GraphQL Vulnerabilities

Introspection, Injection e Access Control

Alessandro Iena

LinuxDay2026 @FermoLUG
`https://www.linuxfm.org/`

October 24, 2026

Licenza: CC BY-NC-SA 4.0

Indice

1. Introduzione
2. Introspection Abuse
3. Injection nei Resolver
4. Denial of Service via Query Complexity
5. Broken Access Control a Livello di Campo
6. Batching e Brute-force
7. Conclusioni

Cos'è GraphQL

- Query language per API, alternativa a REST introdotta da Facebook (2015)
- Endpoint singolo (tipicamente `/graphql`) invece di molte route
- Schema fortemente tipizzato: query, mutation, subscription
- Il client decide esattamente quali campi ricevere $\rightarrow$ niente over-fetching

GraphQL: contesto e adozione

- Nato internamente in Facebook nel 2012 (app mobile), rilasciato open source nel 2015
- Oggi governato dalla GraphQL Foundation, sotto la Linux Foundation
- Adottato da API pubbliche note: GitHub (API v4), Shopify (Admin/Storefront API), tra le altre
- Filosofia *schema-first*: si definisce prima il contratto (lo schema), poi si implementa la logica — favorisce la collaborazione tra team frontend e backend
- Non è "REST migliorato" in automatico: è una scelta architetturale con i propri trade-off, inclusi quelli di sicurezza che vedremo oggi

GraphQL vs REST: un confronto pratico

Recuperare un utente e i suoi ultimi 3 post:

REST

```
GET /users/1
GET /users/1/posts?limit=3
```

Due chiamate; ognuna restituisce una forma di dati fissa, decisa dal server.

GraphQL

```
query {
  user(id: 1) {
    name
    posts(limit: 3) { title }
  }
}
```

Una chiamata; la forma della risposta è decisa dal client.

Query, Mutation, Subscription

- **Query**: operazione di lettura (read)
- **Mutation**: operazione di scrittura (create / update / delete)
- **Subscription**: canale persistente per aggiornamenti in tempo reale (non trattata oggi)
- Sono le tre *root operation type* dichiarate nello schema: ogni richiesta è sempre una di queste tre

Anatomia di una query

Richiesta:

```
query {
  user(id: 1) {
    username
    email
  }
}
```

Risposta:

```
{
  "data": {
    "user": {
      "username": "FermoLUG",
      "email": "info@linuxfm.org"
    }
  }
}
```

- *Selection set*: i campi tra graffe, argomenti tra parentesi (es. `id: 1`)
- La risposta rispecchia esattamente la forma della query, campo per campo
- In REST questa stessa forma ridotta richiederebbe un endpoint dedicato (es. `/users/1?fields=username,email`) o uno nuovo

Anatomia di una mutation

Richiesta:

```
mutation {
  updateProfile(
    input: { bio: "Ciao" }
  ) {
    id
    bio
  }
}
```

Risposta:

```
{
  "data": {
    "updateProfile": {
      "id": "1",
      "bio": "Ciao"
    }
  }
}
```

- Argomenti spesso raggruppati in un *input type* dedicato
- In REST servirebbero verbi diversi (POST/PUT/PATCH/DELETE) su endpoint diversi; qui tutto passa per `mutation` sullo stesso endpoint
- La risposta conferma subito lo stato aggiornato, senza una query separata dopo

Lo schema: tipi e campi

```
type User {
  id: ID!
  username: String!
  friends: [User!]!
}
```

- Tipi *scalari* predefiniti: `Int`, `Float`, `String`, `Boolean`, `ID`
- Tipi *oggetto* custom (es. `User`) rappresentano le entità dell'applicazione
- `!` = campo non annullabile (obbligatorio); `[...]` = lista
- Lo schema è il "contratto" tra client e server: tutto ciò che è interrogabile deve essere dichiarato qui

Dal linguaggio naturale allo schema

Descriviamo un'app di posta elettronica a parole:

- Un utente può avere più account email
- Ogni account ha un indirizzo, una inbox, bozze, cestino e posta inviata
- Ogni email ha un mittente, una data di ricezione, un oggetto e un corpo

Traduciamo direttamente in tipi:

```
type User {
  emailAccounts: [EmailAccount!]!
}
type EmailAccount {
  address: String!
  inbox: [Email!]!
}
type Email {
  sender: String!
  subject: String
}
```

La stessa "lingua" in query e mutation

Leggere l'inbox

```
query {
  user(id: 1) {
    emailAccounts {
      address
      inbox { subject sender }
    }
  }
}
```

Inviare un'email

```
mutation {
  sendEmail(input: {
    recipient: "a@ex.com"
    subject: "Ciao"
  }) { id }
}
```

Lo schema dietro la mutation:

```
input SendEmailInput {
  recipient: String!   # obbligatorio
  subject: String
}
```

- Stessa sintassi (selection set, argomenti) per leggere e per scrivere
- Regola di business "niente email senza destinatario" → campo non annullabile: omesso, errore prima ancora che il resolver venga eseguito

Resolver: chi risponde davvero

- Per ogni campo dello schema esiste una funzione lato server: il *resolver*
- Il resolver decide *come* recuperare quel dato: query a un database, chiamata a un'altra API, calcolo in memoria...
- Lo schema descrive *cosa* è disponibile; il resolver implementa *come* viene prodotto
- Concetto chiave per capire injection e access control più avanti: la sicurezza dipende dal codice del resolver, non dallo schema

Come viaggia una richiesta GraphQL

Quasi sempre un'unica richiesta HTTP POST:

```
POST /graphql
Content-Type: application/json

{
  "query": "query { user(id: 1) { username } }",
  "variables": {}
}
```

- Un solo endpoint, indipendentemente dall'operazione richiesta
- La query viaggia come stringa dentro il body JSON
- Torna utile quando ispezioneremo le richieste nel lab live

Perché una superficie d'attacco diversa

- Un solo endpoint concentra tutta la logica applicativa
- La flessibilità delle query sposta complessità (e rischio) dal client al resolver
- Molti controlli pensati per REST (rate limiting per-endpoint, WAF su URL) non si applicano bene
- Lo schema stesso, se esposto, è già una mappa dell'applicazione

Introspection: cos'è

GraphQL espone nativamente un meccanismo per interrogare il proprio schema.

```
{
  __schema {
    types {
      name
      fields { name }
    }
  }
}
```

- Pensata per tool di sviluppo (autocomplete, doc generator)
- Utile in dev, pericolosa se lasciata attiva in produzione

Introspection: impatto

- Rivela tipi, campi, mutation e argomenti *non documentati pubblicamente*
- Espone anche campi deprecati o "nascosti" (deprecated ma ancora attivi)
- Riduce drasticamente il lavoro di ricognizione per un attaccante
- **Accenno pratico:** strumenti come `InQL` o `GraphQL Voyager` automatizzano l'estrazione e la visualizzazione dello schema una volta ottenuta la risposta

Schema leak: non solo documentazione

Con lo schema in mano, un attaccante ottiene molto più di una semplice lista di campi:

- Mappa mutation nascoste o amministrative mai esposte pubblicamente (es. `deleteUser`, `promoteToAdmin`)
- Individua campi sensibili su cui testare controlli di accesso mancanti $\rightarrow$ *access control*
- Scopre le relazioni tra tipi per costruire query annidate mirate $\rightarrow$ *DoS*
- Conosce esattamente gli argomenti attesi da ogni campo: injection mirata, non a tentativi

Lo schema non è "solo" documentazione rubata: è il punto di partenza per ogni altra vulnerabilità di questa presentazione.

Esempio pratico: dallo schema all'attacco

Estratto di una risposta di introspection:

```
mutation {
  deleteUser(id: ID!): Boolean
  promoteToAdmin(userId: ID!): User
}
```

Mutation mai documentate, mai usate dal client "ufficiale" $\rightarrow$ ma l'attaccante ora le conosce e sa esattamente che argomenti servono:

```
mutation {
  promoteToAdmin(userId: "3") {
    id
    role
  }
}
```

Senza lo schema, questo campo andava indovinato a tentativi.

Senza introspection: lo schema trapela dagli errori

Query con un campo sbagliato (`emial` invece di `email`):

```
query { user(id: 1) { emial } }
```

Risposta del server:

```
{
  "errors": [{
    "message": "Cannot query field \"emial\" on type
      \"User\". Did you mean \"email\"?",
    "locations": [{ "line": 1, "column": 28 }]
  }]
}
```

- L'errore conferma che il tipo `User` esiste e suggerisce un campo reale (*field suggestions*)
- Con tentativi ripetuti (wordlist di nomi comuni) l'attaccante ricostruisce lo schema anche a introspection disattivata
- Disattivare l'introspection non basta: va ridotta anche la verbosità degli errori

Introspection: mitigazione

- Disabilitare introspection in produzione
- Se necessaria (es. API pubbliche documentate), esporre solo un subset controllato dello schema
- Non affidarsi alla sola "security through obscurity": anche senza introspection lo schema è spesso inferibile

Injection: il problema

GraphQL definisce la *forma* della richiesta, non valida il *contenuto* degli argomenti.

```
query {
  user(id: "1 OR 1=1") { username email }
}
```

- Resolver che costruiscono query SQL/NoSQL a partire dagli argomenti sono a rischio identico all'injection "classica"
- Anche filtri di ricerca annidati o comandi di sistema possono essere vettori
- Il layer GraphQL *nasconde* l'injection dietro una struttura tipizzata, ma non la previene

Injection: mitigazione

- Query parametrizzate / prepared statement nei resolver, sempre
- Validazione e whitelisting degli argomenti anche se GraphQL già tipizza l'input
- Non fidarsi della tipizzazione dello schema come unica difesa: un `String` resta testo arbitrario

Query annidate e ricorsive

Gli schemi con relazioni permettono query profondamente annidate:

```
query {
  user(id: 1) {
    friends { friends { friends { name } } }
  }
}
```

- Ogni livello di annidamento moltiplica il numero di risoluzioni lato server
- Amplificazione esponenziale con schemi ciclici (es. `user` → `friends` → `user`)
- Una richiesta "leggera" da inviare può saturare le risorse lato server

DoS: mitigazione

- Limitare profondità massima delle query (*depth limiting*)
- Assegnare un "costo" a ogni campo e limitare il costo totale della query (*query cost analysis*)
- Timeout e limiti di risorse a livello di resolver
- Rate limiting basato sulla complessità della query, non solo sul numero di richieste HTTP

Il problema specifico di GraphQL

```
query {
  publicProfile(id: 42) {
    username
    internalNotes  # pensato per admin
  }
}
```

- In REST l'autorizzazione si applica naturalmente per endpoint; in GraphQL un unico schema espone campi eterogenei
- È facile controllare l'accesso alla *query* principale ma dimenticare un controllo su un *campo* o *mutation* annidata
- Il controllo a livello di query non garantisce che ogni campo annidato erediti la stessa policy

Access control: mitigazione

- Autorizzazione applicata a livello di resolver, non solo di query "entry point"
- Centralizzare la logica di permessi (es. middleware/direttive di autorizzazione riutilizzabili)
- Testare esplicitamente ogni campo sensibile con un utente non privilegiato

Aliasing come vettore

GraphQL permette di aliasare più istanze della stessa query in un'unica richiesta:

```
query {
  a1:    login(user: "admin", pass: "123456")    { token }
  a2:    login(user: "admin", pass: "password")  { token }
  a3:    login(user: "admin", pass: "qwerty")    { token }
  # ... centinaia di alias ...
  a1000: login(user: "admin", pass: "letmein99") { token }
}
```

- **1 richiesta HTTP = 1000 tentativi di login**: per il rate limiter è una sola richiesta
- Il rate limiting tradizionale conta *richieste HTTP*, non operazioni logiche al loro interno
- Con REST, 1000 tentativi sarebbero 1000 richieste, facili da rilevare e bloccare
- Utile per brute-force su login, enumerazione utenti, credential stuffing

Batching: mitigazione

- Limitare il numero di alias/operazioni per singola richiesta
- Rate limiting basato sul costo logico della richiesta, non sul conteggio HTTP
- Applicare lockout/backoff a livello di logica applicativa (es. per campo `login`) indipendentemente dal trasporto

Riepilogo

- Introspection: mappa gratuita dello schema se lasciata attiva
- Injection: la tipizzazione non sostituisce la sanitizzazione
- DoS: la flessibilità delle query è anche superficie di amplificazione
- Access control: va applicato a livello di campo, non solo di query
- Batching: il conteggio delle richieste HTTP non basta più come difesa

Filo conduttore

- Molte vulnerabilità GraphQL sono varianti di problemi noti (injection, access control, DoS)
- Ciò che cambia è *dove* si nascondono: dietro un unico endpoint flessibile e fortemente tipizzato
- La tipizzazione dello schema riduce alcuni errori banali, ma non sostituisce validazione, autorizzazione e limiti di risorse

Grazie per l'attenzione

Domande?