zeitkapsl reasearch OG

Web Application Audit

zeitkapsl reasearch OG
Attn. DI. Peter Spiess-Knafl
Luegerstraße 10
9020 Klagenfurt

PWND Labs GmbH
Nestroygasse 3,
9020 Klagenfurt a. W.
FN651929W

Klagenfurt,

Version: 1.1

1 Document Control

1.1 Document Details

Title	Details
Customer	zeitkapsl reasearch OG
Auditing Company	PWND Labs GmbH
Audit Type	Web Application Audit
Timeframe	Feb 1, 2026 to Apr 1, 2026
Document Version	1.1
Document Classification	TLP:CLEAR

Table 1 - Document Details

1.2 Audit Team

Name	Contact	Role
Dominik Apel	E-Mail: contact@dfpa.at	Pentester
Matthias Pleschinger	E-Mail: matthias@hexsh.at	Pentester
Jannik Schager	E-Mail: jannik.schager@pwnd.at	Pentester

Table 2 - Audit Team

1.3 List of Changes

Version	Description	Date
1.0	First Release Version	Apr 8, 2026
1.1	Post-Remediation Version	Apr 28, 2026

Table 3 - List of Changes

Table of Contents

1 Document Control 2
1.1 Document Details 2
1.2 Audit Team 2
1.3 List of Changes 2
2 Executive Summary 5
2.1 Overview 5
2.2 Identified Vulnerabilities 5
3 Methodology 7
3.1 Scope and Duration 7
3.2 Provided Resources 7
3.3 Audit Foundation 8
3.4 Risk Assessment according to CVSSv4 8
4 Findings 9
M1: Disclosure of Internal Server Errors 10
M2: Broken Access Control in GetMediaPermission 14
M3: Media-Share Corruption 18
M4: Missing Ownership Verification in Email Change Workflow 25
L1: Cache Path Traversal possible on Windows Systems 29
L2: Leaking of Encrypted Media Metadata 31
L3: Go core persists mainKey in local SQLite unnecessarily 37
L4: Missing AAD in AES-GCM allows ciphertext role confusion and swap attacks 39
I1: Analysis of flows using formal methods 42
I2: Email Enumeration Oracles 51
I3: Missing Content-Security-Policy (CSP) 60
I4: Missing Input Validation in crypto primitives 64
I5: Non-Compliance with Security Best Practices 66
I6: Outdated Dependencies 78

I7: Random node id in snowflake generator potentially allows for Snowflake Collision 89
I8: Single snowflake generator instance can produce duplicate IDs due to Race Condition in timestamp capture 91
I9: Snowflake IDs Oracles 94

A Appendix 100
A.1 Disclaimer 100
A.2 Imprint 100
A.3 List of Figures 101
A.4 List of Tables 101

2 Executive Summary

This document describes the results of the **Web Application Audit** from PWND Labs GmbH on behalf of zeitkapsl reasearch OG, conducted from **Feb 1, 2026**, to **Apr 1, 2026**. Further information on the scope and duration can be found in Chapter 3.1.

2.1 Overview

The codebase is clean and well-structured, with solid use of authentication and centralized error handling. No high-risk code-level vulnerabilities were found, but several issues arise from business logic and access control weaknesses. Dependencies are generally up to date, though some security controls, such as the Content Security Policy, could be further improved.
The audit of the cryptographic protocol and its implementations found no vulnerabilities in the core end-to-end encryption scheme. Cryptographic primitives and parameters are well-chosen and aligned with current recommendations. Several protocol flows were formally modelled in ProVerif and verified to preserve confidentiality under different threat models. The issues identified in relation to the cryptographic layer (e.g. missing Additional Authenticated Data (AAD)) require additional assumptions to exploit (such as a malicious server combined with physical observation, or main key compromise).

End-to-end encryption mitigates many attacks but does not address authorization or account management flaws, which require separate controls. Multiple access control issues were identified, including unauthorized access to other users' encrypted media, unauthorized uploads, and exposure of internal errors. Although cryptography protects sensitive data, other information leaks remain, such as cross-user metadata exposure and email enumeration.

Zeitkapsl was responsive and professional throughout the whole audit process. Any vulnerabilities we deemed sufficiently critical were immediately communicated and quickly fixed by them.

2.2 Identified Vulnerabilities

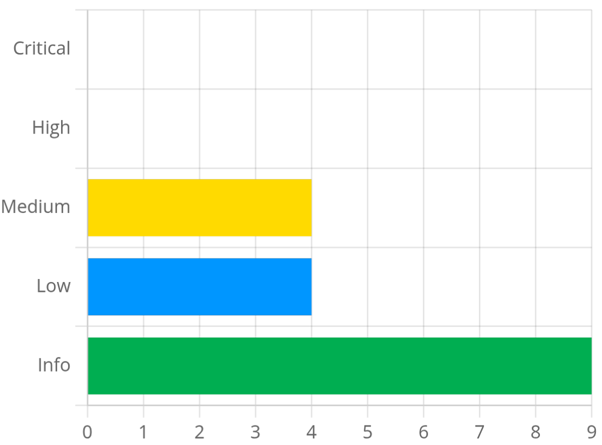


Figure 1 - Distribution of Identified Vulnerabilities

In this assessment **4 Medium**, **4 Low** and **9 Info** vulnerabilities were found.

We recommend prioritizing the remediation of these vulnerabilities. Vulnerabilities with less complex countermeasures and a Medium or lower risk should be prioritized for remediation based on the effort involved. All other vulnerabilities should be addressed as part of a continuous improvement process.

#	CVSS	Description	Page
M1	5.9	Disclosure of Internal Server Errors	10
M2	5.3	Broken Access Control in GetMediaPermission	14
M3	5.3	Media-Share Corruption	18
M4	4.1	Missing Ownership Verification in Email Change Workflow	25
L1	2.3	Cache Path Traversal possible on Windows Systems	29
L2	2.3	Leaking of Encrypted Media Metadata	31
L3	1.0	Go core persists mainKey in local SQLite unnecessarily	37
L4	1.0	Missing AAD in AES-GCM allows ciphertext role confusion and swap attacks	39
I1	0.0	Analysis of flows using formal methods	42
I2	0.0	Email Enumeration Oracles	51
I3	0.0	Missing Content-Security-Policy (CSP)	60
I4	0.0	Missing Input Validation in crypto primitives	64
I5	0.0	Non-Compliance with Security Best Practices	66
I6	0.0	Outdated Dependencies	78
I7	0.0	Random node id in snowflake generator potentially allows for Snowflake Collision	89
I8	0.0	Single snowflake generator instance can produce duplicate IDs due to Race Condition in timestamp capture	91
I9	0.0	Snowflake IDs Oracles	94

Table 4 - Identified Vulnerabilities

3 Methodology

3.1 Scope and Duration

The Audit was conducted from **Feb 1, 2026** to **Apr 1, 2026** using a time-box approach and covered **10 Person Days**.

The security audit covered the provided Git repository, the core service and server components, the web frontend, the Android and iOS applications, and the administrative-dashboard UI. Although every artifact was examined, the assessment focused on the core service and its public API, as these interfaces form the primary attack surface and are consumed by all client-side front ends. The Git commit with hash `68ffc81` was used as the standardized reference point for all auditors. In particular, the cryptographic primitives and their usage throughout the codebase were scrutinized in depth to verify correct implementation, proper key management, and resistance to known cryptographic weaknesses.

A dedicated test instance was supplied and employed throughout the assessment, with additional verification steps also performed using a local development environment.

System	Description
app.test.zeitkapsl.at	Web-Client test server

3.2 Provided Resources

Please ensure that all users and resources (e.g. Virtual Machines) provided during the audit are deprovisioned once they are no longer needed.

> No production user accounts were required for the assessment. The auditors leveraged the local development environment to evaluate the admin dashboard and created dedicated test users within the test instance to exercise the web client and the overall API.

The table below lists the accounts that were created by the auditors for testing purposes.

Username	Email	System
test test	test.account@pwnd.at	app.test.zeitkapsl.at
John Doe	jannik.schager@pwnd.at	app.test.zeitkapsl.at
test test	jannik.schager+1@pwnd.at	app.test.zeitkapsl.at
test test	jannik.schager+2@pwnd.at	app.test.zeitkapsl.at

3.3 Audit Foundation

The following standards and guidelines were used as audit foundations:

- **ISO/IEC 27001:2022**, "Information security management systems"
- **ISO/IEC 27002:2022**, "Information technology security techniques"
- **ISO/IEC 27033-3:2010**, "Information technology - Security techniques - Network security"
- **BSI study** "Implementation Concept for Penetration Tests"
- **BSI IT-Grundschutz**
- **OWASP WSTG** Web Security Testing Guide
- **OWASP MASTG** Mobile Application Security Testing Guide

3.4 Risk Assessment according to CVSSv4

All vulnerabilities identified by PWND Labs GmbH are assessed using version 4 of the CVSS standard to determine their risk and severity levels.

The Common Vulnerability Scoring System (CVSS) provides a way to capture the principal characteristics of a vulnerability and produces a numerical score reflecting its severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes. CVSS is a published standard used by organizations worldwide.

Severity	CVSSv4 Score
Info	**0.0**
Low	**0.1 - 3.9**
Medium	**4.0 - 6.9**
High	**7.0 - 8.9**
Critical	**9.0 - 10.0**

Table 5 - CVSSv4 Score

© PWND Labs GmbH

4 Findings

The discovered vulnerabilities are sorted based on the Generic CVSS Risk Assessment and technically described. The associated risks are explained by describing the vulnerabilities, considering external factors like firewalls or WAFs.

The written risk assessment might differ slightly from the CVSS score to approximate the actual risk. The remediation section provides general recommendations for resolving the issues, which can vary based on the client's circumstances.

> ⓘ **Note**
> Zeitkapsl's cryptographic primitives are not vulnerable to known quantum attacks, so the system remains secure against a quantum adversary - assuming the transport layer (TLS) also uses post-quantum-secure encryption.

M1: Disclosure of Internal Server Errors	
Score	5.9 (Medium)
Vector string	CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N
Target	server/
References	• https://owasp.org/www-community/Improper_Error_Handling • https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html

Overview

Multiple API endpoints return raw internal error messages to users instead of generic responses, exposing details like database structures, third-party account IDs, and storage paths that could help an attacker map out the system's internals. All diagnostic output should be restricted to server-side logs only.

The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.

Status: **Resolved.**

Technical Details

At the time of the audit, the application consistently passed raw error values from upstream dependencies - Stripe, S3 (AWS), PostgreSQL (pgx), and Go's standard library - directly into HTTP error responses. This affected at least nine confirmed endpoints and 27 potential callsites identified through static analysis, several of which required no authentication. The exposed details included internal service account identifiers, database user IDs and query structures, S3 object key formats, Go type and struct names, and in one case a direct link to Stripe's internal debug dashboard.

The root cause across all instances is the same pattern - passing unsanitized error strings directly to the HTTP response writer:

```
http.Error(w, err.Error(), http.StatusInternalServerError)
```

> Some occurrences may appear in clients or be intentional, and are therefore not mentioned here.

One instance of this pattern occurs in the Stripe-facing API. In `CreateCustomerPortalSession`, errors from both the session loader and the Stripe service call are written directly to the HTTP response via `http.Error(w, err.Error(), ...)`. When the Stripe call fails, this exposes the full Stripe API error object - including the internal Stripe account ID, request ID, and a direct Stripe workbench debug URL - to the client.

```
[...]
func (c *AccountController) CreateCustomerPortalSession(w http.ResponseWriter, [...]) {
    acc, quota, err := c.LoadAccountAndQuotaFromSession(request)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    portalSession, err := c.stripeService.CreateCustomerPortalSession([...])
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    [...]
}
[...]
```

server/pkg/controller/account.go:846-865

Through static analysis using different patterns, 27 potential internal error disclosure callsites were identified across the codebase.

All matched patterns involve callsites that pass unsanitized error strings into HTTP responses, covering internal library errors, database driver messages, and third-party API responses.

File	**Lines**	**Occurrences**
`server/pkg/controller/account.go`	849, 855	2
`server/pkg/controller/base.go`	150, 177, 196, 223, 246, 269	6
`server/pkg/controller/media.go`	150, 173, 185, 191	4
`server/pkg/controller/middleware.go`	31, 61	2
`server/pkg/controller/share.go`	230, 240, 246	3
`server/pkg/controller/sse.go`	40, 65, 73	3
`server/pkg/controller/stripe.go`	39, 48, 62, 77, 93	5
`server/pkg/controller/support.go`	517, 661	2
Total		**27 findings**

> The following table lists all endpoints found to disclose internal error details.

Endpoint	Auth Required	Data Leaked
`POST /api/v1/account/create-customer-portal-session`	Session	Full Stripe API error object including internal Stripe account ID (`acct_...Y2E`), Stripe request ID, and direct Stripe workbench log URL
`POST /api/v1/account/login` (any BodyHandler endpoint)	None	Raw Go JSON decode error exposing internal struct name (`LoginRequest`), field names, and Go type names (e.g. `json: cannot unmarshal number into Go struct field LoginRequest.authToken of type string`)
`GET /api/v1/media/url?id=<id>&rep=INVALID`	Session	S3 internal object key format (`<mediaId>/INVALID`) and S3 error string (`The specified key does not exist`) revealing bucket structure
`GET /api/v1/share/media/url?shareId=<id>&accessKey=<key>&id=<id>&rep=INVALID`	None (share link only)	Same S3 key format and bucket error as #3, accessible to unauthenticated users with any valid share link
`DELETE /api/v1/share?shareId=1`	Session	Raw DB error exposing internal user ID, query parameter names, and pgx error string (`could not find share for shareID=1 userID=208...376: no rows in result set`)
`GET /api/v1/sync?since=NOTADATE`	Session	Raw Go `time.Parse` error exposing Go's internal reference time format string (`parsing time "NOTADATE" as "2006-01-02T15:04:05.999999999Z07:00": cannot parse "NOTADATE" as "2006"`)
`POST /stripe/events` (malformed body)	None	Internal JSON parse error (`⚠️ Webhook error while parsing basic request: unexpected end of JSON input`)
`POST /stripe/events` (invalid signature)	None	Internal Stripe signature verification error (`⚠️ Webhook signature verification failed: webhook had no valid signature`)

Steps to reproduce

> Since the root cause is identical across all affected endpoints, a single proof of concept is provided below to demonstrate the issue representatively.

Request with empty body:

```
POST /api/v1/account/create-customer-portal-session HTTP/2
Host: app.test.zeitkapsl.at
[...]
```

Response with internal server error:

```
HTTP/2 500 Internal Server Error
[...]

{
    "code":"parameter_invalid_empty",
    "doc_url":"https://stripe.com/docs/error-codes/parameter-invalid-empty",
    "status":400,
    "message":"You passed an empty string for 'customer'. We assume empty values
        are an attempt to unset a parameter; however 'customer' cannot be unset.
        You should remove 'customer' from your request or supply a non-empty value.",
    "param":"customer",
    "request_id":"req_...VcQ",
    "request_log_url":"https://dashboard.stripe.com/acct_...Y2E/test/
        workbench/logs?object=req_CjXuCn3D8WDVcQ",
    "type":"invalid_request_error"
}
```

Remediation Recommendation

- Replace raw error forwarding with generic client-facing error messages and route all diagnostic details to structured server-side logging.
- Differentiate between client errors (4xx) and server errors (5xx) in responses - client errors should provide just enough context for the caller to correct their request, while server errors should return only a generic message and a correlation ID that maps to the full error in internal logs.

M2: Broken Access Control in GetMediaPermission	
Score	5.3 (Medium)
Vector string	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N
Target	server/
References	-

Overview

The application permits any authenticated user to download another user's media as long as the media's Snowflake ID is known. Although the risk is mitigated by the fact that all media are end-to-end encrypted, it should not be possible to retrieve another user's content at all. Requiring a media's Snowflake ID is not a mitigation for this vulnerability, as Snowflake IDs are not meant to be secret values. Their generation algorithms are predictable.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.
>
> Status: **Resolved.**

Technical Details

At the time of the audit, the application generated pre-signed URLs for any request to retrieve media from S3 without first verifying that the requesting user owned the requested object. Consequently, any authenticated user could obtain a pre-signed URL and download the **encrypted** version of another user's media files.

While the end-to-end encryption prevents the raw content from being exposed, allowing unrestricted access to encrypted blobs is a serious security flaw. An attacker who later compromises a user's decryption key, or exploits a vulnerability in the client-side decryption process, could recover the original data. Moreover, unrestricted access to encrypted assets can be abused for reconnaissance or denial-of-service.

Affected endpoints:

Endpoint	Effect
`GET /api/v1/media/url`	Serves Single Media Access
`GET /api/v1/media/hls`	Serves HLS master playlist
`GET /api/v1/media/hls/rep`	Serves HLS rep playlist

The `GET /api/v1/media/url` endpoint accepts an `id` query parameter and returns a pre-signed URL that grants direct download access to the corresponding object in the S3 bucket. The service constructs the signature on-the-fly, using the supplied `id` to locate the media file and embed temporary credentials into the URL.

```
[...]
func (c MediaController) GetMediaUrlRedirect(w http.ResponseWriter, r *http.Request) {
    session := auth.GetSession(r)
    mediaId := snowflake.IDFromString(r.URL.Query().Get("id"))

    perm := c.mediaService.GetMediaPermission([...], mediaId, session.UserId)
    if perm == media.SharePermissionNone {
        c.Log(r).Info("media url not found", "id", mediaId)
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    [...]
}
[...]
```

server/pkg/controller/media.go:136-196

The `tx.FindMediaPermission` function executes an SQL query via the data-service to fetch the permission record associated with a specific media item.

```
[...]
func (s Service) GetMediaPermission([...], mediaId, accountId snowflake.ID) [...] {
    res, err := tx.FindMediaPermission(ctx, db.FindMediaPermissionParams{
        MediaID:    mediaId,
        CreatedFor: &accountId,
    })
    [...]
}
[...]
```

server/pkg/domain/media/share.go:136-153

The SQL statement that determines a user's permission on a media item is flawed due to its `UNION` clause.

```
[...]
SELECT permission from collection_share cs
INNER JOIN collection_media_key cmk ON cs.collection_id = cmk.collection_id
WHERE cmk.media_id = $1 AND cs.created_for = $2
UNION SELECT 1 FROM media WHERE media.account_id = $2
[...]
```

server/pkg/db/query.sql.go:2083-2086

The permission-checking query is vulnerable because it grants write access (`1`) whenever the requesting user owns *any* media item, not just the specific media referenced by the request. The missing condition allows an attacker who can upload a single media entity to obtain write permission for *all* media belonging to that user.

By tightening the condition to require ownership of the specific media item, the business-logic flaw could be eliminated.

Steps to reproduce

Sending a request using the victim's `mediaId`:

```
GET /api/v1/media/url?id=2087781556518645760&rep=orig&chunk=0
Host: app.test.zeitkapsl.at
[...]
```

Response from server with signed **s3** Redirect:

```
HTTP/1.1 302 Found
Location: https://s3.eu-central-003.backblazeb2.com/zeitkapsl-test/2087781556518645760/
chunks/orig/0000?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-
Credential=0034d2fe16328f7000000001d%2F20260310%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
Date=20260310T110435Z&X-Amz-Expires=30&X-Amz-SignedHeaders=host&X-Amz-
Signature=06d28004ba431820a8f2fe56f8a78abae39820c8b96d77a319e915890efd60a4
[...]

<a href="https://s3.eu-central-003.backblazeb2.com/zeitkapsl-test/2087781556518645760/
chunks/orig/0000?X-Amz-Algorithm=AWS4-HMAC-SHA256&amp;X-Amz-
Credential=0034d2fe16328f7000000001d%2F20260310%2Fus-east-1%2Fs3%2Faws4_request&amp;X-Amz-
Date=20260310T110435Z&amp;X-Amz-Expires=30&amp;X-Amz-SignedHeaders=host&amp;X-Amz-
Signature=06d28004ba431820a8f2fe56f8a78abae39820c8b96d77a319e915890efd60a4">Found</a>
```

Follow Redirect:

```
GET /zeitkapsl-test/2087781556518645760/chunks/orig/0000?X-Amz-Algorithm=AWS4-HMAC-
SHA256&X-Amz-Credential=0034d2fe16328f7000000001d%2F20260310%2Fus-
east-1%2Fs3%2Faws4_request&X-Amz-Date=20260310T110435Z&X-Amz-Expires=30&X-Amz-
SignedHeaders=host&X-Amz-
Signature=06d28004ba431820a8f2fe56f8a78abae39820c8b96d77a319e915890efd60a4
Host: app.test.zeitkapsl.at
[...]
```

Response from s3 with encrypted data from victim:

```
HTTP/1.1 200 OK
[...]

[BINARY CONTENT]
```

Remediation Recommendation

- Add an **explicit ownership/ACL check** before processing media-IDs.
- Harden the SQL query and ensure it returns a permission value only when the queried media is owned by the requesting user.

M3: Media-Share Corruption	
Score	5.3 (Medium)
Vector string	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:L/SC:N/SI:N/SA:N
Target	• server/ • web/
References	-

Overview

The application does not properly enforce access controls, allowing any authenticated user to add content to private collections without prior authorization. While this does not expose the collections directly, it can affect and corrupt their shared versions.

Additionally, media can be added to shared collections without sufficient validation. A crafted payload can introduce an invalid reference that breaks collection rendering, and differing status codes can be used to infer valid media IDs.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.
>
> Status: **Resolved.**

Technical Details

At the time of the audit it was observed that the media-upload endpoint allows any authenticated user to specify a *default collection* as the target without performing any ownership or permission checks. Consequently, an attacker can silently inject media objects into another user's private collection.

> **Effect on the owner's private view** - When the collection is accessed only by its owner, the malformed media entry is not displayed, so the owner sees no immediate anomaly.
>
> **Effect on shared/public views** - If the owner later shares the collection, the client attempts to render the injected media. Because the reference is invalid, the rendering pipeline fails and the entire collection view collapses, resulting in a client-side denial-of-service for anyone trying to view the collection.

The endpoint only accepts existing inventory IDs, so attackers can enumerate the catalogue by probing IDs and observing whether uploads succeed or fail.

CreateMedia sink analysis

The `POST /api/v1/media` endpoint accepts a `collectionId` field in the JSON body, which it uses to automatically assign the media to the specified default collection.

```
[...]
func (c MediaController) CreateMedia(r *Req[api.CreateMediaRequest]) (any, int, error) {
    session := auth.GetSession(r.R)
    var uploadUrls map[string]string

    m, err := db.TxR(r.Ctx, c.Db, func(tx *db.Db) (*db.Medium, error) {
        quota, err := tx.FindQuotaByAccountId(r.Ctx, session.UserId)
        [...]
        photoSizeInfo := media.NewPhotoSizeInfo(r.Body.UploadSizes)
        m, ul, err := c.mediaService.CreatePhotoMedia([...], r.Body.CollectionId, [...])
        [...]
    })
    [...]
}
[...]
```

server/pkg/controller/media.go:92-117

The `collectionId` parameter is not checked against the creator's ownership, allowing an attacker to reference and add media to any private collection belonging to another user.

```
[...]
func (s Service) CreatePhotoMedia([...], collectionId snowflake.ID, [...]) ([...]) {
    m, err := s.CreateMedia([...], collectionId, [...])
    [...]
}
[...]
func (s Service) CreateMedia([...], collectionId snowflake.ID, [...]) ([...]) {
    [...]
    m = NewMedia(id, t, cryMeta, creator, aspectRatio)
    m = &medium
    [...]
    if account.DefaultCollection != collectionId {
        _, err = tx.InsertCollectionMediaKey(ctx, db.InsertCollectionMediaKeyParams{
            MediaID:      m.ID,
            CollectionID: collectionId,
            CryKey:       cryKey,
        })
        [...]
    }
    [...]
}
[...]
```

server/pkg/domain/media/service.go:193-208

CollectionAddMedia sink analysis

The `POST /api/v1/collection/media` endpoint accepts a `mediaId` field in the JSON body, which it uses to add a media reference to the specified share. It only works if the user adding the media is authenticated.

```
[...]
func ([...]) CollectionAddMedia(r *Req[api.CollectionAddMediaRequest]) ([...]) {
    [...]
    session := auth.GetSession(r.R)
    err := db.Tx(r.Ctx, c.Db, func(tx *db.Db) error {
        for _, v := range *r.Body {
            err := c.mediaService.AddToCollection([...])
            if err != nil {
                return err
            }
        }
        return nil
    })
    [...]
    return NoContent(err)
}
[...]
```

server/pkg/controller/media.go:234-254

`tx.FindCollectionMediaKey` first looks up the collection to see whether the supplied media ID already exists. If the media ID is not found, `tx.InsertCollectionMediaKey` is invoked; it simply inserts the `mediaId` together with the associated `cry_key`. Crucially, no validation is performed to verify that the `mediaId` actually belongs to the user who is attempting the insertion.

```
[...]
func (s Service) AddToCollection([...], collectionId, mediaId, [...]) error {
    [...]
    _, err := tx.FindCollectionMediaKey(ctx, db.FindCollectionMediaKeyParams{
        CollectionID: collectionId,
        MediaID:      mediaId,
    })
    [...]
    if err != nil {
        _, err := tx.InsertCollectionMediaKey(ctx, db.InsertCollectionMediaKeyParams{
            MediaID:      mediaId,
            CollectionID: collectionId,
            CryKey:       cryKey,
        })
    [...]
}
[...]
```

server/pkg/domain/media/service.go:471-496

Steps to reproduce

PoC for CreateMedia

Sending Request with victims Collection-Id:

```
POST /api/v1/media HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "mf":2,
    "cryKey":"H74...hEf",
    "type":0,
    "cryMeta":"UUh...s+9",
    "uploadSizes":{
        "250h":13140,
        "150sq":8598,
        "1920le":55880,
        "orig":126107
    },
    "aspectRatio":2,
    "timestamp":"2026-03-22T16:48:07+01:00",
    "collectionId":"208...384"
}
```

Response when it worked:

```
HTTP/2 200 OK
[...]

{
    "i":"208...832",
    "u":"2026-03-22T15:48:16.412692+01:00",
    "t":398...,
    "a":2,
    "m":"UUh...s+9",
    "by":"208...376",
    "UploadUrls":{
        "150sq":"https://s3.eu-central-003.backblazeb2.com/...",
        "1920le":"https://s3.eu-central-003.backblazeb2.com/...",
        "250h":"https://s3.eu-central-003.backblazeb2.com/...",
        "orig":"https://s3.eu-central-003.backblazeb2.com/..."
    }
}
```

PoC for CollectionAddMedia

Sending Request with a **Media ID** that doesn't exist to **any collection we have access to**:

```
POST /api/v1/collection/media HTTP/2
Host: app.test.zeitkapsl.at
[...]

[{
    "mediaId":"2087526891556696061",
    "collectionId":"208...848",
    "cryKey":"new...23z"
}]
```

Response when the **Media ID** doesn't exist:

```
HTTP/2 500 Internal Server Error
[...]

Internal Server Error
```

Sending Request with a **Media ID** that doesn't belong to the user account to **any collection we have access to**:

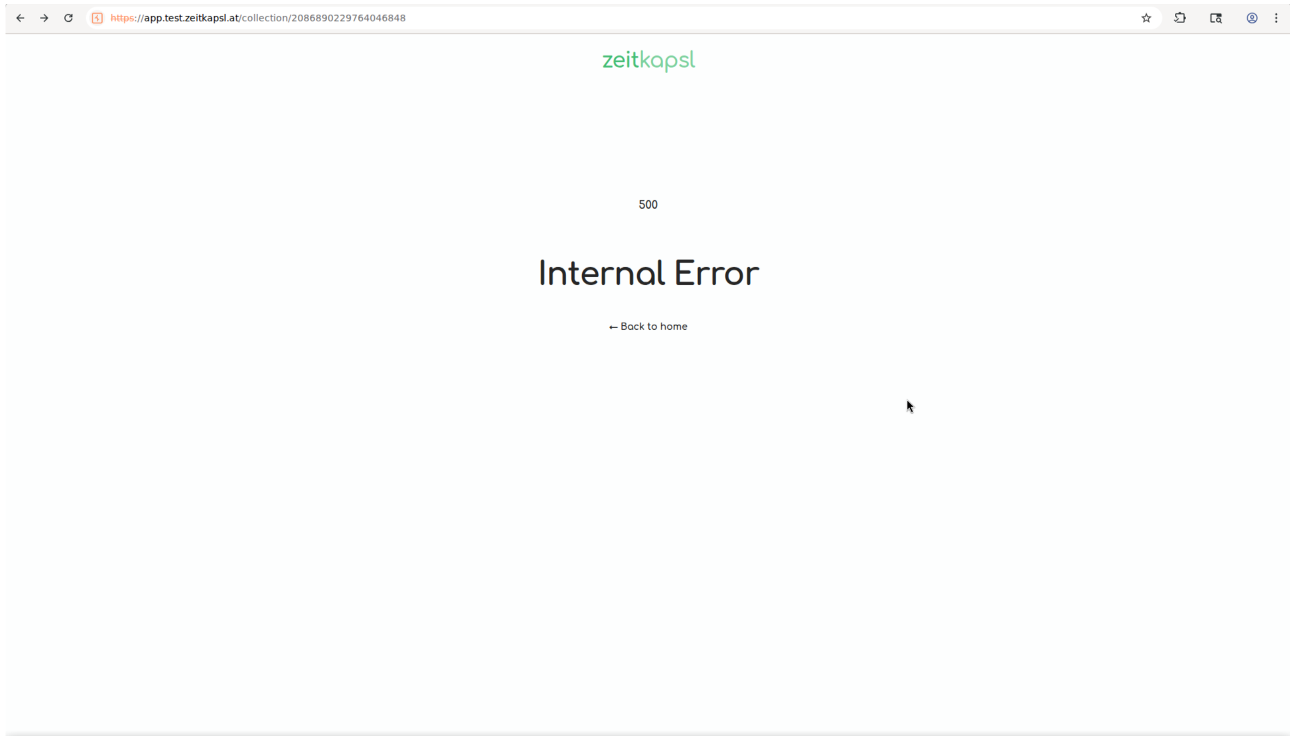
```
POST /api/v1/collection/media HTTP/2
Host: app.test.zeitkapsl.at
[...]

[{
    "mediaId":"2087526891556696064",
    "collectionId":"208...848",
    "cryKey":"new...23z"
}]
```

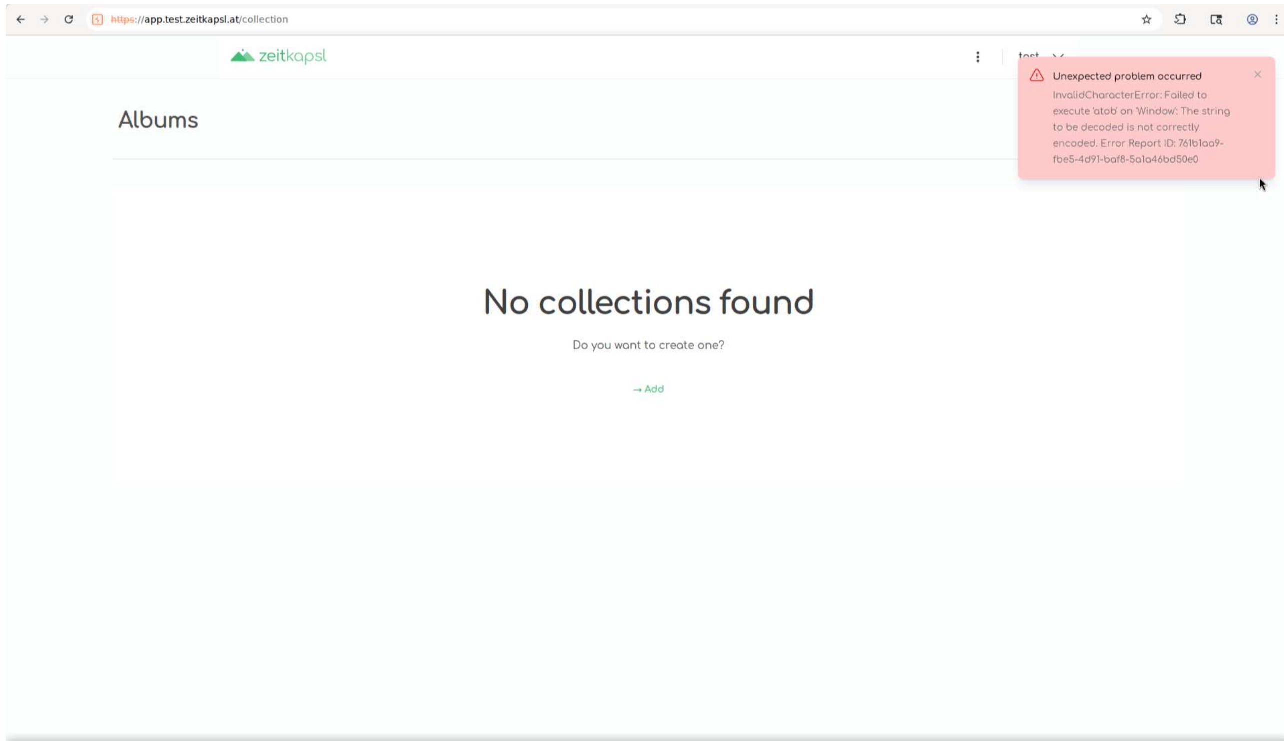
Response when the **Media ID** doesn't belong to the current user account:

```
HTTP/2 204 No Content
[...]
```

Victim's perspective of the collection after attacker uploaded faulty media:



Victim's perspective of the collection listing after attacker uploaded faulty media:



Remediation Recommendation

- Add an **explicit ownership/ACL check** before processing media-IDs or collection-IDs.
- Normalize error responses to avoid information leakage.
- Implement rate-limiting or request throttling to prevent bulk probing.
- Sanitize and validate the `collectionId` and `mediaId` payloads before any DB write.

M4: Missing Ownership Verification in Email Change Workflow	
Score	4.1 (Medium)
Vector string	CVSS:4.0/AV:P/AC:L/AT:P/PR:L/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
Target	server/
References	-

Overview

A critical flaw in the account management process allows an attacker that got access to a user's password or main key to permanently take over a user account by changing the registered email without proper verification. This results in complete and irreversible loss of access for the legitimate user, even when the user still has access to their recovery kit.

> **Note**
> This issue only applies when an attacker already has access to an active session, such as through session compromise or an unattended logged-in device, and also knows the account password required to complete the workflow.
>
> The client is aware of this issue and may implement the suggested change in the workflow in the future.
>
> Status: **Acknowledged.**

Technical Details

At the time of the audit, the email change workflow contains a critical design flaw: the confirmation link for an email change is sent to the **new email address** instead of the currently registered one. A notification about the change request is sent to the current email address, but it does not include any confirmation capability. This breaks the fundamental security assumption that ownership of the existing email must be proven before account-critical changes are finalized.

> The impact becomes especially severe if the email address is changed more than once. After a second change, the legitimate user may no longer know the currently registered email address on the account, preventing them from using recovery codes or standard account recovery flows tied to the current email.

An attacker with full account access (valid session and knowledge of the user password) can initiate an email change to an attacker-controlled address. The confirmation link is delivered exclusively to the *new* email - meaning the attacker confirms the change unilaterally without any action required from the legitimate owner.

The current owner receives only a notification with no ability to cancel or block the operation.

```
[...]
func (s *Service) RequestEmailChange([...]) error {
    [...]
    notificationToCurrentEmail := db.NewNotificationForAccount([...]{
        "newEmail":  newEmail,
        "firstName": acc.FirstName,
    })

    notificationToNewEmail := db.NewNotification([...], newEmail, [...]{
        "link": link,
        "oldEmail": acc.Email,
        "newEmail":  newEmail,
        "firstName": acc.FirstName,
    })
    [...]
}
[...]
```

server/pkg/domain/account/service.go:1174-1232

By executing this flow twice - rotating first to an intermediate address, then to a final attacker-controlled address - the attacker permanently severs the legitimate user's knowledge of the account email. This is critical because the recovery kit requires the user to specify the *current* email address during password reset; once the email is unknown to the victim, the recovery kit becomes non-functional and account recovery is impossible.

Combined with the missing session invalidation on email change (no existing sessions are terminated), the attacker retains uninterrupted access and can additionally use the active session to destroy the legitimate user's own sessions, completing a full and irrecoverable account takeover.

Steps to reproduce

Prerequisites (any one of):

- Password + deactivated 2FA
- Password + active session (2FA bypassed since already logged in)
- Password + TOTP secret (if 2FA is enabled)
- Physical device access which allows extracting the `mainKey` from SQLite and deriving the `passwordResetToken`

The attack chain consists of four steps:

Step 1: Enumerate and destroy victim's sessions.

The attacker calls `GET /api/v1/account/sessions` to list all active sessions for the account, then calls `POST /api/v1/account/logout?id=<session_id>` for each session except their own.

```
[...]
func (c *AccountController) Logout(w http.ResponseWriter, r *http.Request) {
    session := auth.GetSession(r)
    if r.URL.Query().Get("id") != "" {
        sessionId := snowflake.IDFromString(r.URL.Query().Get("id"))
        err := c.Controller.Session.DestroySessionById(sessionId, session.UserId, ...)
        [...]
    }
    [...]
}
[...]
```

server/pkg/controller/account.go:395-428

The `DestroySessionById` call scopes by `session.UserId`, meaning any session belonging to the same account can be destroyed. Since the attacker has a valid session for the account, they can kill all other sessions. The attacker wants to terminate other sessions of this user because otherwise the user can reuse one of their sessions to obtain the new email address, thus thwarting this attack.

Step 2: Request email change.

The attacker calls `POST /api/v1/account/change-email` with a new email address they control (e.g., `catchall@evil.com`).

Step 3: Confirm email change.

The confirmation link is sent to the **new** email, not the current one:

```
[...]
func (s *Service) RequestEmailChange([...]) error {
    [...]
    token, err := utils.GenerateCrockfordCode(16)
    [...]
    link := fmt.Sprintf("%s/api/v1/account/change-email/validate?token=%s", [...])
    [...]
    notificationToNewEmail := db.NewNotification([...], newEmail, [...]{
        "link": link,
        "oldEmail": acc.Email,
    })
    [...]
}
[...]
```

server/pkg/domain/account/service.go:1174-1232

The current email only receives a notification that a change was requested, but **not** the confirmation link.

Step 4: Rotate email again.

The attacker changes the email a second time to an address unknown to the user. The notification of this second change goes to `catchall@evil.com` (the attacker's address from step 3), not to the original user.

The user now has no sessions, no knowledge of the current email, and cannot use the recovery kit (which requires the current email for password reset).

Remediation Recommendation

- Send the email change confirmation link to the **current** email address, not the new one, so the legitimate account holder must approve. Alternatively, require verification from **both** current and new email address.

L1: Cache Path Traversal possible on Windows Systems	
Score	2.3 (Low)
Vector string	CVSS:4.0/AV:N/AC:L/AT:P/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N
Target	server/
References	https://owasp.org/www-community/attacks/Path_Traversal

Overview

The application's cache key normalization function does not account for all operating system path separators. If the server were deployed on a Windows system, this limitation could potentially be bypassed to access files outside the intended cache directory.

> ⓘ **Note**
> This issue is only exploitable on Windows hosts. As the application is deployed exclusively on Linux and no migration is planned, it is not exploitable in the current environment.
>
> Status: **Not Applicable.**

Technical Details

At the time of the audit, the `normalizeKey` function was found to only replace forward slashes (`/`) with underscores, without accounting for backslashes (`\`), which serve as the path separator on Windows systems.

If the server were deployed on Windows and cache keys contain user-controlled input, an attacker could potentially use backslash sequences to traverse outside the intended cache directory and read or write arbitrary files on the filesystem.

The relevant code is shown below:

```
[...]
func normalizeKey(key string) string {
    [...]
    return strings.Replace(key, "/", "_", -1)
}
[...]
func (d *FilesystemTarget) buildPath(key string) string {
    return filepath.Join(d.Path, normalizeKey(key))
}
[...]
```

server/pkg/cache/filesystem.go:50-56,62-64

The sanitized key is passed into `buildPath`, which uses `filepath.Join` to construct a filesystem path. Since `normalizeKey` does not neutralize backslashes, the resulting path could resolve outside the intended cache directory.

The table below lists all filesystem operations that consume the unsanitized path:

Function	Line	Operation
`HasKey`	67	`os.Stat(d.buildPath(key))`
`GetKey`	72	`os.Stat(d.buildPath(key))`
`GetKey`	76	`os.Open(d.buildPath(key))`
`PutKey`	84	`os.Create(d.buildPath(key))`
`DeleteKey`	93	`os.Stat(d.buildPath(key))`
`DeleteKey`	97	`os.Remove(d.buildPath(key))`

Remediation Recommendation

- Replace the custom `normalizeKey` implementation with a well-established path sanitization function such as `filepath.Clean` combined with a base-path containment check, rather than relying on manual character replacement.
- Use `os.PathSeparator` instead of hardcoding `/` to ensure consistent behavior across operating systems.
- Validate the resolved path after `filepath.Join` to ensure it remains within the intended cache directory before performing any filesystem operation.
- Consider using an allowlist-based approach that only permits alphanumeric characters and a limited set of safe symbols in cache keys, rejecting any input that does not conform.

L2: Leaking of Encrypted Media Metadata	
Score	2.3 (Low)
Vector string	CVSS:4.0/AV:N/AC:L/AT:P/PR:L/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N
Target	server/
References	-

Overview

The application lets any logged-in user retrieve the encrypted metadata of a media item simply by supplying its Snowflake ID, without verifying that the requester owns the item.

While the metadata itself remains encrypted and cannot be read directly, the exposure still confirms the existence of specific media items, leaks server-side timestamps, and could enable enumeration of other users' assets - potentially aiding targeted attacks or violating privacy expectations.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.
>
> Status: **Resolved.**

Technical Details

At the time of the audit, two endpoints allowed access to another user's media metadata without ownership verification.

The `UpdateMedia` controller accepts any valid media Snowflake ID without checking whether the caller owns the item. By supplying null values for the update fields, the write is skipped entirely, but the endpoint still returns the full media record - including unencrypted fields such as timestamps, owner ID, and private flags.

Similarly, the ad-hoc share creation endpoint accepts arbitrary media IDs without verifying ownership. An attacker can create a share containing a victim's media, then retrieve it to obtain metadata and pre-signed download URLs for all image representations, including the original - accessible without session authentication.

> The comments in the code suggest that the developers were already partially aware of the issue regarding `UpdateMedia`.

1. UpdateMedia

The `UpdateMedia` controller does not verify that the caller owns the supplied `mediaId`. Although the update operation fails, the response discloses the referenced media's encrypted content to any authenticated caller.

```
[...]
func (c MediaController) UpdateMedia(r *Req[api.UpdateMetaDataRequest]) ([...]) {
    session := auth.GetSession(r.R)
    m, err := db.TxR[api.Media](r.Ctx, c.Db, func(tx *db.Db) (*api.Media, error) {
        var err error
        [...]
        err = c.mediaService.UpdateMetadata([...])
        [...]
        err = c.mediaService.UpdateMediaFlags([...])
        [...]
        m, err := tx.FindMediaById(r.Ctx, r.Body.MediaId)
        [...]
        mediaDto := MapMediaDto(&m)
        return mediaDto, err
    })
    [...]
}
[...]
```

server/pkg/controller/media.go:280-312

The core problem is that the higher-level functions can be bypassed by passing a **null** value. When a null is supplied, the lower-level update logic detects that there is nothing to modify, skips the database UPDATE, and returns without error - effectively allowing the operation to be ignored.

The response contained a dictionary with the following data:

Key	Field	Value	Meaning
`i`	`Id`	`2087526891556696064`	Media snowflake ID
`u`	`UpdatedAt`	`2026-…+01:00`	Last updated timestamp
`s`	`Status`	`1`	`MediaStateUploaded`
`v`	`Version`	`8`	Optimistic-lock version
`t`	`Timestamp`	`3981731520`	Photo capture time or Upload time (fallback)
`a`	`AspectRatio`	`1.0415913`	≈1:1, slightly portrait
`m`	`CryMeta`	`QnzwKk…P0E=`	Encrypted metadata (filename, EXIF, GPS)
`k`	`CryKey`	`2ta4ug…DAw=`	Wrapped media key; non-null means it's in the owner's default collection
`by`	`CreatedBy`	`2087526854236827648`	Owner's account snowflake ID
`li`	`Liked`	`true`	Owner's private "liked" flag

2. CreateAdHocShare

`CreateAdHocShare` creates a new collection and then calls `AddToCollection` for each attacker-supplied `mediaId`. The ownership check in `AddToCollection` only verifies write access to the *collection* (which the attacker owns), never whether the attacker owns the *media*. Any valid media ID - belonging to any user - can be injected.

As a result, an authenticated attacker can leak any victim's encrypted metadata, aspect ratios, and timestamps for any media ID they can guess or enumerate.

```
[...]
func (c ShareController) CreateAdHocShare(r *Req[api.CreateAdHocShareRequest]) ([...]) {
    session := auth.GetSession(r.R)
    var coll *db.Collection
    shareId, err := db.TxR[snowflake.ID](r.Ctx, c.Db, func(tx *db.Db) ([...]) {
        var err error
        coll, err = c.mediaService.CreateCollection([...])
        [...]
        for _, m := range r.Body.Medias {
            err = c.mediaService.AddToCollection([...], m.MediaId, [...])
            [...]
        }
        return c.mediaService.CreateShare([...])
    })
    [...]
}
[...]
```

server/pkg/controller/share.go:76-99

The `GetShare` controller retrieves all media in the collection purely by `collection_id` and upload state, with no ownership validation - since shares were never designed to contain media belonging to other users, no such check was ever implemented. As a result, the injected media is treated as legitimate share content and its encrypted metadata is returned unconditionally.

```
[...]
func (c ShareController) GetShare(_ http.ResponseWriter, r *http.Request) ([...]) {
    share, status, err := c.VerifyShareAccessFromRequest(r)
    if err != nil {
        return nil, status, err
    }
    [...]
    medias, err := c.Db.FindMediaByCollectionIdAndStatus(r.Context(), [...]{
        CollectionID: *share.CollectionID,
        State:        int16(media.MediaStateUploaded),
    })
    [...]
}
[...]
```

server/pkg/controller/share.go:163-189

Steps to reproduce

1. PoC - UpdateMedia

Request to the server with known `mediaId`:

```
PATCH /api/v1/media HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "mediaId": "2087526891556696064",
    "cryMeta": null,
    "liked": null,
    "hidden": null,
    "version": 0
}
```

Answer from the server with media's encrypted metadata and some other details:

```
HTTP/2 200 OK
[...]

{
    "i":"208...064",
    "u":"202...+01:00",
    "s":1,
    "v":8,
    "t":398...,
    "a":1.0415913,
        "m":"Qnz...P0E=",
        "k":"2ta...Srt",
        "by":"208...648",
        "li":true
}
```

2. PoC - AdHoc-Share

First Request for share creation with victim mediaId

```
POST /api/v1/share/adhoc HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
  "cryKey":"GIV.../JO",
  "accessKey": "AAEC",
  "cryPassword": "GIVS.../JO",
  "permission": 0,
  "validUntil": null,
  "collection": {
    "cryKey": "GIV.../JO",
    "cryName": "GIV.../JO",
    "coverMediaId":"208...440"
  },
  "medias": [
    {
      "mediaId":"208...440",
      "cryKey":"GIV.../JO"
    }
  ]
}
```

Response:

```
HTTP/2 200 OK
[...]

{
    "id":"208...056",
    "collectionId":"208...344",
    "link":"https://app.test.zeitkapsl.at/s/208...056"
}
```

Get the AdHoc-Share

```
GET /api/v1/share?shareId=208...056&accessKey=AAEC HTTP/2
Host: app.test.zeitkapsl.at
[...]
```

Response with encrypted meta-data:

```
HTTP/2 200 OK
[...]

{
    "id":"208...056",
    "createdBy":"208...648",
    "permission":0,
    "cryKey":"GIV.../JO",
    "cryName":"GIV.../JO",
    "cryPassword":"GIV.../JO",
    "medias":[{
        "id":"208...440",
        "cryMeta":"s1y...KSw==",
        "cryKey":"GIV...o/JO",
        "type":0,
        "timestamp":398...,
        "aspectRatio":1.7438692,
        "urls":{
            "150sq":"https://app.test.zeitkapsl.at/api/v1/share/media/url?id=...",
            "1920le":"https://app.test.zeitkapsl.at/api/v1/share/media/url?id=...",
            "250h":"https://app.test.zeitkapsl.at/api/v1/share/media/url?id=...",
            "orig":"https://app.test.zeitkapsl.at/api/v1/share/media/url?id=..."
        }
    }],
    "coverId":"2088262019552317440",
    "collectionId":"2088286514921017344"
}
```

Remediation Recommendation

- Add an **explicit ownership/ACL check** before processing media-IDs.
- **Validate Input Early** - Add a guard clause in the public-facing controller (and any wrapper functions) that rejects a request when `mediaId` or all the required update fields are `null` or empty. Return a clear 400 Bad Request instead of silently ignoring the call.
- **Enforce Non-Nullable Columns at the DB Level** - Define the columns that must be updated as `NOT NULL` (or add a check constraint) so that an UPDATE with a null value triggers a database error, which can then be translated into a proper API error response.
- **Fail-Fast on No-Op Updates** - After the validation step, verify that at least one mutable field has a non-null value. If nothing would be changed, return a 422 Unprocessable Entity (or similar) rather than proceeding to the lower-level function.
- **Consistent Error Propagation & Logging** - Ensure that any validation or database error bubbles up to the controller and is logged with context (user ID, media ID, payload). This makes it easier to detect misuse and prevents silent bypasses.

L3: Go core persists mainKey in local SQLite unnecessarily	
Score	1.0 (Low)
Vector string	CVSS:4.0/AV:P/AC:L/AT:P/PR:H/UI:A/VC:N/VI:N/VA:L/SC:N/SI:N/SA:N
Target	core/
References	https://owasp.org/www-community/controls/Least_Privilege_Principle

Overview

The Go core library persists the main key after login. This key is never read back by any code, making it a dead write that needlessly increases the attack surface. An attacker with physical device access can extract this key and can perform a complete account takeover.

> **Note**
> The client is aware of this issue and has chosen to accept it under the current design, as it does not pose an immediate threat and is a deliberate trade-off to allow account recovery if users lose both their recovery kit and password.
>
> Status: **Risk Accepted.**

Technical Details

At the time of the audit, the Go core library (used by Android, iOS, desktop, and CLI clients) persists the root encryption key (`mainKey`) as a base64 string in local SQLite after login.

At login, `SetAccount` writes all derived keys to SQLite, including `mainKey`:

```
[...]
func (r *Repo) SetAccount([...]) error {
    [...]
    return Tx(r.q, func(tx *Db) error {
        [...]
        if err := r.SetSettingTx(ctx, tx, SettingMainKey, api.EncodeBinary(mainKey));
        err != nil {
            return err
        }
        [...]
    })
}
[...]
```

core/pkg/db/repo.go:427-490

`GetMainKey()` (repo.go:290-300) exists but has zero callers anywhere in the codebase, which means storing of the `mainKey` is not necessary. In case an attacker gains physical access to the device, the attacker can extract the key from the database.

With `mainKey`, the attacker can derive the password reset token:

```
resetToken = HKDF(mainKey, "passwordResetToken", 32)
```

The `resetToken` allows the attacker to call the unauthenticated `POST /api/v1/account/reset-password` endpoint to set a new password before taking over the account as described in M4: Missing Ownership Verification in Email Change Workflow.

The web client handles this only slightly better: Instead of storing the main key, it stores the encryption key derived by the user email and password (in `web/src/lib/account.js:186`). However, as this stored encryption key is used to create the recovery kit (only possible on the web client), storing the encryption key could be a deliberate design choice: In case the user forgets their password but retain an active web session, they can download the recovery kit and thereby reset their password.

Steps to reproduce

1. Obtain physical access to a device running a Go-core client (Android, iOS, desktop, or CLI).
2. Extract the SQLite database from the app's data directory.
3. Query: `SELECT value FROM settings WHERE key = 'mainKey'`.
4. Base64-decode the value to obtain the raw 32-byte `mainKey`.
5. Derive: `resetToken = HKDF-SHA512(mainKey, "passwordResetToken", 32)`.
6. Call `POST /api/v1/account/reset-password` with the derived `resetToken`, a new `authToken`, and a new `wrappedMainKey`.
7. The account is now under attacker control, even if the user has since changed their password. The attacker can now rotate the email twice to lock out the user indefinitely, as reported in finding M4: Missing Ownership Verification in Email Change Workflow.

Remediation Recommendation

- In accordance with the concept of Least Priviliges, remove the `mainKey` write at `repo.go:473` and the unused `GetMainKey()` function. The `indexKey` is sufficient to decrypt collections.
- Run `VACUUM` or enable `PRAGMA secure_delete` on the SQLite database to erase previously written sensitive data from disk on logout.

L4: Missing AAD in AES-GCM allows ciphertext role confusion and swap attacks	
Score	1.0 (Low)
Vector string	CVSS:4.0/AV:L/AC:H/AT:P/PR:H/UI:A/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N
Target	-
References	-

Overview

All AES-256-GCM encryption uses nil AAD (Additional Authenticated Data). When the same key encrypts multiple values with different semantic roles, ciphertexts are interchangeable. A malicious server can swap the encrypted collection name with the encrypted media key (both under `collectionKey`), and if the client displays the "name" on a semi-confidential channel (notification, lock screen), an attacker observing the screen learns the media key and can decrypt the uploaded file. This attack is illustrated in two ProVerif models.

> **Note**
> The client is aware that no concrete attack scenario was identified at the time of the audit. However, the issue still exists and may become relevant in the future if related workflow or implementation changes are introduced.
>
> Status: **Acknowledged.**

Technical Details

Both `cryName` and `cryMediaKey` are encrypted under `collectionKey` using the same `Encrypt` function with nil AAD.

```
encryptedName, err := Encrypt([]byte(name), collectionKey)
```

server/pkg/crypto/crypto.go:346 - collection name encryption

```
cryKey, err := crypto.Encrypt(mediaKey, collectionKey)
```

core/pkg/sync/upload_photo.go:125 - media key wrapping

The `Encrypt` function passes nil AAD:

```
result := aesgcm.Seal(nil, iv, data, nil)
```

server/pkg/crypto/crypto.go:136

Without AAD, `sdec(senc(mediaKey, collectionKey), collectionKey)` succeeds and returns `mediaKey` bytes, which the client interprets as the collection name. If displayed, the raw key bytes are exposed. The ProVerif model `zeitkapsl_aad.pv` confirms: `not attacker(file_content)` is **false** (which means that the attacker can learn it), while `not attacker(collectionKey)` is **true** (collectionKey does not leak).

The same problem occurs when decrypting the media and the media metadata:

```
[...]
func (a ZeitkapslGcmBased) DecryptMedia([...]) (*Media, error) {
    [...]
    data, err := Decrypt(media.CryPayload, mediaKey)
    if err != nil {
        return nil, err
    }

    metaData, err := Decrypt(media.CryMetadata, mediaKey)
    if err != nil {
        return nil, err
    }
    [...]
}
[...]
```

server/pkg/crypto/crypto.go:445-467

A malicious server can swap media contents with the metadata, which would not be caught by the DecryptMedia function. This is illustrated in the ProVerif model `zeitkapsl_integrity.pv` which attempts (but fails) to prove the following statement: "Whenever the client succesfully decrypts, the client can be sure that the data it received is the same as what was uploaded."

Steps to reproduce

This attack illustrates a potential ciphertext confusion attack between collection name and media key. It requires a malicious server and a shoulder-surfing scenario:

1. A malicious server stores the legitimate `cryName` and `cryMediaKey` for a collection.
2. When the client requests the collection, the server returns `cryMediaKey` in place of `cryName`.
3. The client decrypts `cryMediaKey` with `collectionKey`, obtains the raw `mediaKey` bytes, and displays them as the collection "name" (e.g., in a notification or UI title).
4. An attacker observing the screen (shoulder surfing, screenshot, lock screen notification) reads the displayed bytes.
5. The attacker uses the `mediaKey` to decrypt `senc(file_content, mediaKey)` from the stored media blobs.

Remediation Recommendation

- Use the AAD parameter in AES-GCM to bind each ciphertext to its semantic role (e.g., `"collectionName"`, `"mediaKey"`, `"metadata"`, `"payload"`).
- Alternatively, derive one-time sub-keys for each role within the same scope (e.g., `nameKey = HKDF(collectionKey, "name")`, `mediaWrapKey = HKDF(collectionKey, "mediaKey")`).

I1: Analysis of flows using formal methods	
Score	0.0 (Info)
Vector string	CVSS:4.0/AV:P/AC:H/AT:P/PR:H/UI:A/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N
Target	-
References	https://bblanche.gitlabpages.inria.fr/proverif/

Overview

This section provides an overview of how formal verification methods were used to analyze the end-to-end encryption flows. The passing ProVerif queries underline the fact that no major issues were found in this audit that would contradict the main design goals stated (informally) in the README.md.

Minor findings illustrated as ProVerif models relate to the absence of Additional Authenticated Data (AAD) in AES-GCM, which is discussed in L4: Missing AAD in AES-GCM allows ciphertext role confusion and swap attacks.

> ⓘ **Note**
> Both ProVerif-illustrated findings stem from the same root cause (nil AAD in AES-GCM) and neither enables a malicious server to recover plaintext user data without an additional side channel.
>
> The client is aware of the issue and may address it in a future release.
>
> Status: **Acknowledged.**

Technical Details

The following pages contain the **Protocol Diagrams** and **Formal Verification Models**.

1. Key Hierarchy

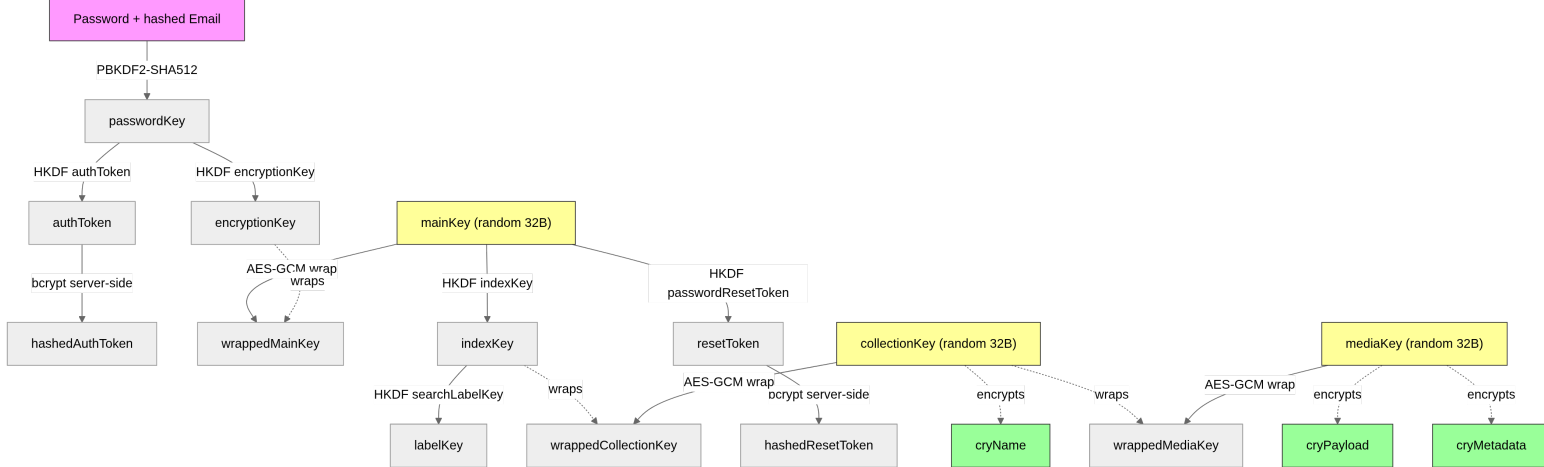


Figure 2 - Key Hierarchy

2. Upload Protocol (zeitkapsl_upload.pv)

Threat model: Honest-but-curious server.

Property verified: Server cannot learn the plaintext media file.

Client | Server (HBC)

Generate mainKey (random)

salt = SHA512(email)

passwordKey = PBKDF2(password, salt)

encryptionKey = HKDF(passwordKey, "encryptionKey")

authToken = HKDF(passwordKey, "authToken")

indexKey = HKDF(mainKey, "indexKey")

wrappedMainKey = AES-GCM(mainKey, encryptionKey)

Registration

email, authToken, wrappedMainKey, resetToken

Stores bcrypt(authToken), wrappedMainKey, bcrypt(resetToken)

Generate collectionKey (random)

cryName = AES-GCM(name, collectionKey)

cryCollKey = AES-GCM(collectionKey, indexKey)

Collection Creation

cryName, cryCollKey

Generate mediaKey (random)

cryPayload = AES-GCM(file, mediaKey)

cryMetadata = AES-GCM(metadata, mediaKey)

cryMediaKey = AES-GCM(mediaKey, collectionKey)

Media Upload

cryPayload, cryMetadata, cryMediaKey

Server has only encrypted blobs. Cannot derive any key without password.

Client | Server (HBC)

Figure 3 - Upload Protocol

ProVerif result: `not attacker(file_content)` is **true**, which means that the server cannot obtain the file contents. Since `event(FileUploaded())` is **true**, the client process can work as intended.

3. Integrity Model (zeitkapsl_integrity.pv)

Threat model: Fully malicious server (Dolev-Yao attacker). **Property verified:** If decryption succeeds, the recovered content matches what was originally uploaded.

Client
Attacker (Malicious Server)
Upload Phase
email, authToken, wrappedMainKey, resetToken
cryName, cryCollKey
cryPayload, cryMediaKey
event FileUploaded(file_content)
Download Phase
Attacker chooses what to deliver. May replay, modify, or fabricate.
resp_cryMediaKey
kdec(resp_cryMediaKey, collectionKey) -- fails if tampered (GCM tag)
resp_cryPayload
sdec(resp_cryPayload, mediaKey) -- fails if tampered (GCM tag)
event ClientDecrypted(content)
RESULT: ClientDecrypted(x) ==> FileUploaded(x) is FALSE
(server can swap cryMetadata for cryPayload -- same key, no AAD)
Client
Attacker (Malicious Server)

Figure 4 - Integrity Discussion

ProVerif result: Correspondence **fails** - the server can substitute `cryMetadata` for `cryPayload` (both under the same `mediaKey` without AAD). The client recovers `metadata` instead of `file_content`.

---

4. Password Change (zeitkapsl_password_change.pv)

Threat model: Fully malicious server (DY attacker). **Properties verified:** Both passwords, mainKey, and file_content remain secret. Integrity holds. Rollback of wrappedMainKey1 is detected.

Client | Attacker (DY)

Phase 1: Registration (password1)

passwordKey1 = PBKDF2(password1, salt)

encKey1 = HKDF(passwordKey1, "encryptionKey")

wrappedMainKey1 = AES-GCM(mainKey, encKey1)

Client → Attacker (DY): email, authToken1, wrappedMainKey1, resetToken

Phase 2: Upload

Client → Attacker (DY): cryName, cryCollKey

Client → Attacker (DY): cryPayload, cryMediaKey

Phase 3: Password Change

passwordKey2 = PBKDF2(password2, salt)

encKey2 = HKDF(passwordKey2, "encryptionKey")

wrappedMainKey2 = AES-GCM(mainKey, encKey2)

mainKey unchanged, only re-wrapped

Client → Attacker (DY): authToken1, authToken2, wrappedMainKey2

Phase 4: Download (with password2)

Re-derives encKey2 from password2

Attacker (DY) → Client: resp_wrappedMainKey

kdec(resp, encKey2) -> mainKey. Old wrappedMainKey1 fails: encKey1 != encKey2

Attacker (DY) → Client: resp_cryCollKey

Attacker (DY) → Client: resp_cryMediaKey, resp_cryPayload

Full decryption chain succeeds

RESULT: password1, password2, mainKey, file_content all secret. Integrity holds.

Client | Attacker (DY)

Figure 5 - Password Change

5. Server Compromise + Old Password Leak (zeitkapsl_server_compromise.pv)

Threat model: Full DB breach after password change + independent leak of old password.

Properties verified: New password, mainKey, and file_content remain secret despite full DB + old password.

Client
Server
Attacker
Phases 1-3: Over TLS (private)
Registration (wrappedMainKey1)
Upload (cryPayload, cryMediaKey)
Password Change (wrappedMainKey2)
Replaces wrappedMainKey1 with wrappedMainKey2
Server Compromise
password1 (leaked externally)
DB dump: wrappedMainKey2, cryCollKey, cryMediaKey, cryPayload
Route A: password1 -> encKey1 -> need wrappedMainKey1 (DELETED)
Route B: wrappedMainKey2 -> need encKey2 -> need password2 (UNKNOWN)
Both routes blocked
Phase 4: Download from compromised server
wrappedMainKey2, cryCollKey, cryMediaKey, cryPayload
Decrypts with encKey2 (from password2)
Full chain succeeds
RESULT: file_content, mainKey, password2 all secret. Forward secrecy holds.
Client
Server
Attacker

Figure 6 - Old Password Leak

6. AAD Attack (zeitkapsl_aad.pv)

Threat model: Malicious server + shoulder surfing.

Property verified (negative): Missing AAD allows ciphertext swap, leaking file_content.

Client
Malicious Server
Shoulder Surfer
Upload (over TLS)
Registration data
cryName = AES-GCM(name, collectionKey)
cryMediaKey = AES-GCM(mediaKey, collectionKey)
cryPayload = AES-GCM(file, mediaKey)
cryName and cryMediaKey both under collectionKey, no AAD
Attack: Swap cryName with cryMediaKey
cryCollKey (genuine)
Decrypts -> collectionKey (OK)
cryMediaKey (in place of cryName!)
Decrypts with collectionKey -> gets mediaKey bytes as "name"
Displays "name" on screen
mediaKey bytes shown on screen
Reads mediaKey from screen
Decrypts cryPayload with mediaKey
file_content leaked!
RESULT: attacker(file_content) = CAN learn. attacker(collectionKey) = CANNOT learn.
Client
Malicious Server
Shoulder Surfer

Figure 7 - Ciphertext

7. Phase Corruption / Temporal Attacker (zeitkapsl_phase_corruption.pv)

Threat model: Phased temporal attack. Old password leaked after rotation. Server enforces auth. **Properties verified:** file_content and mainKey remain secret despite old password leak.

Client | Server | Attacker

Phase 0: Registration + Upload (TLS)

salt = SHA512(email)

Keys from password1 + salt

bcrypt(authToken1), wrappedMainKey1, cryCollKey, cryMediaKey, cryPayload

Phase 1: Password Rotation (TLS)

Keys from password2 + salt

wrappedMainKey2 = AES-GCM(mainKey, encKey2)

wrappedMainKey2, bcrypt(authToken2), authToken2

Replaces stored auth + wrappedMainKey

Phase 1: Download

authToken2 (on public channel)

Verifies bcrypt(authToken2) = stored hash (OK)

wrappedMainKey2, cryCollKey, cryMediaKey, cryPayload

Decrypts full chain (OK)

Phase 2: Old Password Leaked

Learns password1 + email (public)

Computes authToken1 from password1

Presents authToken1 to server

bcrypt(authToken1) != stored hash -- REJECTED

Cannot access wrappedMainKey2 (needs password2)

wrappedMainKey1 never on public channel

RESULT: file_content and mainKey remain secret.

Client | Server | Attacker

Figure 8 - Phase Corruption

Modelling Choices

- **AES-256-GCM** is modelled as a perfect authenticated cipher with separate constructors for data encryption (`senc` / `sdec`) and key wrapping (`kenc` / `kdec`), except in the AAD model which uses a single `senc` to reflect the real code. AES-GCM is modelled as deterministic (`senc(m, k)` with no explicit nonce). This is sound for the properties verified (secrecy, integrity, reachability). Randomized encryption would only be needed for observational equivalence properties, which are not queried.
- **HKDF** is modelled as an uninterpreted function - different labels produce unrelated outputs; no cross-label derivation is possible.
- **PBKDF2** and **bcrypt** are modelled as an uninterpreted one-way function (no inverses).
- **cryMetadata is intentionally omitted** from most models because the real code encrypts both payload and metadata under the same mediaKey without AAD. This allows a same-key substitution attack demonstrated in models 5 and 6.

Steps to reproduce

The models are provided to the client in a private GitHub repository.

Using ProVerif, the queries can be evaluated to verify the above claims.

Remediation Recommendation

While the main design goal of the system is clear (in layman's terms "no password, no access to private data"), consider stating further assumptions, threat models and intended security properties of the system. Some examples are listed here:

- What should happen if a user's main key ever gets leaked? Should it be possible to recover the account by rotating the main key in this case, requiring re-wrapping of downstream keys? At the moment, this is not implemented, potentially causing problems for real-world users - and it is not clear if this is out of scope for the product or a deviation from the specification.
- What should happen if a user removes another user from a collection? Should it be possible to rotate the collection key in order to lock out the user with revoked access permanently? Similar to the previous example, this is currently not implemented.
- What are the trust assumptions on the server? What should happen if the server starts deleting, duplicating, reordering or adding previously deleted media objects in a collection? Should the client be able to detect tampering not just on the media data itself, but also on collection structure? This is not possible at the time of the audit, and it is not clear what the assumptions on the server are in this regard.

I2: Email Enumeration Oracles	
Score	0.0 (Info)
Vector string	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:N/VI:N/VA:N/SC:N/SI:N/SA:N
Target	server/
References	-

Overview

Several unauthenticated API endpoints expose whether a given email address is registered in the system - through differences in response status codes, error messages, and timing behavior. This enables an attacker to silently compile a list of valid user accounts, which can then be leveraged for targeted credential-stuffing, phishing, or social engineering attacks.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.
>
> This issue was rated as informational, as confirming the existence of an account associated with a specific email address through enumeration or brute-force techniques represents only a limited confidentiality impact and does not directly expose sensitive account data.
>
> Status: **Resolved.**

Technical Details

At the time of the audit, multiple unauthenticated API endpoints were found to expose whether a given email address is associated with a registered account. Three distinct enumeration vectors were identified across the support ticket, login, and password reset endpoints - leaking account existence through response body content, timing differentials, differing error messages, and distinct HTTP status codes.

Combined, these vectors allow an attacker to silently enumerate valid user accounts at scale, enabling follow-up attacks such as credential stuffing, targeted phishing, or further exploitation of account-specific functionality.

1. Ticket Support AccountId oracle

The ticket creation endpoint accepts an email address in the request body and attempts to resolve it to an existing account. If a matching account is found, the corresponding `account_id` is included in the response. This not only confirms the existence of the account but also exposes its internal identifier, which could be leveraged for further targeted attacks.

```
[...]
func (c *SupportController) CreatePublicTicket(r *Req[...]) ([...]) {
    emailAddr, err := mail.ParseAddress(r.Body.Email)
    [...]
    email := strings.ToLower(strings.TrimSpace(emailAddr.Address))
    [...]
    ticket, err := c.supportService.CreateTicket(r.Ctx, subject, email, accountID)
    [...]
    response := TicketResponse{
        [...]
        AccountID:      ticket.AccountID,
        [...]
    }

    return Ok(response, nil)
}
[...]
```

server/pkg/controller/support.go:765-877

> The `account_id` serves no functional purpose in the response for the ticket creator and should be omitted.
>
> The server already logs the association internally (`"has_account", accountID != nil`), which is sufficient for administrative purposes.

2. Login response and timing oracle

When the email is unknown, the server rejects the request immediately.

When a matching account exists, it proceeds to verify the password - introducing a measurable delay:

```
[...]
func (c *AccountController) Login(request *Req[api.LoginRequest]) (any, error) {
    acc, err := account.FindAccountByEmail(request.Ctx, c.Db, request.Body.Email)
    if err != nil {
        [...]
        return nil, domain.ErrUnauthorized("invalid username or password")
    }

    err = acc.Login(request.Body.AuthToken)
    if err != nil {
        [...]
        return nil, domain.ErrUnauthorized("Unauthorized")
    }
    [...]
}
[...]
```

server/pkg/controller/account.go:335-350

The `acc.Login` method internally calls `bcrypt.CompareHashAndPassword`, which is intentionally computationally expensive and adds ~130 ms to the response time:

```
[...]
func (account *Account) Login(authToken string) error {
    [...]
    err := bcrypt.CompareHashAndPassword(account.AuthTokenHash, []byte(authToken))
    [...]
}
[...]
```

server/pkg/db/account.go:114-124

Additionally, the error message returned allows to distinguish between `invalid username or password` and plain `Unauthorized`.

3. Password response and timing oracle

The password reset endpoint returns different HTTP status codes depending on whether an account exists for the provided email - a `404 Not Found` for unknown emails and a `403 Forbidden` when the account exists but the reset token is invalid:

```
[...]
func (c *AccountController) ResetPassword(r *Req[api.ResetPasswordRequest]) ([...]) {
    [...]
    acc, err := c.Db.FindActiveAccountByEmail(r.Ctx, r.Body.Email)
    if err != nil {
        return nil, domain.ErrNotFound([...])
    }

    err = acc.ResetPassword(r.Body.PasswordResetToken, r.Body.NewAuthToken, [...])
    [...]
}
[...]
```

server/pkg/controller/account.go:479-485

Additionally, when an account is found, `bcrypt.CompareHashAndPassword` is called to verify the reset token - introducing the same timing side-channel as the login endpoint. Since both the status code difference and the timing differential stem from the same early account lookup, they can be addressed in a single fix.

```
[...]
func (account *Account) ResetPassword(passwordResetToken, newAuthToken [...]) error {
    err := bcrypt.CompareHashAndPassword(account.ResetTokenHash, [...])
    if err != nil {
        return domain.ErrForbidden("invalid password reset token supplied")
    }
    [...]
}
[...]
```

server/pkg/db/account.go:142-158

Steps to reproduce

1. PoC - Ticket Support oracle

Request with an email address that has no associated account:

```
POST /api/v1/public/support/ticket HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email":"test.test@test.test",
    "subject":"test",
    "message":"test",
    "nonce":null,
    "solution":null,
    "difficulty":1,
    "expires_at":1
}
```

Response from server **without** an `account_id`:

```
HTTP/2 200 OK
[...]

{
    "id":"208...288",
    "ticket_ref":"ZK-...",
    "subject":"test",
    "from_email":"test.test@test.test",
    "state":0,
    "created_at":"202...51Z",
    "updated_at":"202...51Z",
    "last_message_at":"202...51Z",
    "message_count":1,
    "has_unresponded":false
}
```

Request with a registered email address:

```
POST /api/v1/public/support/ticket HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email":"test.account@pwnd.at",
    "subject":"test",
    "message":"test",
    "nonce":null,
    "solution":null,
    "difficulty":1,
    "expires_at":1
}
```

Response from server **with** an `account_id`:

```
HTTP/2 200 OK
[...]

{
    "id":"208...480",
    "ticket_ref":"ZK-...",
    "subject":"test",
    "from_email":"test.account@pwnd.at",
    "state":0,
    "created_at":"202...41Z",
    "updated_at":"202...41Z",
    "last_message_at":"202...41Z",
    "account_id":"208...448",
    "message_count":1,
    "has_unresponded":false
}
```

The response includes the account ID of the associated user, if one exists for the given email address.

2. PoC - Login oracle

Request with an email address that has no associated account:

```
POST /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email": "test.test@test.test",
    "authToken": "dummy"
}
```

The server responds after **39 ms** with:

```
HTTP/2 401 Unauthorized
[...]

invalid username or password
```

Request with a registered email address:

```
POST /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email": "test.account@pwnd.at",
    "authToken": "dummy"
}
```

The server responds after **186 ms** with:

```
HTTP/2 401 Unauthorized
[...]

Unauthorized
```

Request Type	Avg. Response Time (10 requests)
Unregistered email	~49.6 ms
Registered email	~178.2 ms

3. PoC - Password reset oracle

Request with an email address that has no associated account:

```
POST /api/v1/account/reset-password HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email": "test.test@test.test",
    "passwordResetToken": "test",
    "newAuthToken":[...],
    "newWrappedMainKey":[...]
}
```

The server responds with a **404 Not Found**, confirming that no account exists for this email:

```
HTTP/2 404 Not Found
[...]

Not Found
```

Request with a registered email address:

```
POST /api/v1/account/reset-password HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email": "test.account@pwnd.at",
    "passwordResetToken": "test",
    "newAuthToken":[...],
    "newWrappedMainKey":[...]
}
```

The server responds with a **403 Forbidden**, implicitly confirming the account exists but rejecting the invalid token:

```
HTTP/2 403 Forbidden
[...]

invalid password reset token supplied
```

Remediation Recommendation

- For endpoints that call `bcrypt.CompareHashAndPassword`, ensure the bcrypt comparison is always executed - regardless of whether the account exists - so that both code paths take comparable time. A dummy comparison against a fixed hash can be used when no account is found.
- Return consistent, generic HTTP status codes and error messages within each endpoint, regardless of whether the email is registered. For operations like password resets, consider succeeding silently (no-op) for unknown emails rather than signaling failure.
- Never expose sensitive internal identifiers such as `account_id` in responses from unauthenticated endpoints, regardless of whether the email is associated with an existing account.
- For the password reset endpoint, always return a `200 OK` with a generic message such as "If an account exists, a reset link has been sent" - irrespective of account existence.

I3: Missing Content-Security-Policy (CSP)	
Score	0.0 (Info)
Vector string	N/A
Target	• web/ • admin/
References	• https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP • https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-XSS-Protection

Overview

The Content Security Policy (CSP) is an effective security feature offered by modern browsers to implement an additional line of defense against cross-site scripting (XSS) attacks among others. The Zeitkapsl web client and admin dashboard either lack a Content Security Policy entirely or implement only a partial CSP, rendering this protection ineffective for a global scale.

> ⓘ **Note**
> The client is aware of this recommendation and may implement the suggested hardening in the future, following prior alignment.
>
> Status: **Acknowledged.**

Technical Details

The Content Security Policy (CSP) is a browser-based security mechanism designed to reduce the risk of specific client-side attacks. It allows a website to define a set of directives that instruct the browser to enforce restrictions on the execution of scripts and the loading of external resources. The primary purpose of CSP is to control which resources can be loaded and executed by the application. When properly implemented, CSP provides an effective layer of defense against cross-site scripting (XSS) by limiting the ability of attackers to inject and run malicious code within a web page.

Specifically for Zeitkapsl, restricting image sources to URLs that point only to the designated S3 bucket (e.g., `https://s3.eu-central-003.backblazeb2.com/...`) would be an effective way to mitigate potential attacks.

The backend does define a restrictive **Content-Security-Policy**.

```
[...]
func SecurityHeaders(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if strings.HasPrefix(r.URL.Path, "/api/v1/jobs") {
            w.Header().Set("Content-Security-Policy", "default-src 'self'
                'unsafe-inline';")
        } else if strings.HasPrefix(r.URL.Path, "/api/") {
            w.Header().Set("Content-Security-Policy", "default-src 'none';")
        }
        [...]
    })
}
```

server/pkg/controller/middleware.go:166-179

However, this configuration has no security effect. The CSP is enforced by the browser at document level - the policy is established once when the initial HTML response is received. All subsequent `fetch()` and XHR calls are governed by the policy of the originating document, not by headers on API responses. The browser discards CSP headers on non-document responses entirely. The CSP is present where it is meaningless and absent where it would be actionable.

The following diagram illustrates how the policy window closes at document load, making the API-level CSP unreachable:

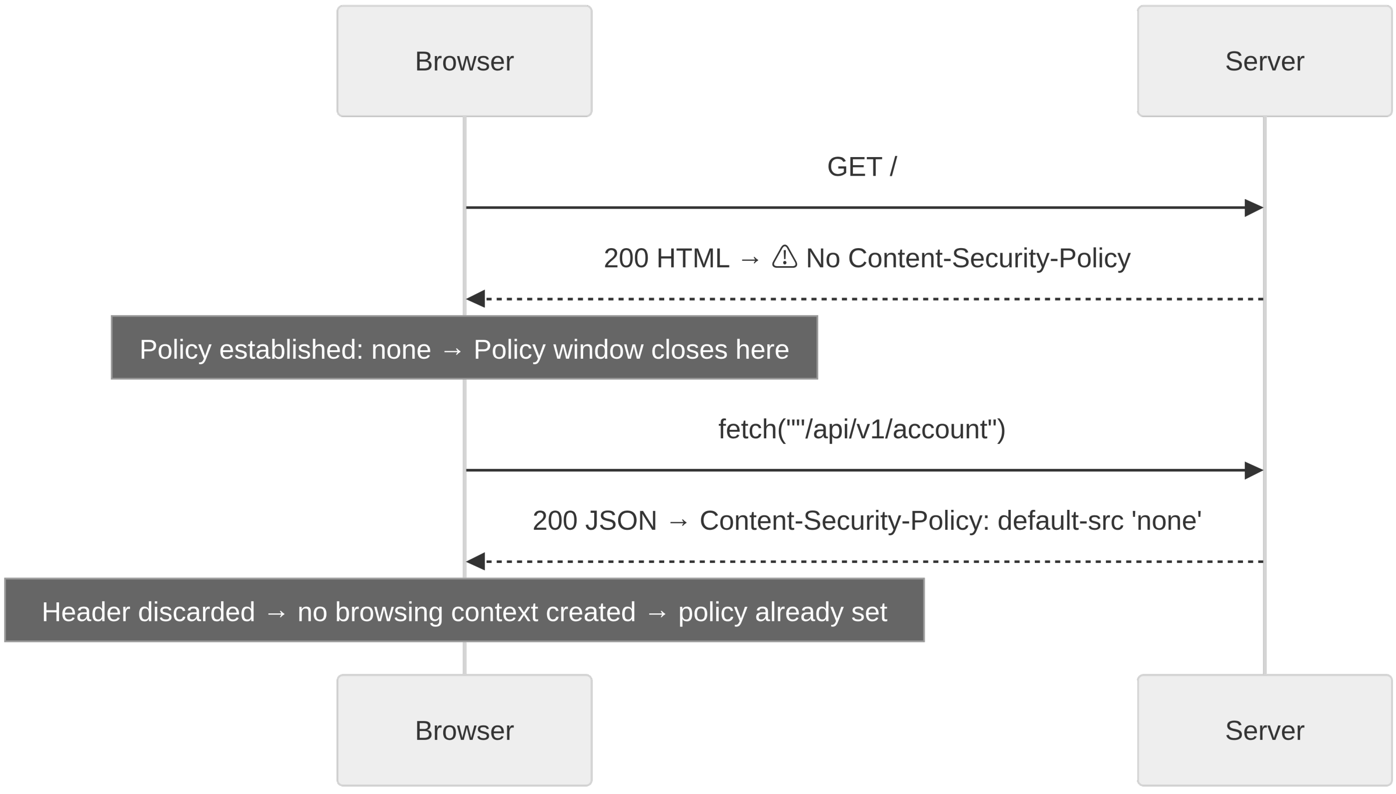


Figure 9 - Outdated Dependency Hierarchy

A review of the source code shows that a strict Content-Security-Policy was already considered but is currently disabled.

```
[...]
const config = {
    [...]
    kit: {
        [...]
        /*csp: {
            directives: {
                'default-src': ['self'],
                'script-src': ["'self' https://js.stripe.com"],
                'frame-src': ["'self' https://js.stripe.com"],
                'object-src': ['none'],
                'img-src': ["'self' https://tile.openstreetmap.org data:"],
                'frame-ancestors': ['none'],
            },
            mode: 'auto',
        },*/
        [...]
    }
};
[...]
```

web/svelte.config.js:5-30

In the service-worker code, several security-related HTTP headers are already applied to specific endpoints. For example, the video-download handler sets the following response headers:

```
[...]
function handleVideoDownload(playlistUrl, key, filename, fileSize) {
    [...]
    const responseHeaders = new Headers({
        'Content-Type': 'application/octet-stream; charset=utf-8',
        'Content-Security-Policy': "default-src 'none'",
        'X-Content-Security-Policy': "default-src 'none'",
        'X-WebKit-CSP': "default-src 'none'",
        'X-XSS-Protection': '1; mode=block',
        'Cross-Origin-Embedder-Policy': 'require-corp'
    });
    [...]
}
```

web/src/service-worker.js:95-128

- The `Content-Security-Policy` header (and its legacy equivalents) restricts all resource loading to **none**, effectively disabling any external content for this response.
- `X-XSS-Protection` enables the browser's built-in XSS filter and forces it to block the request when an attack is detected.
- `Cross-Origin-Embedder-Policy: require-corp` ensures that only resources that explicitly allow cross-origin embedding (via CORP) can be rendered, mitigating clickjacking and related attacks.

It is recommended to adopt a strict, modern Content-Security-Policy and retiring legacy headers such as `X-XSS-Protection`.

The admin dashboard sets a **Content-Security-Policy** header only on a handful of pages - e.g., via an inline meta tag that injects `script-src 'none'` - instead of applying a uniform CSP across the entire application.

```
[...]
function sanitizedHtml() {
    return `<html>
        <head>
            <meta http-equiv="Content-Security-Policy" content="script-src 'none'" />
        [...]
    </html>`
}
[...]
```

admin/src/lib/SafeHtml.svelte:26-39

Remediation Recommendation

- A **Content Security Policy** with the appropriate guidelines should be implemented. The following **reference** provides guidance on this topic. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP`.
- Make sure to choose fitting CSP directives. Even when present, some CSP directives can make the CSP less secure or even remove its security benefits entirely.

I4: Missing Input Validation in crypto primitives	
Score	0.0 (Info)
Vector string	N/A
Target	server/
References	https://owasp.org/www-project-mobile-top-10/2023-risks/m4-insufficient-input-output-validation

Overview

A missing safety check in the server's core decryption function can cause the program to crash when it receives empty or too short data. Several code paths already pass unvalidated data into this function, making the crash reachable in production. A related key-derivation function also silently produces weak keys when given empty input, potentially undermining encryption strength.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.
>
> Status: **Resolved.**

Technical Details

At the time of the audit, the `Decrypt` function was found to lack nil and length validation on its input, causing a runtime panic when called with nil or too-short data. Unlike `Encrypt`, which already guards against empty input, `Decrypt` immediately slices into the data buffer without checks.

Multiple callers such as `Share.Key()` and `Share.Name()` pass database-sourced byte slices directly into `Decrypt` without nil guards, making the panic reachable whenever a database field is null. Additionally, the `Hkdf` function accepts a nil `baseKey` without error, silently deriving a weak deterministic key that undermines the intended encryption guarantees.

The `Decrypt` function slices into `data` without performing nil or length validation:

```
[...]
func Decrypt(data []byte, key []byte) ([]byte, error) {
    [...]
    res, err := aesgcm.Open(nil, iv, data[aesgcm.NonceSize():], nil)
    [...]
}
[...]
```

server/pkg/crypto/crypto.go:140-150

If `data` is nil, the slice operation triggers a runtime panic. In contrast, `Encrypt` (lines 116–122) already validates input before processing.

Several callers pass database-sourced fields directly into `Decrypt` without prior nil guards, such as `Share.Key()`:

```
[...]
func (s *Share) Key(indexKey []byte) ([]byte, error) {
    return crypto.Decrypt(s.CryKey, indexKey)
}
[...]
```

server/pkg/api/share.go:233-235

The correct defensive pattern is already present in the codebase - `Share.Password()` (share.go: 246) performs a nil check before calling `Decrypt`, confirming this is an oversight rather than a deliberate design choice.

Steps to reproduce

One potential exploitation path is to upload a crafted media object to a collection. This crafted media object has ciphertexts that are shorter than the hardcoded IV length, causing clients to fail at decryption when syncing this collection.

Remediation Recommendation

- Add nil and minimum-length checks at the top of `Decrypt`, matching the pattern in `Encrypt`.
- Add a nil check on `baseKey` in `Hkdf` that returns an error instead of silently producing a weak key.
- Guard callers like `Share.Key()` and `Share.Name()` with nil checks, matching `Share.Password()`.

I5: Non-Compliance with Security Best Practices	
Score	0.0 (Info)
Vector string	N/A
Target	server/
References	• https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html • https://www.rfc-editor.org/rfc/rfc6238

Overview

Several instances, where the application does not fully align with established security best practices were identified. These include, among others, areas related to session management and the handling of security-critical account operations. While none of these findings represent directly exploitable vulnerabilities, they collectively widen the attack surface and could increase the impact of other issues if chained together.

The issues were remediated during the engagement and re-verification confirmed that the original findings have been successfully addressed.

Status: **Partially Resolved and Acknowledged.**

Technical Details

At the time of the audit, the application exhibited several deviations from security best practices across different areas. These include insufficient session management - such as sessions not being invalidated after password changes and TOTP codes being reusable within the same time window - as well as the absence of a password complexity policy. Additionally, inconsistencies were identified in error handling, input validation, and data format enforcement, alongside minor code quality concerns such as instances of dead code. While none of these findings represent directly exploitable vulnerabilities in isolation, addressing them would strengthen the application's overall security posture and reduce the potential impact should other issues be discovered or chained together.

1. TOTP Code Reuse Within Same Window

The same TOTP token could be reused multiple times within the same time window for a login. This deviates from the recommendations in RFC 6238, which states that a one-time password should not be accepted more than once within its validity period to prevent replay attacks.

```
[...]
func (c *AccountController) Login(request *Req[api.LoginRequest]) (any, error) {
    [...]
    if acc.MfaCodeEnabled {
        expectedCode := auth.GenerateTOTPFromSecret(acc.MfaCode)
        if expectedCode != request.Body.SecondFactor {
            return nil, errors.New(doma>in.ErrForbidden("MFA code required or invalid"))
        }
    }
    [...]
}
[...]
```

server/pkg/controller/account.go:335-393

The code validates the TOTP code against the expected value for the current time window but does not track whether the code has already been consumed.

Once a valid code is generated, it remains accepted for the entirety of the time step.

```
[...]
func GenerateTOTPFromSecret(decodedSecret []byte) string {
    return GenerateTOTPFromSecretAndTime(decodedSecret, time.Now())
}
[...]
```

server/pkg/auth/2fa.go:29-31

2. Sessions Not Invalidated on Password Change

Upon changing a password, the application does not invalidate other active sessions associated with the same account.

This deviates from established best practices such as the OWASP Session Management Cheat Sheet, which recommends that all existing sessions be invalidated after a credential change to ensure that any previously compromised or stale session - such as one held by an attacker or an unattended device - cannot persist beyond the remediation action.

```
[...]
func (c *AccountController) LoadAccountFromSession(request *http.Request) ([...]) {
    session := auth.GetSession(request)
    acc, err := c.Db.FindAccountById(request.Context(), session.UserId)
    if err != nil {
        return nil, domain.ErrNotFound(err)
    }

    return &acc, err
}
[...]
func (c *AccountController) ChangePassword(r *Req[api.ChangePasswordRequest]) ([...]) {
    acc, err := c.LoadAccountFromSession(r.R)
    if err != nil {
        return nil, err
    }

    err = acc.ChangePassword(r.Body.AuthToken, r.Body.NewAuthToken, [...])
    [...]
    err = db.Tx(r.Ctx, c.Db, func(tx *db.Db) error {
        err = account.Update(r.Ctx, c.Db, acc)
        [...]
        return c.accountService.SendCustomerActionMail(r.Ctx, tx, acc, [...])
    })
    [...]
}
[...]
```

server/pkg/controller/account.go:914-922,516-548

3. No Password Complexity Policy

Because the application derives a cryptographic key from the user's raw password via PBKDF2-SHA512 (1,000,000 iterations) before transmitting it to the server, the server never sees the plaintext password and therefore cannot enforce any password policy. Client-side enforcement is the only layer at which a minimum length or complexity requirement can be applied.

At the time of testing, all password entry points displayed a visual strength indicator via the `check-password-strength` library, however this indicator was purely informational and did not prevent form submission. A single-character password such as `0` was accepted without rejection at either layer.

```
[...]
const schema = yup.object().shape({
    currentPassword: yup.string().required([...]),
    newPassword: yup.string().required([...]),
    newPasswordConfirmation: yup.string()
        .oneOf([yup.ref("newPassword"), null], [...])
        .required([...])
});
[...]
```

web/src/routes/(app)/settings/password/+page.svelte:20-24

The Yup validation schema only enforces that the new password field is not empty (`.required()`), with no minimum length or complexity rule. The strength indicator is computed on input but never gates form submission:

```
[...]
let passwordStrengthResult;
function checkPasswordStrength() {
    passwordStrengthResult = passwordStrength(newPassword);
}
[...]
```

web/src/routes/(app)/settings/password/+page.svelte:28-30

4. Audit Log Records Success on Failure

Several controller methods write a success audit log entry regardless of whether the preceding database transaction actually succeeded.

The audit call is not guarded by an error check, which can produce a misleading audit trail - recording an operation as completed when it may have partially or fully failed. In the case of `ChangePassword`, a failed transaction results in both a "Failed" and a "Successful" entry being written for the same operation.

The affected functions are listed below:

Function	Line	Written When
`ResetPassword`	508	After `db.Tx` - unconditionally, even if the transaction failed
`ChangePassword`	543	After `db.Tx` - unconditionally; on failure, both a "Failed" (line 534) and "Successful" (line 543) entry are written
`ReactivateAccount`	631	After `db.Tx` - unconditionally, even if the transaction failed
`ChangeNotification-Settings`	760	After `db.Tx` - unconditionally, even if the transaction failed
`ChangeName`	797	After `UpdateContactName` - unconditionally, even if the external call failed
`ChangeEmail`	823	After `db.Tx` - unconditionally, even if the transaction failed
`Logout` (**else** branch)	422	After `DestroySession` - error is ignored, audit written unconditionally

All references: server/pkg/controller/account.go

5. Stripe and campaign tool email updates not rollback-safe

During email change verification, Stripe and campaign tool emails are updated via external API calls inside a DB transaction. If a later step fails and the transaction rolls back, external systems retain the new email while the DB keeps the old one, causing permanent inconsistency.

```
[...]
func (s *Service) VerifyEmailChange(ctx context.Context, tx *db.Db, token string) error {
    [...] // tx starts here
    if len(quota.StripeCustomerID) > 0 {
        _, err = s.StripeService.UpdateEmail(quota.StripeCustomerID, req.NewEmail)
        if err != nil {
            return fmt.Errorf("failed...%s: %w", req.NewEmail, err)
        }
    }

    err = s.EmailCampaignService.UpdateContactEmail(ctx, [...], req.NewEmail)
    if err != nil {
        return fmt.Errorf("failed...%s: %w", req.NewEmail, err)
    }
    [...] // tx potentially rolls back here
}
[...]
```

server/pkg/domain/account/service.go:1234-1278

Move external API calls after the transaction commits, or use a custom rollback pattern.

Additional Issues that were identified

> Beyond the findings detailed above, several additional deviations from best practices were identified across error handling, input validation, data format consistency, and general code quality. While these do not represent directly exploitable vulnerabilities in isolation, addressing them would improve the overall robustness and maintainability of the codebase.

Error handling:

- `DestroySession` and `DestroySessionById` always return `nil` regardless of database errors - callers may incorrectly assume the session was destroyed (server/pkg/auth/middleware.go: 129-169).
- `VerifyEmailChange` returns the wrong error variable (`err` instead of `auditError`), silently swallowing audit log failures (server/pkg/domain/account/service.go:1272-1275).
- `ChangePassword` uses `c.Db` instead of the transaction handle `tx` inside a `db.Tx` block, bypassing transactional atomicity (server/pkg/controller/account.go:533).
- MFA check wraps a domain error in `errors.New()`, producing a double-wrapped error (server/pkg/controller/account.go:356).

Input validation:

- CryKey length validation is inconsistent: some endpoints use `<` (allowing oversized keys) while others use `!=` (server/pkg/api/media.go:27, 194; server/pkg/api/video.go:87).
- Share AccessKey and CryPassword use loose range bounds instead of exact expected lengths (server/pkg/api/share.go:46-52, 106-112).
- AuthToken and PasswordResetToken validated with max length 100 instead of exact expected ~44 characters (server/pkg/api/account.go:163-169, 208).
- AspectRatio allows zero, which could cause downstream division issues (server/pkg/api/media.go:103, 139).
- `CreateAnonymousMediaRequest` has fewer validation checks than `CreateMediaRequest` (server/pkg/api/media.go:125-148).

Data format issues:

- `IsInstant` JSON struct tag contains a tab character, causing the field to never parse correctly (server/pkg/api/account.go:432).
- Latitude/longitude comma-to-dot replacement is not reversible on round-trip (server/pkg/api/metadata.go:52, 58, 172, 179).

Code quality:

- `IDFromString` silently falls back to 0 on parse error (server/pkg/snowflake/snowflake.go:143-145).
- Snowflake error message prints the current timestamp instead of the maximum supported timestamp (server/pkg/snowflake/snowflake.go:96).
- Dead code: unreachable `err != nil` check in `NewFilesystemTarget` (server/pkg/cache/filesystem.go:24-26).
- `ClearAll` does not reset LRU state or `sizeUsed` - likely only matters if called during normal operation rather than shutdown (server/pkg/cache/cache.go:175-177; server/pkg/cache/memory.go:96-101).
- There is a hardcoded invite code that allows to bypass code validation (server/pkg/domain/account/service.go:1394). Consider removing this hardcoded magic value from the codebase and pass it to the server at runtime.
- A previous auditor left a comment about zero salt in share password derivation (core/pkg/core/share.go:545, web/src/lib/share.js:71, web/src/lib/share.js:13). Using a unique salt would be best practice, although in this case the password is high-entropy, making brute-force anyway infeasible.

Steps to reproduce

1. PoC - TOTP Code Reuse Within Same Window

First request within the time window using a TOTP code:

```
POST /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email":"test.account@pwnd.at",
    "authToken":"Knq...jQ=",
    "secondFactor":"456675"
}
```

Successful response from Server:

```
HTTP/2 200 OK
[...]

{
    "id":[...],
    "firstName":[...],
    "lastName":[...],
    "email":"test.account@pwnd.at",
    "wrappedMainKey":"iG5...4Xw",
    "defaultCollection":"208...377",
    "wrappedDefaultCollectionKey":"6/J...JgF",
    "language":"en-US",
    "status":1,
    "apiKey":"208...10g==",
    "kdfSettings":{
        "algorithm":"PBKDF2-SHA512",
        "iterations":1000000
    },
    "readOnly":false
}
```

Second request within the same time window reusing the identical TOTP code:

```
POST /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "email":"test.account@pwnd.at",
    "authToken":"Knq...jQ=",
    "secondFactor":"456675"
}
```

Successful response from Server:

```
HTTP/2 200 OK
[...]

{
    "id":[...],
    "firstName":[...],
    "lastName":[...],
    "email":"test.account@pwnd.at",
    "wrappedMainKey":"iG5...4Xw",
    "defaultCollection":"208...377",
    "wrappedDefaultCollectionKey":"6/J...JgF",
    "language":"en-US",
    "status":1,
    "apiKey":"208...YpQ==",
    "kdfSettings":{
        "algorithm":"PBKDF2-SHA512",
        "iterations":1000000
    },
    "readOnly":false
}
```

2. PoC - Sessions Not Invalidated on Password Change

> For this proof of concept, two concurrent sessions were established - Session 1 and Session 2. Session 1 was then used to perform a password change, after which Session 2 should have been invalidated or required to re-authenticate.

Authenticated request using Session 2 prior to password change:

```
GET /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
Cookie: zeitkapsl_user=208...ohQ==
[...]
```

Successful response from server:

```
HTTP/2 200 OK
[...]

{
    "id":[...],
    "firstName":[...],
    "lastName":[...],
    "email":[...],
    "wrappedMainKey":"CuN...lvC",
    "defaultCollection":"208...377",
    "wrappedDefaultCollectionKey":"6/J...JgF",
    "language":"en-US",
    "status":1,
    "apiKey":"",
    "kdfSettings":{
        "algorithm":"PBKDF2-SHA512",
        "iterations":1000000
    },
    "readOnly":false
}
```

Password change request issued via Session 1:

```
POST /api/v1/account/change-password HTTP/2
Host: app.test.zeitkapsl.at
Cookie: zeitkapsl_user=208...v6A==
[...]

{
    "authToken":"Knq...jQ=",
    "newAuthToken":"2Jl...Gk=",
    "newWrappedMainKey":"CuN...lvC"
}
```

Authenticated request using Session 2 after password change:

```
GET /api/v1/account/login HTTP/2
Host: app.test.zeitkapsl.at
Cookie: zeitkapsl_user=208...ohQ==
[...]
```

Successful response from server:

```
HTTP/2 200 OK
[...]

{
    "id":[...],
    "firstName":[...],
    "lastName":[...],
    "email":[...],
    "wrappedMainKey":"CuN...lvC",
    "defaultCollection":"208...377",
    "wrappedDefaultCollectionKey":"6/J...JgF",
    "language":"en-US",
    "status":1,
    "apiKey":"",
    "kdfSettings":{
        "algorithm":"PBKDF2-SHA512",
        "iterations":1000000
    },
    "readOnly":false
}
```

Remediation Recommendation

- Track consumed TOTP codes within their validity period and reject any code that has already been used, in accordance with RFC 6238.
- Enforce a minimum password length and complexity policy on the client side and block form submission when the policy is not met, consistently across all password entry points.
- Move success audit log entries inside the success path of their respective transactions so they are only written when the operation has actually completed.
- Standardize input validation across all endpoints to use exact expected lengths, propagate errors explicitly rather than silently falling back to default values, and disallow zero or out-of-range values where applicable.
- Correct data format issues such as the malformed `IsInstant` JSON struct tag and ensure that string-based transformations like latitude/longitude formatting are lossless on round-trip.
- Remove dead code, fix inconsistent error variable usage, and ensure database operations within transaction blocks use the correct transaction handle.

I6: Outdated Dependencies	
Score	0.0 (Info)
Vector string	N/A
Target	• web/ • server/ • core/ • admin/ • desktop/ • cli/ • tools/ • translations/ • ios/ • android/
References	• https://owasp.org/www-project-dependency-check/ • https://nvd.nist.gov/vuln/search#/nvd/home?resultType=records • https://docs.npmjs.com/cli/v10/commands/npm-audit?v=true • https://go.dev/doc/modules/managing-dependencies

Overview

Even without insecure code, software systems can have vulnerabilities due to outdated or insecurely implemented dependencies. Multiple applications used several outdated software dependencies that contain known security vulnerabilities.

The client is aware of the outdated dependencies and has already upgraded several of them to the extent feasible within the current scope.

Status: **Partially Addressed.**

Technical Details

At the time of the audit, multiple libraries used across the projects were outdated and contained publicly disclosed vulnerabilities. Attackers may exploit these vulnerabilities to compromise the server or target clients interacting with the application.

The codebase references these outdated dependencies directly, which increases the risk of system compromise if the vulnerabilities are exploited.

Outdated npm dependencies in subdirectory `web/`

> **Current state:** The dependency set for the `web` module is relatively tidy - only a single high-severity issue is present, with a handful of medium- and low-severity advisories, indicating that version management is generally good but could be tightened.

Recommendation: Prioritise the high-severity Rollup upgrade, then address the medium-severity packages in the upcoming release, and schedule the low-severity updates for the next routine dependency bump. Prompt remediation will reduce supply-chain risk and keep the application aligned with security best practices.

Package	Version	Vulnerability	Severity	Scope
rollup	4.52.4	CVE-2026-27606	HIGH	Dev-Dep.
svelte	5.53.0	CVE-2026-27901, CVE-2026-27902	MEDIUM	Shared
devalue	5.6.3	CVE-2026-30226, GHSA-mwv9-gp5h-frr4	MEDIUM	Dev-Dep.
dompurify	3.3.1	CVE-2026-0540	MEDIUM	Dep.
cookie	0.6.0	CVE-2024-47764	LOW	Dev-Dep.
@sveltejs/kit	2.53.0	GHSA-fpg4-jhqr-589c	LOW	Dev-Dep.

The dependency chart below shows the top-level packages that introduce the outdated and vulnerable libraries.

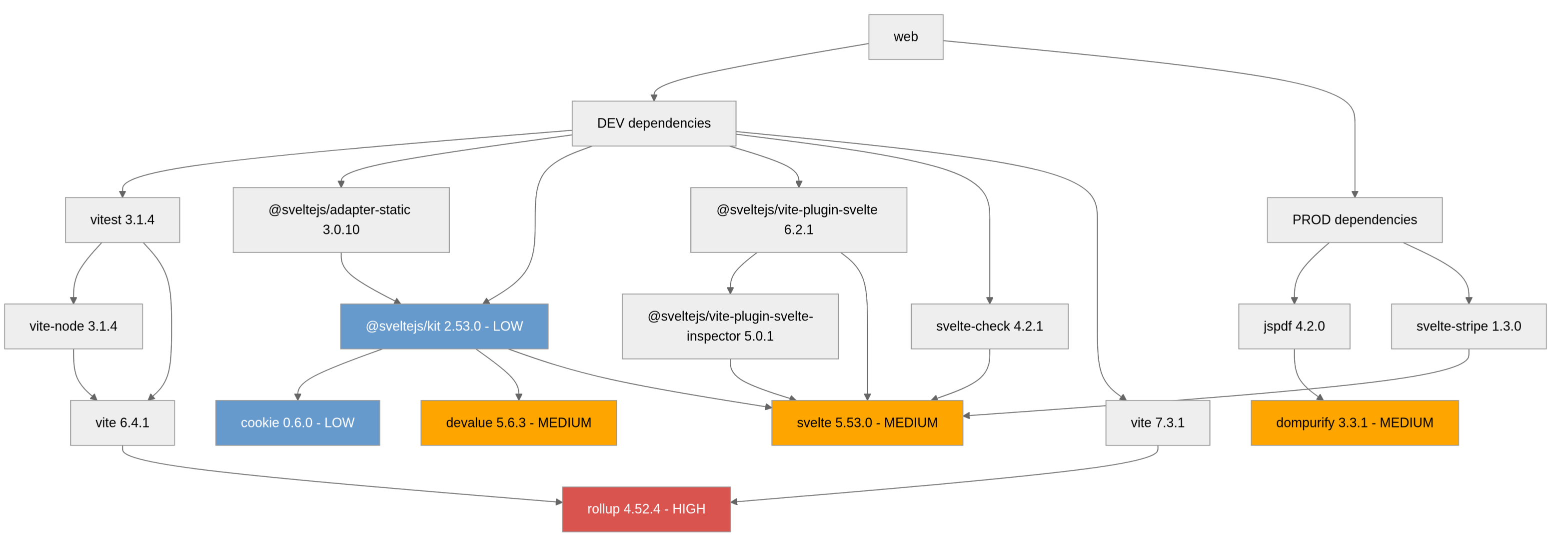


Figure 10 - Outdated Dependency Hierarchy

Outdated npm dependencies in subdirectory `admin/`

> **Current state:** The overall dependency landscape is moderately populated with vulnerabilities - several high- and medium-severity issues are present, indicating room for improvement in version management.

Recommendation: Prioritise updating the multiple high-severity packages, then address the medium-severity dependencies in the upcoming release, and schedule the low-severity updates for the next routine dependency bump.

Package	Version	Vulnerability	Severity	Scope
rollup	4.46.2	CVE-2026-27606	HIGH	Dev-Dep.
devalue	5.3.2	CVE-2026-22775, CVE-2026-22774, GHSA-33hq-fvwr-56pm, GHSA-8qm3-746x-r74r, CVE-2026-30226, GHSA-mwv9-gp5h-frr4	HIGH	Dev-Dep.
tar	7.4.3	CVE-2026-23950, CVE-2026-24842, CVE-2026-23745, CVE-2026-26960, CVE-2026-29786, CVE-2026-31802	HIGH	Dev-Dep.
@sveltejs/kit	2.43.8	CVE-2025-67647	HIGH	Dev-Dep.
vite	7.1.5	CVE-2025-62522	MEDIUM	Dev-Dep.
svelte	5.38.0	CVE-2026-27125, CVE-2026-27122, CVE-2026-27121, CVE-2026-27901	MEDIUM	Dev-Dep.
dompurify	3.3.0	CVE-2026-0540	MEDIUM	Dep.
cookie	0.6.0	CVE-2024-47764	LOW	Dev-Dep.

The dependency chart below shows the top-level packages that introduce the outdated and vulnerable libraries.

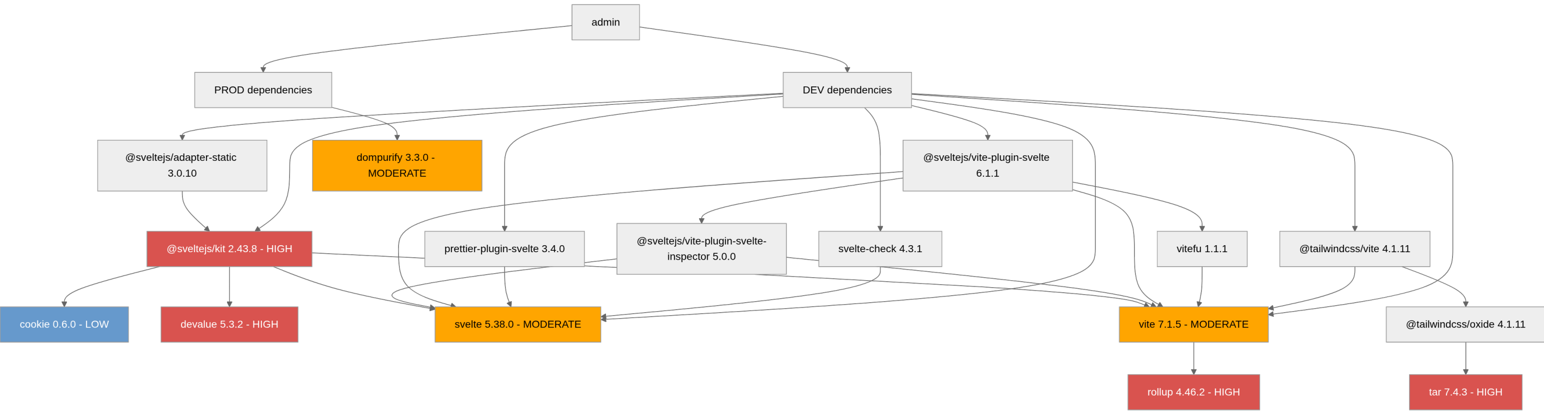


Figure 11 - Outdated Dependency Hierarchy

Outdated npm dependencies in subdirectory `desktop/frontend/`

> **Current state:** The dependency set has several high- and medium-severity vulnerabilities, showing that version management needs improvement.

Recommendation: Update the high-severity packages first, then the medium-severity ones in the next release, and handle low-severity fixes during the regular dependency bump.

Package	Version	Vulnerability	Severity	Scope
glob	10.4.5	CVE-2025-64756	HIGH	Dev-Dep.
minimatch	9.0.5	CVE-2026-26996, CVE-2026-27903, CVE-2026-27904	HIGH	Dev-Dep.
rollup	4.52.4	CVE-2026-27606	HIGH	Dev-Dep.
svelte	5.33.2	CVE-2026-27125, CVE-2026-27122, CVE-2026-27121, CVE-2026-27901	MEDIUM	Dev-Dep.
vite	7.1.5	CVE-2025-62522	MEDIUM	Dev-Dep.
brace-expansion	2.0.1	CVE-2025-5889	LOW	Dev-Dep.

The dependency chart below shows the top-level packages that introduce the outdated and vulnerable libraries.

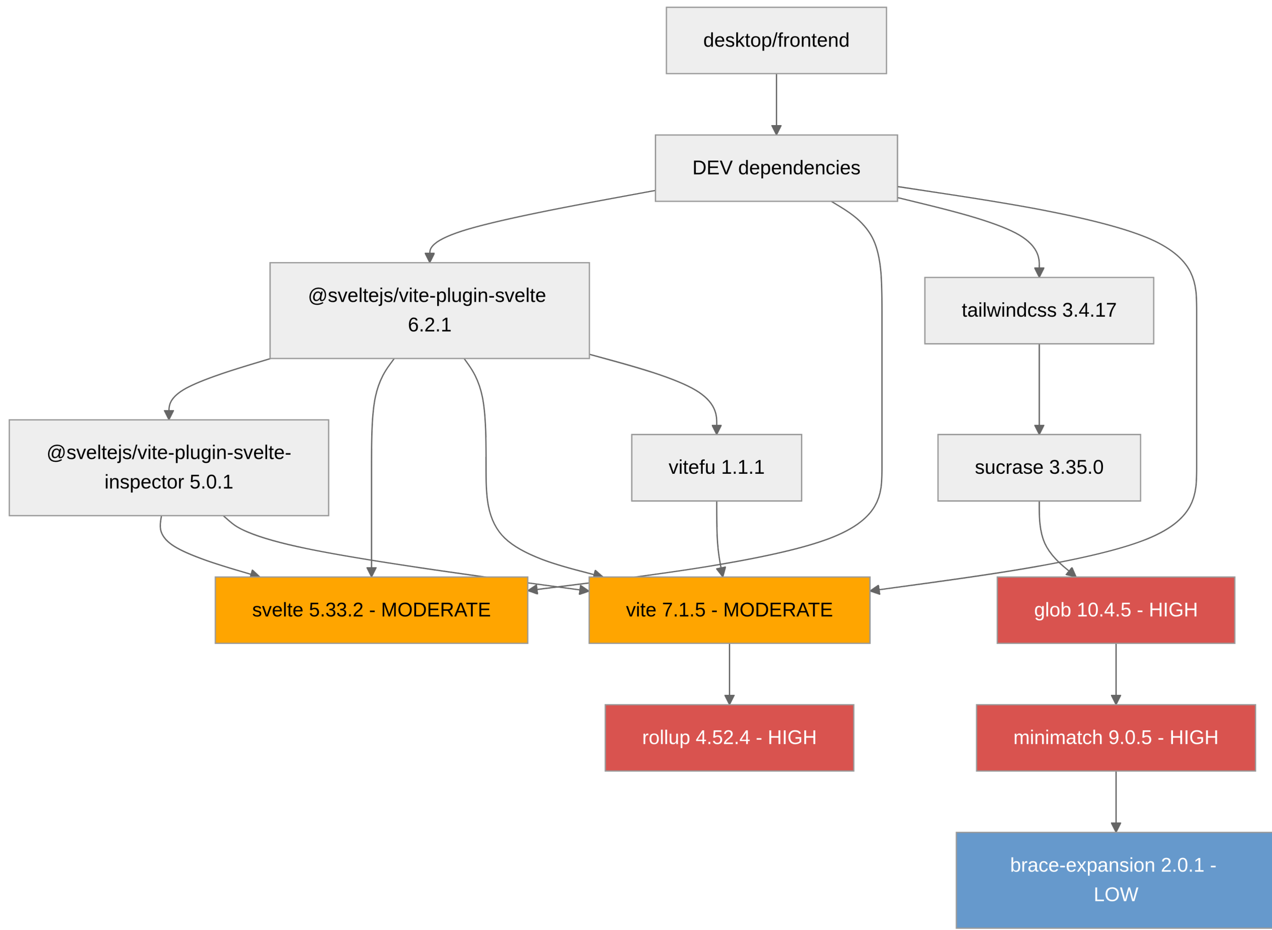


Figure 12 - Outdated Dependency Hierarchy

Outdated go dependencies in subdirectory `desktop/`

> **Current state:** The project's Go code is built on **go1.25.1**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

> **Additional note:** All third-party modules referenced by the project are up-to-date and, according to the latest vulnerability scans, do **not** contain any known security issues. The only exposure stems from the outdated Go standard library.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121 CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
stdlib	go1.25.1	CVE-2026-27142, CVE-2025-58183	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
net/mail	go1.25.1	CVE-2025-61725	HIGH
archive/zip	go1.25.1	CVE-2025-61728	MEDIUM
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `server/`

> **Current state:** The project's Go code is built on **go1.25.1**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121, CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
html/template	go1.25.1	CVE-2026-27142	HIGH
net/mail	go1.25.1	CVE-2025-61725	HIGH
archive/tar	go1.25.1	CVE-2025-58183	MEDIUM
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
github.com/go-chi/chi/v5	v5.2.3	GO-2026-4316	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
stdlib	go1.25.1	CVE-2025-61728	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `core/`

> **Current state:** The project's Go code is built on **go1.25.1**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

> **Additional note:** All third-party modules referenced by the project are up-to-date and, according to the latest vulnerability scans, do **not** contain any known security issues. The only exposure stems from the outdated Go standard library.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121, CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
stdlib	go1.25.1	CVE-2026-27142, CVE-2025-58183	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
net/mail	go1.25.1	CVE-2025-61725	HIGH
archive/zip	go1.25.1	CVE-2025-61728	MEDIUM
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `cli/`

> **Current state:** The project's Go code is built on **go1.25.1**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

> **Additional note:** All third-party modules referenced by the project are up-to-date and, according to the latest vulnerability scans, do **not** contain any known security issues. The only exposure stems from the outdated Go standard library.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121, CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
stdlib	go1.25.1	CVE-2026-27142, CVE-2025-58183	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
net/mail	go1.25.1	CVE-2025-61725	HIGH
archive/zip	go1.25.1	CVE-2025-61728	MEDIUM
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `tools/`

> **Current state:** The project's Go code is built on **go1.25.1** and **go1.25.3**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

> **Additional note:** All third-party modules referenced by the project are up-to-date and, according to the latest vulnerability scans, do **not** contain any known security issues. The only exposure stems from the outdated Go standard library.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121, CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
stdlib	go1.25.1	CVE-2026-27142, CVE-2025-58183	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
net/mail	go1.25.1	CVE-2025-61725	HIGH
archive/zip	go1.25.1	CVE-2025-61728	MEDIUM
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `tools/scraper/`

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.3	CVE-2025-61730, CVE-2025-68121	CRITICAL
crypto/x509	go1.25.3	CVE-2025-61727, CVE-2025-61729	HIGH
net/url	go1.25.3	CVE-2026-25679, CVE-2025-61726	HIGH
stdlib	go1.25.3	CVE-2026-27142, CVE-2025-61728	HIGH
os	go1.25.3	CVE-2026-27139	LOW

Outdated go dependencies in subdirectory `translations/`

> **Current state:** The project's Go code is built on **go1.25.1**, whose standard-library packages contain a **critical flaw** (crypto/tls) plus numerous high- and medium-severity vulnerabilities. Automated scans have already identified dozens of vulnerable function calls throughout the codebase (e.g., `html/template.Execute`, `url.Parse`, `tls.Conn.HandshakeContext`, `x509.Certificate.Verify`, `net/http.Cookie`, etc.), confirming that the problems are actively exercised in production paths.

> **Additional note:** All third-party modules referenced by the project are up-to-date and, according to the latest vulnerability scans, do **not** contain any known security issues. The only exposure stems from the outdated Go standard library.

Recommendation: Upgrade the Go toolchain to at least **go1.25.8** (or the latest stable release) – this resolves the critical, high- and many medium-severity stdlib package issues in a single step.

Package	Version	Vulnerability	Severity
crypto/tls	go1.25.1	CVE-2025-61730, CVE-2025-68121, CVE-2025-58189	CRITICAL
crypto/x509	go1.25.1	CVE-2025-61727, CVE-2025-61729, CVE-2025-58188, CVE-2025-58187	HIGH
stdlib	go1.25.1	CVE-2026-27142, CVE-2025-61728, CVE-2025-58183, CVE-2025-61725	HIGH
net/url	go1.25.1	CVE-2026-25679, CVE-2025-61726, CVE-2025-47912	HIGH
encoding/pem	go1.25.1	CVE-2025-61723	HIGH
encoding/asn1	go1.25.1	CVE-2025-58185	MEDIUM
net/http	go1.25.1	CVE-2025-58186	MEDIUM
net/textproto	go1.25.1	CVE-2025-61724	MEDIUM
os	go1.25.1	CVE-2026-27139	LOW

Outdated ruby dependencies in subdirectory `ios/` and `android/`

> Both the **iOS** and **Android** codebases rely exclusively on up-to-date third-party modules, and no known vulnerabilities were detected in their respective dependency graphs. This demonstrates that the package management for each project is clean, well-maintained, and free of outdated or insecure dependencies.

These vulnerabilities are not relevant in this context, as both affected libraries are transitive dependencies pulled in by Fastlane solely for release automation tasks such as uploading builds to Google Play - they are never used to handle user-controlled input or interact with any S3 storage directly. They are nonetheless included in this report and can be addressed with a low-priority dependency upgrade when convenient.

Package	Version	Vulnerability	Severity
faraday	1.10.4	CVE-2026-25765	MEDIUM
aws-sdk-s3	1.199.0	CVE-2025-14762	MEDIUM

Remediation Recommendation

- Make sure to always update dependencies to the latest version with security patches.
- Upgrade the Go toolchain to the latest stable release.
- Employ an SBOM generation and auditing tool (in your CI/CD) to automatically get notified when new library versions become available or vulnerabilities for currently used library versions are found.

I7: Random node id in snowflake generator potentially allows for Snowflake Collision	
Score	0.0 (Info)
Vector string	N/A
Target	server/
References	-

Overview

The application's ID generation mechanism selects node identifiers randomly from a pool of only 256 possible values, creating a non-negligible probability of collisions as the number of generator instances grows. Since these IDs serve as primary keys for sessions, media, and other core entities, duplicates could lead to unexpected database query failures, data integrity issues or unintended cross-referencing between records in production environments.

> ⓘ **Note**
> The issue was remediated during the engagement and re-verification confirmed that it had been successfully addressed and the affected code path was fixed.
>
> Status: **Resolved.**

Technical Details

At the time of the audit, the application's snowflake ID generator was found to select its node ID randomly from only 256 possible values (`NodeBits = 8`). By the birthday paradox, with as few as 19 concurrent generator instances, the probability of at least one node ID collision exceeds 50%.

This risk is further amplified by the fact that `NewID()` creates a fresh generator per invocation rather than reusing a shared instance, making collisions with the global `IdGen` highly likely. Since snowflake IDs are used as primary keys for sessions, media, and other core entities, node ID collisions can result in duplicate identifiers being issued, potentially leading to failed database queries, data overwrites, session conflicts, or unintended association between unrelated records.

The `NewSnowflakeGenerator` function selects a random 8-bit node ID:

```
[...]
func NewSnowflakeGenerator() (*SnowflakeGenerator, error) {
    node, err := NewNode(rand.Int63n(-1 ^ (-1 << NodeBits)))
    [...]
}
[...]
```

server/pkg/snowflake/snowflake.go:164-166

With `NodeBits = 8`, there are only 256 possible node IDs. Applying the birthday paradox, just 19 generator instances are sufficient for a greater than 50% probability of at least one collision. Furthermore, `NewID()` creates a fresh generator per call (see related finding), making collisions with the global `IdGen` particularly likely.

Steps to reproduce

Instantiate multiple `SnowflakeGenerator` instances and generate snowflake IDs within the same millisecond to verify whether duplicate identifiers are produced.

Remediation Recommendation

- Assign node IDs deterministically, for example derived from configuration or instance index, rather than selecting them randomly.
- Alternatively, increase the value of `NodeBits` to significantly reduce the probability of collisions.
- Remove the `NewID()` function and rely exclusively on the global `IdGen` singleton to avoid spawning redundant generator instances. "On the fly generation" of snowflake generators to generate one random value defeats the purpose of having a snowflake generator.

I8: Single snowflake generator instance can produce duplicate IDs due to Race Condition in timestamp capture	
Score	0.0 (Info)
Vector string	N/A
Target	server/
References	-

Overview

A race condition in the snowflake ID generator allows concurrent requests to produce duplicate identifiers Since these IDs serve as primary keys for sessions, media, and other core entities, duplicates could lead to data integrity issues or unintended cross-referencing between records in production environments - though the precise timing required and additional safeguards such as separate signing keys and database constraints make this extremely difficult to exploit in practice.

Note
The issue was remediated during the engagement and re-verification confirmed that it had been successfully addressed and the affected code path was fixed.

Status: **Resolved.**

Technical Details

At the time of the audit, the snowflake ID generator's `Generate()` method captures the current timestamp via `time.Now()` before acquiring the mutex, then passes it into `GenerateWithTime()` where the lock is held. Under concurrent load, this allows for the following schedule: Goroutine G1 and G2 read T0, while G3 reads T1. G1 obtains the lock first and creates an ID for T0 and step 0. Before G2 reaches the lock, it is preempted by G3, therefore G3 obtains the lock before G2. This advances the snowflake generator to T1, thereby resetting the step to 0. Finally, G2 enters the lock, `GenerateWithTime` sees T0 as a "new" timestamp and "advances" to it, thereby resetting the step to 0. Therefore, G1 and G2 obtain the same snowflake ID.

Since snowflake IDs serve as session identifiers, a precisely timed attack exploiting this race could yield a session token matching that of another active user, enabling session hijacking and unauthorized access. The fix involves moving the timestamp capture inside the mutex so that ordering is always monotonic and the step counter increments correctly.

`Generate()` calls `time.Now()` before entering the critical section, passing the result into `GenerateWithTime()` where the mutex is actually acquired:

```
[...]
func (n *Node) Generate() ID {
    id, err := n.GenerateWithTime(time.Now())
    [...]
}
[...]
```

server/pkg/snowflake/snowflake.go:70-71

Inside `GenerateWithTime`, the passed-in timestamp is compared against the last recorded time. If it differs - including when a stale timestamp is older than the current value because another goroutine advanced it in the meantime - the step counter resets to 0 and the internal clock is overwritten:

```
[...]
func (n *Node) GenerateWithTime(ts time.Time) (ID, error) {
    n.mu.Lock()
    defer n.mu.Unlock()
    now := ts.Sub(n.epoch).Milliseconds()
    if now == n.time {
        n.step = (n.step + 1) & n.stepMask
        [...]
    } else {
        n.step = 0
    }
    n.time = now
    [...]
}
[...]
```

server/pkg/snowflake/snowflake.go:78-93

This means the scheduling scenario G1(T0) → G3(T1) → G2(T0) produces the same snowflake twice: goroutine G2 reads timestamp T0 but is preempted before locking; goroutine G3 acquires the lock, advances the internal time to T1 with step 0, and releases; G2 then enters the lock with its stale T0, resets step to 0, and generates `T0 || node_id || 0` - an ID already issued in an earlier millisecond.

Steps to reproduce

While this race condition is real, exploiting it in practice requires precise goroutine scheduling that is difficult to trigger deterministically. Moreover, cross-escalation between user and admin sessions is not possible, as the two session tables use separate HMAC signing keys - a token valid in one context will be rejected by the other. Even within the same table, a duplicate snowflake ID would violate the `PRIMARY KEY` constraint, causing the second `INSERT` to fail with a database error. The practical impact is therefore a denied login for the unlucky request, not session hijacking.

Remediation Recommendation

- Move the `time.Now()` call inside the mutex in `GenerateWithTime()`, or call it within `Generate()` after acquiring the lock.
- Alternatively, detect when the passed timestamp is older than `n.time` and map it forward to the current time instead of resetting the step counter.

I9: Snowflake IDs Oracles	
Score	0.0 (Info)
Vector string	N/A
Target	server/
References	-

Overview

These issues permit an attacker confirm if certain snowflake ids exist. This may not have a major impact but if enumerating collection id's of other users for followup exploits, could still be a useful gadget.

The issues were remediated during the engagement and re-verification confirmed that the original exploitation is no longer possible.

Status: **Resolved.**

Technical Details

At the time of the audit it was discovered that an attacker could query several different endpoints to determine whether specific media IDs or collection IDs exist. Although this does not currently expose sensitive data or constitute a high-severity vulnerability, it violates the principle of least privilege and can be used for reconnaissance. The application should be updated to block enumeration of Snowflake IDs - e.g., by normalising responses, enforcing stricter access controls, or consolidating the checks into a single, properly-authenticated endpoint.

1. CreateShare

The `POST /api/v1/share` endpoint leaks information about a collection's existence through its error handling:

- If the supplied **collection ID does not exist**, the controller returns an **HTTP 500 Internal Server Error**.
- If the collection **exists but belongs to another user**, the service returns a **Forbidden** error, which ultimately results in a **different HTTP status code** (e.g., 403).

Because the responses differ, an attacker can use the endpoint as an **existence oracle** to confirm whether a particular `collectionId` is valid.

The controller treats every error from `CreateShare` as a generic *500* response. When the service fails because the collection ID is missing, the client receives a distinct status code from the case.

```
[...]
func (c ShareController) CreateShare(r *Req[api.CreateShareRequest]) ([...]) {
    session := auth.GetSession(r.R)
    shareId, err := db.TxR[snowflake.ID](r.Ctx, c.Db, func(tx *db.Db) ([...]) {
        return c.mediaService.CreateShare([...], r.Body.CollectionId, [...])
    })
    if err != nil {
        return nil, http.StatusInternalServerError, err
    }
    [...]
}
[...]
```

server/pkg/controller/share.go:63-73

When the collection exists but the caller is not its owner, the service returns a `domain.ErrForbidden` error. This error propagates back to the controller, resulting in a response that differs from the "not-found" case.

```
[...]
func (s Service) CreateShare([...], collectionId, createdBy snowflake.ID, [...]) ([...]) {
    c, err := tx.FindCollectionById(ctx, collectionId)
    if err != nil {
        return nil, err
    }
    if c.CreatedBy != createdBy {
        return nil, errors.New(domain.ErrForbidden("Forbidden"))
    }
    [...]
}
[...]
```

server/pkg/domain/media/share.go:19-50

2. DeleteCollection

The `DELETE /api/v1/collection` endpoint also reveals whether a collection ID is valid.
Because the controller and service return **different HTTP status codes** for "collection does not exist" versus "collection exists (no-matter who owns it)", an attacker can use the endpoint to confirm the presence of a `collectionId`.

The controller forwards **any** error returned by `mediaService.DeleteCollection` to the generic `NoContent` helper. otherwise it returns **204 No Content**.

```
[...]
func (c MediaController) DeleteCollection([...], r *http.Request) ([...]) {
    session := auth.GetSession(r)
    id := snowflake.IDFromString(r.URL.Query().Get("id"))
    err := db.Tx(r.Context(), c.Db, func(tx *db.Db) error {
        return c.mediaService.DeleteCollection([...], id, [...])
    })
    if err != nil {
        err = fmt.Errorf("could not delete collection: %w", err)
        c.Log(r).Error(err.Error(), "id", id)
    } else {
        c.Log(r).Info("deleted collection", "id", id)
    }
    return NoContent(err)
}
[...]
```

server/pkg/controller/media.go:347-361

If `FindCollectionById` cannot locate the row, the service **returns a** ***Forbidden*** **error** (HTTP 403 after the controller's translation).
If the row is found, the function proceeds to delete the *share* entry. **It never checks that `c.CreatedBy == accountId`**, so the request succeeds even when the caller is not the owner.

```
func (s Service) DeleteCollection([...], collectionId snowflake.ID, [...]) error {
    c, err := tx.FindCollectionById(ctx, collectionId)
    if err != nil {
        return errors.New(domain.ErrForbidden("Forbidden"))
    }

    err = tx.DeleteCollectionShare(ctx, db.DeleteCollectionShareParams{
        CollectionID: &collectionId,
        CreatedFor:   &accountId,
    })
    if err != nil {
        return err
    }
    [...]
}
```

server/pkg/domain/media/service.go:539-565

Steps to reproduce

1. PoC for CreateShare

Sending Request with **Collection ID** that doesn't exist:

```
POST /api/v1/share HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "collectionId":"2088064593836433401",
    "cryKey":"29y...+OV",
    "cryPassword":"KST...aBU=",
    "accessKey":"zer...yc=",
    "permission":1,
    "validUntil":"202...00Z"
}
```

Response when the **Collection ID** doesn't exist:

```
HTTP/2 500 Internal Server Error
[...]

Internal Server Error
```

Sending Request with **Collection ID** that doesn't belong to the user account:

```
POST /api/v1/share HTTP/2
Host: app.test.zeitkapsl.at
[...]

{
    "collectionId":"2087871897292560384",
    "cryKey":"29y...+OV",
    "cryPassword":"KST9...aBU=",
    "accessKey":"zer...c=",
    "permission":1,
    "validUntil":"202...00Z"
}
```

Response when the **Collection ID** doesn't belong to the current user account:

```
HTTP/2 403 Forbidden
[...]

Forbidden
```

2. PoC for DeleteCollection

Sending Request with **Collection ID** that doesn't exist:

```
DELETE /api/v1/collection?id=2088064593836433401 HTTP/2
Host: app.test.zeitkapsl.at
[...]

{}
```

Response when the **Collection ID** doesn't exist:

```
HTTP/2 403 Forbidden
[...]

Forbidden
```

Sending Request with **Collection ID** that doesn't belong to the user account:

```
DELETE /api/v1/collection?id=2087871897292560384 HTTP/2
Host: app.test.zeitkapsl.at
[...]

{}
```

Response when the **Collection ID** doesn't belong to the current user account:

```
HTTP/2 204 No Content
[...]
```

Remediation Recommendation

- Normalize the HTTP response (always return the same status code, e.g., 204, regardless of whether the media ID exists or 500 for generic error messages).
- Wrap low-level "row not found" errors in a domain error and map them to a fixed status (e.g., 404 or 403) or swallow them.
- Add an **explicit ownership/ACL check** before processing media-IDs or collection-IDs.
- Implement rate-limiting or request throttling to prevent bulk probing.

A Appendix

A.1 Disclaimer

The audit was conducted in accordance with current best practices and standards, as thoroughly as possible within the time-limited scope and reliance on information provided by zeitkapsl reasearch OG. Due to the nature of information security and the complexity and continuous evolution of information systems, complete security cannot be guaranteed. The auditor cannot guarantee the completeness of the identified vulnerabilities and recommended countermeasures, nor the security of the reviewed information systems.

A.2 Imprint

PWND Labs GmbH

Business purpose: Consulting and services in information technology, particularly in the field of offensive cybersecurity; software development and programming services, especially in the area of offensive cybersecurity, and trade with goods of all kinds.

VAT Identification Number: ATU82082657
Company Register Number: 651929W
Company Register Court: Regional Court of Klagenfurt
Company Headquarters: Klagenfurt
Address: Nestroygasse 3, 9020 Klagenfurt
Memberships: WKO
Applicable Legal Regulations: www.ris.bka.gv.at

Authorized Representatives:

- Felix Slama (Managing Director)
- Jonas Heschl (Managing Director)

Contact Information:

- **Phone:** +43 677 6145 2158
- **Email:** contact@pwnd.at

A.3 List of Figures

Figure 1 - Distribution of Identified Vulnerabilities 5
Figure 2 - Key Hierarchy 43
Figure 3 - Upload Protocol 44
Figure 4 - Integrity Discussion 45
Figure 5 - Password Change 46
Figure 6 - Old Password Leak 47
Figure 7 - Ciphertext 48
Figure 8 - Phase Corruption 49
Figure 9 - Outdated Dependency Hierarchy 61
Figure 10 - Outdated Dependency Hierarchy 79
Figure 11 - Outdated Dependency Hierarchy 80
Figure 12 - Outdated Dependency Hierarchy 81

A.4 List of Tables

Table 1 - Document Details 2
Table 2 - Audit Team 2
Table 3 - List of Changes 2
Table 4 - Identified Vulnerabilities 6
Table 5 - CVSSv4 Score 8