d’un IDS/IPS comme Suricata. La base de données utilisée est une base MySQL/MariaDB simple nommée vulnapp avec une table users qui contient trois colonnes : id (int auto-incrémenté, clé primaire), username (varchar(50)) et password (varchar(50)).

## 2. Structure de la base de données

La table users a été créée avec la structure suivante :

| **Nom** | **Type** | **Null** | **Valeur par défaut** | **Extra** |
| --- | --- | --- | --- | --- |
| id | int(11) | Non | Aucun | AUTO\_INCREMENT |
| username | varchar(50) | Oui | NULL |  |
| password | varchar(50) | Oui | NULL |  |

## 3. Description des scripts PHP pour simuler les injections SQL

### 3.1. Script login\_or.php — Injection de type OR 1=1

Ce script simule une injection classique qui contourne la vérification d’authentification avec la condition OR 1=1, toujours vraie, afin de récupérer tous les enregistrements de la table.

#### Code PHP :

php

Copier le code

<?php

$conn = new mysqli("localhost", "root", "", "vulnapp");

if ($conn->connect\_error) {

die("Connexion échouée : " . $conn->connect\_error);

}

if (isset($\_GET['username'])) {

$username = $\_GET['username'];

// Injection brute : PAS DE GUILLEMETS autour de $username

$sql = "SELECT \* FROM users WHERE username = $username";

$result = $conn->query($sql);

if ($result && $result->num\_rows > 0) {

echo "Utilisateur trouvé !";

} else {

echo "Aucun utilisateur trouvé.";

}

} else {

echo "Utilisez /login\_or.php?username=...";

}

$conn->close();

?>

#### Explications détaillées :

* La variable $username est récupérée directement depuis la requête GET, sans échappement ni ajout de guillemets dans la requête SQL.
* Cette absence de guillemets permet d’injecter directement du code SQL, par exemple :

lua

Copier le code

'alice' OR 1=1--

* La requête SQL générée devient alors :

sql

Copier le code

SELECT \* FROM users WHERE username = 'alice' OR 1=1--

* La condition OR 1=1 est toujours vraie, ce qui fait que la requête renvoie tous les utilisateurs.
* Le script affiche simplement « Utilisateur trouvé ! » si au moins une ligne est retournée.

#### Comment tester avec curl :

bash

Copier le code

curl "http://192.168.60.131/login\_or.php?username='alice'%20OR%201=1--"

### 3.2. Script login\_union.php — Injection de type UNION SELECT

Ce script simule une injection plus avancée, utilisant la clause UNION SELECT pour récupérer des données arbitraires ou masquer des données malicieuses.

#### Code PHP :

php

Copier le code

<?php

$conn = new mysqli("localhost", "root", "", "vulnapp");

if ($conn->connect\_error) {

die("Connexion échouée : " . $conn->connect\_error);

}

if (isset($\_GET['username'])) {

$username = $\_GET['username'];

// Injection brute : PAS DE GUILLEMETS autour de $username

$sql = "SELECT \* FROM users WHERE username = $username";

$result = $conn->query($sql);

if ($result && $result->num\_rows > 0) {

echo "Utilisateur trouvé !<br>";

while ($row = $result->fetch\_assoc()) {

echo "id: " . $row['id'] . " | username: " . $row['username'] . " | password: " . $row['password'] . "<br>";

}

} else {

echo "Aucun utilisateur trouvé.";

}

} else {

echo "Utilisez /login\_union.php?username=...";

}

$conn->close();

?>

#### Explications détaillées :

* Comme pour le script précédent, la variable $username est utilisée telle quelle dans la requête SQL, sans guillemets.
* Cela permet d’injecter une requête UNION SELECT complète, par exemple :

sql

Copier le code

'alice' UNION SELECT 1,'test','test'--

* La requête SQL finale sera :

sql

Copier le code

SELECT \* FROM users WHERE username = 'alice' UNION SELECT 1,'test','test'--

* Le script affiche alors toutes les lignes retournées, ce qui permet de voir les données injectées (test, test) dans la sortie.
* Cela simule une extraction de données via injection SQL.

#### Comment tester avec curl :

bash

Copier le code

curl "http://192.168.60.131/login\_union.php?username='alice'%20UNION%20SELECT%201,'test','test'--"

## 4. Points importants à retenir

* **Absence de guillemets dans la requête SQL** :  
  Cette technique permet d’injecter librement des chaînes SQL complexes, apostrophes comprises.
* **Injection par URL** :  
  Les attaques sont simulées via des paramètres URL dans les requêtes HTTP.
* **Affichage des résultats** :  
  Le script login\_or.php se contente d’indiquer si un ou plusieurs utilisateurs ont été trouvés.  
  Le script login\_union.php affiche en plus les données retournées ligne par ligne.
* **Danger réel** :  
  Ces scripts ne doivent jamais être utilisés en production car ils sont volontairement vulnérables.
* **Détection par Suricata** :  
  Les règles Suricata peuvent être écrites pour détecter les motifs d’injection OR 1=1, UNION SELECT, etc. dans le trafic réseau HTTP.

## 5. Exemples de détection Suricata (extraits)

Voici des exemples simplifiés de règles Suricata permettant de détecter ces injections SQL dans des requêtes HTTP GET.

rules

Copier le code

alert http any any -> any any (msg:"SQL Injection OR 1=1 detected"; flow:to\_server,established; content:" OR 1=1"; nocase; http\_uri; sid:1000001; rev:1;)

alert http any any -> any any (msg:"SQL Injection UNION SELECT detected"; flow:to\_server,established; content:"UNION SELECT"; nocase; http\_uri; sid:1000002; rev:1;)

## 6. Conclusion

Grâce à ces deux scripts PHP simples, il est possible de simuler efficacement des attaques d’injection SQL classiques et avancées. En combinant ces tests avec un IDS/IPS comme Suricata et des règles adaptées, on peut détecter ces attaques en temps réel, ce qui renforce la sécurité du système d’information.