the BLOCKCHAIN team

SMART CONTRACT SECURITY ASSESSMENT

Waldrop Protocol Move Contracts

Decentralized ETL on Sui · Walrus storage settlement layer

0	3	8	10	6
CRITICAL	HIGH	MEDIUM	LOW	INFO

Audited by **The Blockchain Team** · https://theblockchain.team

PROJECT	waldrop-contracts - `waldrop-labs/waldrop-contracts`
COMMIT	`d889b0c` – "fix: correct billing period epoch units and default" (branch `dev`)
SCOPE	7 Move modules, 3,720 lines of source
TOOLCHAIN	Sui CLI 1.76.0 · Move 2024 edition
REPORT DATE	14 August 2026
STATUS	Final - pre-deployment review

Contents

1 Executive Summary

Waldrop is a decentralized ETL platform that moves data from conventional sources onto Walrus storage, settling subscriptions, quotas and access control on Sui. This assessment covered the seven Move modules that implement that settlement layer: capability and configuration management, subscription lifecycle, treasury, migration job tracking, blob bookkeeping with SEAL access control, provisioning, and Object Display registration.

The codebase is in good shape structurally. Capability gating is applied consistently and correctly - every privileged mutation takes `&AdminCap`, and no authorization bypass of the admin surface was found. The `seal_approve` callback correctly binds the supplied SEAL identity to the store's object ID and extracts the per-blob marker from the identity rather than trusting a caller-supplied argument, which closes the two most common failure modes in SEAL integrations. Arithmetic is guarded, saturating subtraction is used where drift is possible, and unbounded iteration has been deliberately pushed out of the transfer path into owner-driven batch clears. The comments are unusually candid about past mistakes and the reasoning behind current invariants.

No critical issues were found. There is no path by which an attacker drains the treasury, forges an `AdminCap`, decrypts a blob they were never granted access to, or takes over another user's objects.

The weakness that runs through the findings is different in character: **the contract's resource metering does not hold.** Waldrop sells storage, and storage costs the protocol real money on Walrus. Three independent High-severity paths let a user consume substantially more storage than the plan they paid for permits:

- **Concurrent migration jobs are admitted against a counter that is only written at completion** (H-01). A Pro subscriber can open ten 200 GiB jobs, each of which passes the quota check against an account showing zero bytes, then complete all ten - crediting 2 TiB against a 200 GiB plan.
- **Blob registration never checks that the subscription is still live** (H-02). A subscriber can stop paying and keep writing at their old tier, with SEAL encryption, forever. `migration::create_job` checks `is_active`; `storage::register_blob` does not, and the two disagree.
- **`extend_blob` caps the increment rather than the resulting lifetime** (H-03), so any blob can be pushed to an unbounded Walrus expiry one epoch at a time, on the Free tier.

The Medium findings compound this. Plan file-count caps are counted per `BlobStore` while stores are free and unlimited to create (M-02); `Subscription` objects are not bound to the transaction sender in `extend_blob` or `create_job`, permitting a single paid subscription to be rented across wallets (M-01); and the two code paths that credit stored bytes do so independently with no reconciliation, so `total_bytes_stored` - the single number every quota check reads - is not a trustworthy ledger (M-05). Two access-control findings are also worth prompt attention: per-blob share allowlists survive blob deletion and silently reattach if the SEAL marker is reused (M-04), and `transfer_account` never verifies that the `BlobStore` handed to it is the one bound to the account being transferred (M-03).

Every High and Medium finding in this report was reproduced against the compiled package with an executable Move test. The harness is reproduced in Appendix A; all seven proofs pass on commit `d889b0c` using Sui CLI 1.76.0.

> **RECOMMENDATION**
>
> Do not deploy to mainnet before H-01, H-02 and H-03 are fixed. Each is a direct, unprivileged path to consuming protocol-funded Walrus storage well beyond the plan the user paid for, and each is exploitable by a single user acting alone with no special position. M-03 and M-04 should be fixed in the same release - both silently violate an access expectation the UI will present to users as a guarantee.

Separately from the code itself, one deployment-time decision deserves the same weight as the findings above. Waldrop's confidentiality guarantee is enforced entirely by `storage::seal_approve`, and that function is only as durable as the package's upgrade policy. The `UpgradeCap` minted at publish is left at its default and is not referenced anywhere in the codebase, so its holder can publish a version in which `seal_approve` approves everyone - retroactively exposing every blob ever encrypted, since the ciphertext does not change, only the on-chain predicate guarding the key does (M-07). Resolving this is a choice about how to publish, not a code fix, and it should be made before mainnet rather than after.

2 Scope & Methodology

2.1 In scope

All Move sources under `sources/` at commit `d889b0c`, reviewed together with their unit tests, the deployment scripts, and the CI workflow.

MODULE	LOC	RESPONSIBILITY	COVERAGE
`admin.move`	471	`AdminCap` singleton, `GlobalConfig`, user registry, pause & version gating	86.3%
`subscription.move`	1,059	Plan registry, per-user accounts, subscribe / upgrade / downgrade / renew / cancel	46.3%
`storage.move`	1,359	Blob bookkeeping, folders, viewer & per-blob ACLs, `seal_approve`, account transfer	74.0%
`migration.move`	328	Migration job lifecycle: create, complete, release	75.1%
`treasury.move`	225	Multi-coin fee collection, admin withdrawal and refunds	95.1%
`display.move`	229	Object Display V2 registration for user-facing types	5.1%
`provision.move`	49	Composes subscribe + blob store creation into one transaction	0.0%

2.2 Out of scope

The off-chain transform engine, the Walrus and SEAL key-server deployments themselves, the web application and SDK, and the operational security of the keys that will hold `AdminCap`, `Publisher` and `UpgradeCap`. Several findings depend on how off-chain components consume on-chain events; those dependencies are stated explicitly where they arise.

2.3 Method

The review was adversarial and manual, conducted line by line across all seven modules, with findings confirmed empirically rather than argued from inspection alone. Specifically:

- **Capability and authorization tracing.** Every public and entry function was checked for the guard set it should carry: capability parameter, sender binding on each object argument, pause gate, and version gate. Divergence between sibling functions was treated as a finding rather than a style note - H-02 and L-01 were both found this way.
- **Invariant testing.** For each documented invariant (plan storage cap, file cap, epoch ceiling, one account per address, shares do not survive a transfer) an attempt was made to violate it in a test.

- **Object-ability analysis.** Every struct's abilities were checked against the custody model the module assumes. `store` on a struct whose `owner` field is treated as authoritative is a recurring source of the Medium findings.
- **Time-of-check / time-of-use analysis** on every reservation-style flow, which is what surfaced H-01.
- **SEAL identity handling.** The `seal_approve` path was traced end to end, including marker extraction, prefix binding, folder resolution, and every write path that can mutate an allowlist. This path is largely sound; M-04 is its one gap.
- **Proof-of-concept development.** Seven Move tests were written against the compiled package. All findings rated High or Medium carry a passing reproduction.
- **Build and suite verification.** `sui move build` is clean; `sui move test` reports 124 passing tests; coverage was measured with `sui move test --coverage`.

3 Severity Classification

Severity combines impact with the difficulty and privilege required to reach it. Because Waldrop's economic exposure is Walrus storage purchased on users' behalf, quota and metering failures are treated as financial findings, not as business-logic notes.

SEVERITY	DEFINITION
CRITICAL	Direct, unprivileged theft or permanent loss of funds or user data; forgery of the admin capability; decryption of data by a party who was never granted access.
HIGH	A single unprivileged actor causes material, repeatable economic loss to the protocol, or defeats a stated security or billing guarantee. Exploitable today, with no special position and no dependence on another party's mistake.
MEDIUM	Loss or access-control violation that requires a precondition (an object obtained from another party, a specific sequence of operations), or which corrupts state that other guarantees are built on.
LOW	Limited impact, self-inflicted harm, incorrect accounting that favours the protocol, or a defence-in-depth gap not currently reachable.
INFO	Code quality, dead code, documentation drift, operational and process observations, and centralization disclosures.

4 Findings Summary

ID	SEVERITY	TITLE	MODULE	POC
H-01	HIGH	Storage quota bypassed by concurrent migration jobs	`migration`	✓
H-02	HIGH	Expired subscriptions retain full paid-tier write access	`storage`	✓
H-03	HIGH	`extend_blob` caps the increment, not the resulting lifetime	`storage`	✓
M-01	MEDIUM	`Subscription` arguments are not bound to the sender	`storage`, `migration`	✓
M-02	MEDIUM	Per-plan file caps reset by creating a second `BlobStore`	`storage`	✓
M-03	MEDIUM	`transfer_account` accepts an unrelated `BlobStore`	`storage`	✓
M-04	MEDIUM	Per-blob share allowlists survive deletion and reattach on marker reuse	`storage`	✓
M-05	MEDIUM	Stored bytes are credited twice across the migration and storage paths	`migration`, `storage`	-
M-06	MEDIUM	Core objects carry `store`, desynchronizing the user registry	`subscription`, `admin`	-
M-07	MEDIUM	Package upgrade authority can retroactively rewrite SEAL access policy	`storage`, `deploy`	-
M-08	MEDIUM	Jobs at advertised Pro scale can never be completed - `blob_ids` exceeds Sui size limits	`migration`	-
L-01	LOW	Pause and version gates applied inconsistently across entry points	`all`	-
L-02	LOW	Mid-cycle upgrade over-charges on a second upgrade	`subscription`	-
L-03	LOW	`admin_extend_enterprise` ungated and silently revives cancelled subs	`subscription`	-
L-04	LOW	`register_quilt` ignores `max_files_per_job`	`storage`	-
L-05	LOW	User-supplied `cost_usdc_equivalent` pollutes revenue events	`migration`	-
L-06	LOW	`add_blob_share` accepts markers bound to no blob	`storage`	-
L-07	LOW	Unbounded free account and `BlobStore` creation	`storage`, `subscription`	-

L-08	LOW	No revocation path for a compromised `AdminCap`	`admin`	-
L-09	LOW	Bulk share revocation emits no events - mass access changes are unobservable	`storage`	-
L-10	LOW	Deploy script gives mainnet a smaller gas budget than testnet; `UpgradeCap` unhandled	`scripts`	-
I-01	INFO	Centralization: unrestricted admin powers, no timelock, unmanaged `UpgradeCap`	`admin`	-
I-02	INFO	Dead code and orphaned state from the removed loyalty module	`admin`, `subscription`	-
I-03	INFO	`grace_period_epochs` is configured but never enforced	`admin`	-
I-04	INFO	README documents a module that no longer exists and mis-states ownership	`docs`	-
I-05	INFO	CI path filter never matches this repository - the workflow never runs	`CI`	-
I-06	INFO	Test coverage gaps on the subscription, provision and display modules	`tests`	-

5 Detailed Findings

5.1 High

H-01 HIGH Storage quota bypassed by concurrent migration jobs

> **Location:** `sources/migration.move:139–143` (check), `sources/migration.move:219–251` (credit)
> **Impact:** Protocol pays for Walrus storage far beyond the plan sold · **Reproduced:** `poc_concurrent_jobs_bypass_storage_quota`

`create_job` admits a job by checking the declared job size against the account's current usage:

```
// migration.move:139–143 — the admission check
let effective_limit = subscription::effective_storage_limit(account, plan_registry, tier);
assert!(
    subscription::account_total_bytes(account) + total_size_bytes <= effective_limit,
    EStorageLimitExceeded,
);
```

But `account_total_bytes` is only advanced in `complete_job`, and `complete_job` performs no quota check at all - it does not even receive the `PlanRegistry` it would need to perform one:

```
// migration.move:219–249 — the credit, with no re-check
public fun complete_job(
    config: &GlobalConfig,
    account: &mut UserAccount,
    job: &mut MigrationJob,
    /* ... */
) {
    // checks are only against the job's OWN declaration:
    assert!(verified_bytes <= job.total_size_bytes, EOverDeclared);
    // ...
    subscription::add_storage(account, verified_bytes, blob_count, job.encrypted);
    // ^ no plan limit is consulted here
}
```

The admission check is therefore a time-of-check/time-of-use window that stays open for the entire life of the job. Every job created before any job completes measures itself against the same zero baseline. The only bound is `max_concurrent_jobs`, which is 3 on Starter, 10 on Pro, and `0` - meaning unlimited - on Enterprise.

REPRODUCTION

A Pro subscriber (200 GiB limit, 10 concurrent jobs) opens two 200 GiB jobs and completes both:

```
// Both admitted: each sees account_total_bytes == 0.
let mut j1 = migration::create_job(&config, &reg, &mut acct, &sub, /* .. */ 200 * GB, /* .. */);
let mut j2 = migration::create_job(&config, &reg, &mut acct, &sub, /* .. */ 200 * GB, /* .. */);

migration::complete_job(&config, &mut acct, &mut j1, vector[s(b"b1")], 200 * GB, 10, 0, ctx);
migration::complete_job(&config, &mut acct, &mut j2, vector[s(b"b2")], 200 * GB, 10, 0, ctx);

assert!(subscription::account_total_bytes(&acct) == 400 * GB, 1);  // PASSES
assert!(subscription::account_total_bytes(&acct) > limit, 2);     // PASSES
```

At the 10-job Pro ceiling this is 2 TiB of Walrus storage against a 200 GiB plan - a 10× overrun for the price of one subscription. The effect is not merely a wrong counter: the off-chain engine acts on `JobCompleted`, so the protocol actually purchases the storage.

RECOMMENDATION

Reserve the budget at admission rather than at completion. Increment a `reserved_bytes` field on `UserAccount` inside `create_job`, check `total_bytes_stored + reserved_bytes + total_size_bytes <= effective_limit`, and release the reservation in both `complete_job` (converting reserved to stored) and `release_job` (discarding it). This makes the invariant hold regardless of how many jobs are open. As defence in depth, pass `plan_registry` and `sub` into `complete_job` and re-assert the limit before crediting.

H-02 HIGH Expired subscriptions retain full paid-tier write access

> **Location:** `sources/storage.move:356–444` (`register_blob`), `sources/storage.move:452–557` (`register_quilt`)
> **Impact:** Subscribers stop paying and keep writing at their old tier indefinitely · **Reproduced:** `poc_expired_subscription_still_registers_blobs`

Sui has no epoch-tick callback, so an expired subscription remains `STATUS_ACTIVE` forever - the contract acknowledges this explicitly at `subscription.move:529`. Liveness therefore has to be evaluated at every use site via `is_active`, which compares `expires_epoch` against the current epoch.

Three of the four gated paths do this. `migration::create_job` checks it (`migration.move:134`), and so does `storage::extend_blob` (`storage.move:576`). But the two functions that actually admit new data - `register_blob` and `register_quilt` - do not:

```
// storage.move:366-377 — the full guard set of register_blob
admin::assert_not_paused(config);
admin::assert_version(config);
assert!(ctx.sender() == store.owner, ENotOwner);
assert!(subscription::sub_owner(sub) == ctx.sender(), ENotOwner);
assert!(subscription::account_owner(account) == ctx.sender(), ENotOwner);
assert!(!store.blobs.contains(blob_ref.blob_id), EDuplicateBlob);

let tier = subscription::sub_tier(sub);       // tier read from a possibly-dead sub
let plan = subscription::get_plan(plan_registry, tier);

// MISSING: assert!(subscription::is_active(sub, ctx.epoch()), ESubscriptionNotActive);
```

Ownership is verified; liveness is not. The tier is then read straight off the dead subscription and used to size the storage limit, to select the file-count cap, and to decide whether SEAL encryption is permitted. A user subscribes to Pro for one month and retains Pro storage limits and Pro-only encrypted uploads permanently.

REPRODUCTION

```
// Subscribe Pro (30-epoch period), then roll 100 epochs forward.
assert!(!subscription::is_active(&sub, ctx.epoch()), 0);   // sub is definitively dead

// Encrypted upload — a paid-plan-only feature.
let br = storage::new_blob_ref(s(b"after-expiry"), 1024, 100, 130, true, /* .. */);
storage::register_blob(&mut store, &config, &reg, &mut acct, &sub, br, option::none(), ctx);
assert!(storage::store_total_blobs(&store) == 1, 1);       // PASSES
```

RECOMMENDATION

Add `assert!(subscription::is_active(sub, ctx.epoch()), ESubscriptionNotActive)` to both `register_blob` and `register_quilt`, alongside the existing owner checks. If a grace period is intended before writes are cut off - and `GlobalConfig.grace_period_epochs` suggests it is, see I-03 - then implement it explicitly here rather than leaving the check out entirely.

H-03 HIGH `extend_blob` caps the increment, not the resulting lifetime

> **Location:** `sources/storage.move:562–603`
> **Impact:** Any user extends any blob to an unbounded Walrus expiry · **Reproduced:** `poc_extend_blob_repeats_past_plan_ceiling`

The function's own documentation states the intended rule - and the code implements a different one:

```
/// Extend a blob's expiry. The new lifetime (from now) must fit within
/// the caller's plan's `max_walrus_epochs` ...

// storage.move:582–595 — what is actually enforced
let max_allowed = if (plan_max_epochs > 0 && plan_max_epochs < global_max_epochs) {
    plan_max_epochs
} else { global_max_epochs };

let blob = store.blobs.borrow_mut(blob_id);
assert!(additional_epochs <= max_allowed, EEpochsExceedPlanMax);  // bounds the STEP

let old_expiry = blob.expiry_epoch;
let new_expiry = old_expiry + additional_epochs;              // unbounded accumulation
blob.expiry_epoch = new_expiry;
```

Nothing constrains `new_expiry` itself, and the function may be called any number of times. The check bounds each step but not the sum, so the ceiling is not a ceiling - it is a rate limit of one call per transaction. The prior release removed an expiry assertion here (commit `2c4331a`, "drop bogus extend_blob expiry assertion"); the replacement bound was never added.

The cheapest instance is the Free tier, whose `max_walrus_epochs` is 1. A Free user registers a blob with a one-epoch lifetime and then calls `extend_blob` with `additional_epochs = 1` as many times as they like:

```
let mut i = 0;
while (i < 10) {
    storage::extend_blob(&mut store, &config, &reg, &sub, s(b"blob1"), 1, ctx);
    i = i + 1;
};
assert!(storage::blob_expiry_epoch(blob) == 11, 0);  // PASSES — 11x the plan ceiling of 1
```

Each Walrus epoch is 14 days on mainnet, so the plan's one-epoch Free allowance becomes years of protocol-funded storage. The global `max_storage_epochs` ceiling does not help: it is folded into `max_allowed` and so also bounds only the step.

RECOMMENDATION

Assert on the resulting lifetime measured from now, which is what the doc comment already promises:

```
let new_expiry = old_expiry + additional_epochs;
assert!(new_expiry > current_epoch, EInvalidEpoch);
assert!(new_expiry - current_epoch <= max_allowed, EEpochsExceedPlanMax);
```

Note that `max_allowed` already correctly takes the minimum of the plan and global ceilings, so no change is needed there.

5.2 Medium

M-01 MEDIUM `Subscription` arguments are not bound to the sender

Location: `sources/storage.move:562–576`, `sources/migration.move:114–177`
Impact: One paid subscription rented across many wallets · **Reproduced:** `poc_extend_blob_accepts_foreign_subscription`

`register_blob` and `register_quilt` bind every object argument to the sender:

```
assert!(subscription::sub_owner(sub) == ctx.sender(), ENotOwner);
assert!(subscription::account_owner(account) == ctx.sender(), ENotOwner);
```

`extend_blob` and `migration::create_job` do not. `extend_blob` checks only `ctx.sender() == store.owner`, then reads the tier from whatever `Subscription` it was handed. `create_job` checks neither `sub_owner` nor `account_owner`, and does not verify that the account and the subscription belong to the same user.

Sui's ownership rules mean the caller must genuinely own the object to pass it, so this is not a theft primitive. It is a licensing one. `Subscription` has the `store` ability, so it can be moved between wallets with a plain `public_transfer`. A single Pro subscription can therefore be passed around a group, each holder using it in turn to obtain Pro epoch ceilings, Pro file caps and Pro SEAL entitlement on their own store and their own account - while `sub.owner` still names someone else entirely.

```
assert!(subscription::sub_owner(&pro) == USER, 0);   // NOT the caller
assert!(subscription::sub_tier(&pro) == 2, 1);       // Pro

// ATTACKER is on Free (max_walrus_epochs = 1). Pro allows 53.
storage::extend_blob(&mut store, &config, &reg, &pro, s(b"blob1"), 53, ctx);
assert!(storage::blob_expiry_epoch(blob) == 54, 2);   // PASSES
```

RECOMMENDATION

Add the same two assertions `register_blob` already carries to `extend_blob`, `create_job`, `complete_job` and `release_job`. Consider a single shared helper - `assert_caller_owns(account, sub, ctx)` - so the guard set cannot drift again between sibling functions. Also assert that the `MigrationJob` passed to `complete_job` belongs to the same account whose counters it mutates.

M-02 MEDIUM Per-plan file caps reset by creating a second `BlobStore`

> **Location:** `sources/storage.move:316–348`, `sources/storage.move:388–391`
> **Impact:** `max_total_files` is unenforceable · **Reproduced:** `poc_max_total_files_reset_by_second_store`

Byte quotas are checked against the `UserAccount`, which is per-user - correct. But the file-count cap is checked against the `BlobStore`:

```
// storage.move:388-391
let max_total = subscription::plan_max_total_files(plan);
if (max_total > 0) {
    assert!(store.total_blobs + 1 <= max_total, ETotalFilesExceeded);
};
```

and `create_blob_store` / `create_shared_blob_store` are `public`, require no capability, no subscription and no payment, carry no pause or version gate, and are subject to no per-user limit. Each new store starts at `total_blobs = 0`. `register_blob` accepts any store the sender owns, so the cap resets on demand.

```
storage::register_blob(&mut store_a, /* .. */);          // store A now at its 1-file cap
let mut store_b = storage::create_blob_store(ctx);       // free, unlimited, starts at 0
storage::register_blob(&mut store_b, /* .. */);          // accepted
assert!(subscription::account_total_blobs(&acct) == 2, 1);  // PASSES under a 1-file plan
```

The transfer-path comment at `storage.move:1148` states the intended model: "*Quotas are gated on Subscription, so owning multiple BlobStores is safe.*" That holds for bytes but not for file counts, because `total_blobs` lives on the store rather than the account.

RECOMMENDATION

Check the file cap against `subscription::account_total_blobs(account)`, which the account already tracks and which `add_storage` / `release_storage` already maintain. Applies to `register_quilt` (`storage.move:479–482`) identically. Separately, consider gating store creation on an active subscription so the per-user object count is bounded (see L-07).

M-03 MEDIUM `transfer_account` accepts an unrelated `BlobStore`

> **Location:** `sources/storage.move:1153–1205`
> **Impact:** An account can be "transferred" while the seller keeps the data · **Reproduced:** `poc_transfer_account_with_unrelated_store`

`transfer_account` is the sanctioned path for handing an account to a new owner. It moves the `UserAccount` and `Subscription` by value, repoints the shared `BlobStore`'s owner, clears the viewer set, and requires that per-blob and per-folder shares have already been drained. All three objects are checked to be owned by the sender:

```
assert!(subscription::account_owner(&user_account) == sender, ENotOwner);
assert!(subscription::sub_owner(&subscription) == sender, ENotOwner);
assert!(blob_store.owner == sender, ENotOwner);
```

What is never checked is that `blob_store` is *the* store bound to `user_account`. The binding exists - `UserAccount.blob_store_id` was added in commit `5dd1a90` precisely so the app can resolve a user's store - but `transfer_account` does not consult it.

Since stores are free to create (M-02), a seller can create a throwaway store in the same transaction and pass that instead. The buyer receives the account and subscription; the seller keeps the store that actually holds the blobs, still owned by them, still readable by them, and still bearing whatever shares they had granted:

```
let mut decoy = storage::create_blob_store(ctx);   // created in this very tx
storage::transfer_account(&mut config, acct, sub, &mut decoy, ATTACKER, ctx);

assert!(storage::store_owner(&decoy) == ATTACKER, 0);       // decoy handed over
assert!(storage::store_owner(&real_store) == USER, 1);      // real store untouched
```

The share-clearing preconditions are also evaluated against the decoy, so they pass vacuously - the whole point of `ESharesNotCleared` and `EFolderViewersNotCleared` is defeated. Worse, the account's `blob_store_id` still points at the seller's store, so the buyer's client will resolve to an object they do not own.

RECOMMENDATION

Assert the binding before any mutation:

```
let bound = subscription::account_blob_store_id(&user_account);
assert!(option::is_some(&bound), EStoreNotBound);
assert!(*option::borrow(&bound) == object::id(blob_store), EWrongBlobStore);
```

Legacy accounts created before `blob_store_id` existed carry `None`; decide explicitly whether to reject them or to allow a one-time admin-assisted binding, rather than letting `None` fall through.

M-04 MEDIUM Per-blob share allowlists survive deletion and reattach on marker reuse

> **Location:** `sources/storage.move:607–654` (`release_blob`), `sources/storage.move:405–410` (uniqueness check)
> **Impact:** Revoked-by-deletion access silently restored on a new blob · **Reproduced:** `poc_stale_share_survives_blob_release`

`release_blob` cleans up the marker index but not the allowlist keyed by the same marker:

```
// storage.move:637–642 — the only marker cleanup performed
if (option::is_some(&blob.seal_marker)) {
    let marker = *option::borrow(&blob.seal_marker);
    if (store.marker_folder.contains(marker)) {
        store.marker_folder.remove(marker);      // index freed
    };
};
// store.shares[marker] and store.share_markers are left untouched.
```

`register_blob` rejects a duplicate marker by consulting `marker_folder` only (`storage.move:405–410`). Because `release_blob` has just cleared that entry, the marker is considered free - while `shares[marker]`, holding the previous blob's viewer allowlist, is still populated. A new blob registered under that marker inherits the old blob's viewers, and `seal_approve` will authorize them (`storage.move:728–733`).

```
storage::add_blob_share(&mut store, &config, marker, ATTACKER, ctx);
let _dropped = storage::release_blob(&mut store, &mut acct, s(b"secret-v1"), ctx);

assert!(!storage::marker_in_use(&store, marker), 1);                    // marker freed
assert!(storage::store_blob_share_contains(&store, marker, ATTACKER), 2); // grant survives

// A new, unrelated blob reuses the freed marker.
storage::register_blob(&mut store, /* .. */ option::some(marker) /* .. */);
assert!(storage::store_blob_share_contains(&store, marker, ATTACKER), 4); // silently granted
```

Markers are 16 random bytes supplied by the client, so collision by chance is negligible. The realistic trigger is deliberate or defective reuse by the client - re-uploading a file after an edit and reusing its marker is a natural implementation choice, and the contract gives no signal that doing so restores old grants. Deleting a shared file is the action a user would reasonably expect to revoke sharing; here it leaves the grant in place and arms it for the next blob. It also leaves `share_markers` permanently non-empty, which blocks `transfer_account` until `clear_blob_shares` is called for a blob that no longer exists.

RECOMMENDATION

Drop the allowlist in `release_blob` alongside the index entry:

```
if (table::contains(&store.shares, marker)) {
    let allowlist = table::remove(&mut store.shares, marker);
    let _ = vec_set::into_keys(allowlist);
    let (found, idx) = vector::index_of(&store.share_markers, &marker);
    if (found) { vector::remove(&mut store.share_markers, idx); };
}
```

The `shares` entry for one marker is bounded by `max_viewers_per_store` , so removing it inline is safe and does not reintroduce the unbounded-iteration problem that motivated `clear_blob_shares` .

M-05 MEDIUM Stored bytes are credited twice across the migration and storage paths

> **Location:** `sources/migration.move:249`, `sources/storage.move:434`, `sources/migration.move:293–303`
> **Impact:** The quota ledger every limit check reads is unreliable in both directions

Two independent code paths credit the same underlying data to the same counter:

```
// migration.move:249 — on job completion
subscription::add_storage(account, verified_bytes, blob_count, job.encrypted);

// storage.move:434 — on blob registration
subscription::add_storage(account, size_bytes, 1, encrypted);
```

In the documented flow these describe the same bytes: a migration job uploads files to Walrus, and those blobs are then registered in the user's `BlobStore`. Nothing cross-references the two - `complete_job` does not know which blobs will be registered, and `register_blob` does not know they came from a job. The account is charged twice.

`release_job` then refunds the migration-side credit for a `COMPLETED` job (`migration.move:293–303`), without touching the blobs. A user who completes a job and then releases it returns `total_bytes_stored` to its prior value while the blobs remain live on Walrus and registered in the store.

The consequence is that `total_bytes_stored` - the single number consulted by every storage-limit check in `register_blob`, `register_quilt` and `create_job` - can be either too high (double-counted, denying the user storage they paid for) or too low (refunded while the data persists, granting storage they did not). Combined with H-01, the counter cannot be relied on as a quota ledger at all.

RECOMMENDATION

Choose one authoritative ledger. The cleanest model is to make blob registration the sole writer of `total_bytes_stored`, and have the migration module track only the reservation described in H-01 - `complete_job` would release the reservation without crediting bytes, leaving the subsequent `register_blob` / `register_quilt` calls to record actual usage. If jobs must credit storage directly, then `register_blob` needs to recognise blobs already accounted for by a job and skip the second credit.

M-06 MEDIUM Core objects carry `store`, desynchronizing the user registry

> **Location:** `sources/subscription.move:111, 132`, `sources/migration.move:52`, `sources/admin.move:83`
> **Impact:** Registry, `owner` fields and actual custody diverge permanently

`UserAccount`, `Subscription` and `MigrationJob` all declare `has key, store`. The `store` ability makes them freely transferable with `transfer::public_transfer` by any holder, entirely outside `transfer_account`.

Three pieces of state then disagree about who owns an account: `GlobalConfig.user_accounts` (address → account ID), the `owner` field on the object itself, and actual Sui custody. A raw transfer updates only the third. The consequences are self-reinforcing:

- The recipient holds a `UserAccount` whose `owner` is someone else, so every `account_owner(account) == ctx.sender()` check fails - the object is inert in their hands.
- The recipient cannot subscribe afresh to obtain a working account, because they now hold one - and if they were already registered, `reassign_user_account` would reject them anyway (`ERecipientAlreadyRegistered`).
- The original owner cannot subscribe again either: `register_user` aborts with `EUserAlreadyRegistered` and there is no reachable way to clear the entry - `deregister_user` exists but is `public(package)` and has no caller anywhere in the codebase (I-02).

The result is an unrecoverable state for both parties, reachable by one ordinary transfer that the contract cannot prevent or observe.

RECOMMENDATION

Drop the `store` ability from `UserAccount` and `Subscription` so `transfer_account` becomes the only transfer path. Both are currently moved with `public_transfer` at `subscription.move:627–628`, `storage.move:1203–1204` and `provision.move:46–47`; switch those to `transfer::transfer`, which works on `key`-only types from within the defining module. Note this also removes the M-01 rental primitive at its root. If `store` must be retained for composability, add an admin-gated recovery function that repairs a desynchronized registry entry, and wire up the existing `deregister_user`.

M-07 MEDIUM Package upgrade authority can retroactively rewrite SEAL access policy

> **Location:** `sources/storage.move:709–742`, package `UpgradeCap` (created at publish, not managed in code)
> **Impact:** Whoever controls upgrades can grant themselves decryption of every user's encrypted blobs, including data uploaded before the upgrade

All confidentiality in Waldrop rests on `storage::seal_approve`. SEAL key servers release a decryption key only if that function returns without aborting, so the Move code *is* the access policy. The implementation is correct (section 1), but it is not immutable.

The `UpgradeCap` minted at publish is transferred to the deployer by the Sui framework and is never referenced, restricted, wrapped or made immutable anywhere in this package. Its holder can publish a version 2 in which `seal_approve` is:

```
entry fun seal_approve(id: vector<u8>, store: &BlobStore, ctx: &TxContext) {
    return  // approves everyone, for every blob, in every store
}
```

Key servers evaluate the policy at the current package version, so this applies **retroactively**: blobs encrypted months earlier under the honest policy become decryptable, because the ciphertext never changes - only the on-chain predicate guarding the key does. No user action, notification or consent is involved, and no event is emitted. A user who audited the package before uploading has no ongoing guarantee.

This is materially different from the other admin powers in section 6. Those are *disclosed* capabilities over funds and configuration that users can reason about. This one silently converts an end-to-end encryption guarantee - the product's core promise - into a trust-the-operator arrangement, and it is invisible from the contract source because the risk lives in the upgrade policy rather than in any function.

Note this compounds M-06's recommendation: a v2 could equally re-add abilities, drop owner checks, or mint capabilities. The upgrade path is the root authority over every other guarantee in this report.

RECOMMENDATION

Decide the upgrade policy explicitly and publish that decision, rather than leaving the default in place:

- **Preferred:** split `seal_approve` and the types it reads into a separate package and call `make_immutable` on that package's `UpgradeCap`. The mutable business logic can keep evolving while the access policy becomes unforgeable. This is the only option that removes the trust assumption rather than narrowing it.
- **Otherwise:** place the `UpgradeCap` behind a multisig and a timelock via a custom upgrade policy, so an upgrade is visible on-chain for a fixed window before it can take effect and users can withdraw. Document the delay publicly.
- State the chosen policy in `README.md` next to the encryption claims, and record the `UpgradeCap` object ID so users can verify the policy themselves.

M-08 MEDIUM Jobs at advertised Pro scale can never be completed

Location: `sources/migration.move:66` (`blob_ids: vector<String>`), `sources/migration.move:236, 241`
Impact: A paid-tier migration at the documented file limit aborts and the job is stranded; the plan advertises a capacity the contract cannot honour

`complete_job` stores the full list of resulting blob IDs on the job object, bounded only by the job's own declaration:

```
// migration.move:236 — the only bound on the vector
assert!(blob_ids.length() <= job.file_count, EOverDeclared);
// ...
job.blob_ids = blob_ids;   // written into the MigrationJob object
```

`job.file_count` is itself checked against `plan_max_files` at creation - but only when that value is non-zero, and `0` means unlimited in this contract:

```
// migration.move:145–148
let max_files = subscription::plan_max_files(plan);
if (max_files > 0) {                // Pro and Enterprise set max_files_per_job = 0
    assert!(file_count <= max_files, EFileLimitExceeded);
};
```

So on Pro and Enterprise, `file_count` is unbounded and the blob-ID vector inherits that. Sui imposes two hard ceilings that the contract never accounts for: a Move object may not exceed 256 KB, and transaction inputs are limited to roughly 128 KB. A Walrus blob ID is a base64-encoded 32-byte hash - about 45 bytes once `String` overhead is counted - so the practical ceiling is on the order of a few thousand IDs, with the transaction-input limit biting first.

The Pro plan advertises `max_total_files: 50_000` (`subscription.move:302`). A user who migrates anywhere near that in a single job cannot complete it: the transaction aborts on size, every retry aborts identically, and the job stays `STATUS_RUNNING` forever. Their only exit is `release_job`, which is defined as an abandon - no bytes were credited, so nothing is refunded, and the off-chain work is wasted. Nothing in `create_job` warns them; the limit is only discovered at completion, after the migration has already run.

This is a correctness and availability defect rather than a security one - there is no attacker, and the damage is confined to the user's own job. It is rated Medium because it makes the headline capability of the most expensive self-serve tier unusable, with no on-chain signal and no recovery path other than discarding the work.

RECOMMENDATION

Bound `file_count` explicitly at `create_job` with a constant that reflects the real object-size ceiling - something like `MAX_BLOB_IDS_PER_JOB: u64 = 2_000` - and assert it regardless of whether the plan limit is "unlimited", so `0` never means "unbounded vector". Large migrations then split naturally across jobs, which the concurrent-job limits already accommodate. If the full list must be retained for jobs of any size, move `blob_ids` out of the object into an emitted event, or into a dynamic field written in bounded batches; the on-chain vector is not read by any function in this package other than for its length.

5.3 Low

L-01 LOW Pause and version gates applied inconsistently across entry points

Location: all modules · **Impact:** Pause is partial; version gating is not a reliable upgrade barrier

Of the roughly forty state-changing public functions, the pause and version guards are applied to a minority, and the pattern does not track a clear rule. `assert_version` is absent from every function in `treasury`, from `set_plan_coin_price`, `remove_plan_coin_price`, `update_plan_limits`, `admin_extend_enterprise`, `cancel`, `request_downgrade` and `cancel_downgrade` in `subscription`, and from `release_blob`, `remove_viewer`, `remove_blob_share`, `clear_blob_shares`, `rename_folder`, `delete_folder`, `remove_folder_viewer`, `clear_folder_viewers`, `create_blob_store` and `move_blob_to_folder` in `storage`. `add_viewer` and `add_blob_share` check pause but not version.

Some omissions are clearly deliberate and correct - revocation must keep working while paused, and the code says so at `storage.move:994–999`. Others are not: `create_blob_store` is a pure state-growth operation that should not be available during an emergency pause, and the plan-registry mutators are exactly the surface that a version gate is meant to freeze during a migration.

Because `assert_version` is the mechanism that halts the old package after an upgrade, every entry point missing it stays live and mutable against a `GlobalConfig` that a newer package version considers stale.

RECOMMENDATION

Classify each function explicitly as grant / revoke / read / admin, then apply the guard set that class requires: grants take pause + version, revocations take version only, admin mutations take version. Record the classification in a comment beside each function so omissions read as intentional. A table in `CONTRIBUTING.md` would make the rule reviewable in PRs.

L-02 LOW Mid-cycle upgrade over-charges on a second upgrade

> **Location:** `sources/subscription.move:417–455` · **Impact:** User pays more than the pro-rata amount; favours the protocol

The pro-rata credit for the unused part of the cycle divides cumulative cycle spend by the full billing period:

```
let remaining_value = if (billing_period > 0) {
    (subscription.amount_paid_in_coin * remaining_epochs) / billing_period
} else { 0 };
// ...
subscription.amount_paid_in_coin = subscription.amount_paid_in_coin + difference;
```

After one upgrade, `amount_paid_in_coin` holds total spend for the cycle, but that spend covered only the epochs remaining at the time it was made - not the whole period. Dividing by `billing_period` therefore understates the per-epoch rate on any subsequent upgrade.

Worked example with a 30-epoch period, Starter at 10 and Pro at 30. A Free user upgrades to Starter at epoch 10: credit 0, cost `10×20/30 = 6`, so `amount_paid_in_coin = 6`. At epoch 20 they upgrade to Pro: the credit computed is `6×10/30 = 2`, but the payment of 6 bought 20 epochs of Starter, so 10 unused epochs are worth 3. The user is under-credited by 1 and pays 14 for a cycle worth 13.33.

The direction favours the protocol, and a single upgrade per cycle - the common case - is exact. Only the second and later mid-cycle upgrades drift.

RECOMMENDATION

Track the span the current payment covers rather than assuming it is the full billing period. Storing `paid_from_epoch` alongside `amount_paid_in_coin` and dividing by `expires_epoch − paid_from_epoch` makes every upgrade exact. Alternatively set `amount_paid_in_coin = new_price_full` on upgrade and document the field as "current tier list price for this cycle".

L-03 LOW `admin_extend_enterprise` ungated and silently revives cancelled subscriptions

> **Location:** `sources/subscription.move:633–655` · **Impact:** Works while paused; reactivates a subscription the user cancelled

Unlike `admin_activate_enterprise`, which calls both `assert_not_paused` and `assert_version`, `admin_extend_enterprise` takes neither `GlobalConfig` nor any gate. It remains fully functional during an emergency pause and after a package upgrade that has not yet been migrated.

It also flips `status` back to `STATUS_ACTIVE` unconditionally, so an admin extending a subscription the user had cancelled silently reactivates it with no distinguishing event - the emitted `EnterpriseActivated` does not record that a cancellation was reversed. The same revival behaviour exists in the unused `extend_expiry` (`subscription.move:852–864`).

RECOMMENDATION

Take `&GlobalConfig` and apply the same gates as `admin_activate_enterprise`. Emit a distinct event when a cancelled subscription is reactivated, or require the status to be `STATUS_ACTIVE` and force cancellation reversal through an explicit function.

L-04 LOW `register_quilt` ignores `max_files_per_job`

Location: `sources/storage.move:474–482` · **Impact:** One documented plan limit is unenforced on the quilt path

`register_quilt` enforces `max_files_per_batch` and `max_total_files` but never consults `max_files_per_job`, which `migration::create_job` does enforce (`migration.move:145–148`). On the Free tier the two differ sharply - `max_files_per_job = 5` against `max_files_per_batch = 1` - so it is not obvious which is intended to bind. Confirm whether the omission is deliberate; if the batch cap is meant to be the only quilt-path limit, say so in the function's doc comment, since the plan struct presents both.

L-05 LOW User-supplied `cost_usdc_equivalent` pollutes revenue events

Location: `sources/migration.move:226, 242, 261` · **Impact:** Off-chain analytics can be poisoned by any user

`complete_job` accepts `cost_usdc_equivalent` directly from the caller, writes it to the job object, and emits it in `JobCompleted` without validation. Any user can declare an arbitrary value up to `u64::MAX`. If an indexer or dashboard sums this field as revenue or cost, a single transaction can distort it arbitrarily.

The treasury's own `total_collected_usdc_equiv` is not affected - it is written only by `treasury::deposit` from `public(package)` call sites - so this is confined to job telemetry.

RECOMMENDATION

Either derive the figure on-chain from `verified_bytes` and the plan, or drop the field from the event and compute it off-chain from the trusted inputs already emitted. If it must stay, document it clearly as untrusted, user-declared metadata.

L-06 LOW `add_blob_share` accepts markers bound to no blob

Location: `sources/storage.move:791–822` · **Impact:** Grants can be created before, and outlive, the blob they refer to

`add_blob_share` validates only that the marker is 16 bytes. It does not check `marker_folder` to confirm the marker belongs to a blob in this store, so an owner can pre-authorize a viewer for a marker no blob has claimed yet. Combined with M-04, this means the `shares` table's lifetime is entirely decoupled from the blobs it governs in both directions.

Impact is limited on its own - only the store owner can create grants, and only over their own store - but it removes the contract's ability to tell an operator, or the UI, which grants are live. Requiring `marker_in_use(store, seal_marker)` would make the share table self-consistent and would fix M-04's blocked-transfer symptom as a side effect.

L-07 LOW Unbounded free account and `BlobStore` creation

> **Location:** `sources/storage.move:316–348`, `sources/subscription.move:336–391` · **Impact:** Unbounded growth of a shared object's `Table`; Sybil surface

`GlobalConfig.user_accounts` is a `Table` on a shared object that only ever grows - nothing removes entries, since `deregister_user` is unreachable (I-02). Free-tier subscription costs only gas: `subscribe` with `tier = TIER_FREE` accepts a zero-value coin of any type, and one address may register once. An attacker with *n* addresses adds *n* permanent entries, and each can then create unlimited `BlobStore` objects (M-02), every one of them a shared object with four `Table`s inside.

Sui's storage-fund model means the attacker pays for this, so it is a cost-of-attack question rather than a free denial of service. Still, the growth is permanent and irreversible. Consider gating `create_blob_store` on an active subscription and limiting each account to one store - which `UserAccount.blob_store_id` already implies is the intended model.

L-08 LOW No revocation path for a compromised `AdminCap`

> **Location:** `sources/admin.move:65–67, 152–156` · **Impact:** A leaked capability stays valid for the life of the package

`AdminCap` is a bare capability - a `UID` and nothing else - minted once in `init` and gated on by possession alone. Every privileged function takes `_: &AdminCap` and ignores its identity:

```
public struct AdminCap has key, store { id: UID }

public fun pause(_: &AdminCap, config: &mut GlobalConfig, ctx: &TxContext) { /* ... */ }
```

If the cap is stolen or its custodian is compromised, there is no mechanism to invalidate it. There is no registry of valid capability IDs to check against, no version or epoch field on the capability that privileged functions consult, and no burn function. Because it declares `store`, the thief can freely move it onward. The legitimate operator cannot mint a replacement - `init` runs once at publish and never again - so the only remedy is a package upgrade that changes the gating, which requires the `UpgradeCap` (M-07) and is a live race against an attacker who can drain the treasury in one transaction.

Sui's security guidance is explicit that a revocation path must be designed *before* publication, precisely because the options afterwards are poor. The related weakness is that `AdminCap` carries no binding to the objects it governs: nothing ties it to a particular `GlobalConfig`, `PlanRegistry` or `Treasury`. That is safe today only because `init` guarantees exactly one of each - an implicit invariant worth stating in a comment, since it is what makes the missing ID check acceptable.

RECOMMENDATION

Add an `admin_epoch: u64` to `GlobalConfig` and to `AdminCap`, have every privileged function assert the two match, and provide a rotation function that mints a fresh cap and bumps the config's value - invalidating the old cap without needing to possess it. The check costs one comparison. Pair it with a documented rotation runbook, and prefer multisig custody so a single key compromise does not require rotation at all.

L-09 LOW Bulk share revocation emits no events

> **Location:** `sources/storage.move:865–877`, `sources/storage.move:1111–1134` · **Impact:** Mass access-control changes are invisible to indexers and audit logs

Single-grant changes are fully instrumented: `add_viewer`, `remove_viewer`, `add_blob_share`, `remove_blob_share`, `add_folder_viewer` and `remove_folder_viewer` each emit an event naming the store, the marker or folder, and the affected address.

The two bulk equivalents emit nothing at all. Both mutate the same allowlists and both are `public`:

```
public fun clear_blob_shares(store: &mut BlobStore, max_markers: u64, ctx: &TxContext) {
    assert!(ctx.sender() == store.owner, ENotOwner);
    while (n < max_markers && !vector::is_empty(&store.share_markers)) {
        /* ... drops the entire allowlist for each marker ... */
    };
    // no event::emit anywhere in this function — same in clear_folder_viewers
}
```

This is not a niche path. `transfer_account` *requires* both to have been run to completion (`ESharesNotCleared`, `EFolderViewersNotCleared`), so every account transfer is necessarily preceded by a mass revocation that leaves no on-chain trace. An indexer or off-chain access mirror built from the granular events will continue to believe revoked viewers still have access, and an operator reconstructing an access history from events will see grants that apparently never end.

Sui's guidance is explicit that allowlist updates should emit events precisely so that off-chain services can track membership changes. The granular functions follow it; these two are the exception.

RECOMMENDATION

Emit a summary event from each - for example `BlobSharesCleared { store_id, markers_cleared, remaining, epoch }` and `FolderViewersCleared { store_id, folders_cleared, viewers_removed, epoch }`. A count-based summary keeps the event small while still letting consumers detect that a bulk revocation happened and re-read the store. Emitting per-address events here would be unbounded, which is the situation the batched design exists to avoid.

L-10 LOW Deploy script gives mainnet a smaller gas budget than testnet

Location: `scripts/deploy.sh:15–16`, `README.md:118` · **Impact:** Mainnet publish may abort on gas; `UpgradeCap` custody is never addressed

```
GAS_BUDGET_TESTNET=1000000000   # 1.0 SUI
GAS_BUDGET_MAINNET=200000000    # 0.2 SUI — 5x smaller, on the network that matters
```

The README documents this as deliberate: "*Mainnet prompts for confirmation and uses a smaller gas budget.*" The reasoning is inverted. A gas budget is a ceiling, not a charge - unused gas is refunded - so a lower budget saves nothing and only adds failure risk. Publishing seven modules of this size is not obviously affordable within 0.2 SUI, and a publish that aborts on `InsufficientGas` still consumes gas while leaving `Published.toml` and `Move.lock` in a state the script has already rewritten.

Related, and more consequential: the script extracts `PACKAGE_ID` and the shared object IDs from the publish output but never mentions the `UpgradeCap` or the `AdminCap`. Both land silently in the deployer's hot wallet - the one holding the gas coin the script just used. This is the operational half of M-07 and L-08: the two most security-critical objects in the system are created by this script and it neither reports them nor moves them anywhere safe.

RECOMMENDATION

Set the mainnet budget at or above the testnet value; if the intent was a guard against typos, assert on the *estimated* cost from a dry run instead. Extend the script to extract and print the `UpgradeCap` and `AdminCap` object IDs alongside the package ID, write them to `.deploy/<network>-latest.json`, and end a mainnet run with an explicit checklist: transfer both to their intended custody, apply the chosen upgrade policy (M-07), and record the IDs publicly.

5.4 Informational

I-01 INFO Centralization: unrestricted admin powers, no timelock, unmanaged `UpgradeCap`

Detailed in section 6. Summarized here for completeness: the `AdminCap` holder can withdraw the entire treasury to any address in one transaction, rewrite every plan limit, pause the protocol indefinitely, and mint Enterprise subscriptions - with no timelock, no on-chain multisig requirement, and no two-step handover. The `UpgradeCap` issued at publish is not restricted or managed in code, so the deployer can replace the package logic arbitrarily.

I-02 INFO Dead code and orphaned state from the removed loyalty module

Commit `5dd1a90` removed the loyalty module, but its supporting surface remains. The following have no caller anywhere in `sources/`:

SYMBOL	LOCATION	NOTE
`deregister_user`	`admin.move:403`	`public(package)`, unreachable - see M-06
`extend_expiry`	`subscription.move:852`	`public(package)`, was a loyalty redemption
`add_bonus_storage`	`subscription.move:827`	only writer of `bonus_storage_bytes`, which `effective_storage_limit` still reads
`add_lifetime_points`	`subscription.move:847`	only writer of `lifetime_points_earned`
`calculate_refund`	`treasury.move:190`	pure view; `process_refund` does not use it
`create_shared_blob_store`	`storage.move:345`	documented as the production path, but `provision` uses `create_blob_store`
`points_per_gb` / `update_points_config`	`admin.move:208, 427`	no on-chain consumer

Two comments still reference the deleted module: `admin.move:28–30` describes an inflation invariant "pinned in `loyalty::update_redemption_rates`", and `storage.move:1151` tells the reader that `Token<LOYALTY>` balances are not carried by `transfer_account`. Both should be removed. Note also that `calculate_refund` being unused means refunds are entirely at admin discretion - no pro-rata amount is enforced on-chain.

I-03 INFO `grace_period_epochs` is configured but never enforced

`GlobalConfig` carries four knobs with no on-chain consumer: `grace_period_epochs`, `sponsored_tx_enabled`, `max_sponsored_tx` and `quilt_threshold_bytes`. Each has an admin setter, an event and a getter, but no contract logic reads them.

`quilt_threshold_bytes` is explicitly documented as off-chain-only (`admin.move:305`), which is fine. The others are not. `grace_period_epochs` is described in `admin.move:245` as "epochs after expiry before read-only mode", but no read-only mode exists in the code - and H-02 shows that expiry currently gates nothing at all on the write path. Anyone reading the config would reasonably conclude a grace period is enforced. Either implement it as part of the H-02 fix or annotate these fields as engine configuration, as `quilt_threshold_bytes` is.

I-04 INFO README documents a module that no longer exists and mis-states ownership

`README.md` has drifted from the code in ways that matter to an integrator:

- It documents a `loyalty` module, a `WPT` closed-loop token, and earn/redeem mechanics. None of it exists. The module table, the architecture diagram, the "Loyalty (WPT)" section and the project-structure listing all reference it.
- It states that "Every user gets their own `UserAccount`, `Subscription`, and `BlobStore` (owned objects)". `BlobStore` is a **shared** object - necessarily so, because `seal_approve` is dry-run by viewers who do not own it (`storage.move:113–116`). This is a security-relevant difference: a shared object is world-readable and world-referenceable.
- The Security section claims "All financial state-changing functions outside of subscribe/migrate require `AdminCap`", which is true, but omits that plan limits are also admin-mutable at any time with immediate effect on existing subscribers.
- Deploy instructions give two different function names for the same step (`create_display` and `setup_display`) and the "## Deploy" heading appears twice.

The `provision` module - the actual user entry point - is not documented at all.

I-05 INFO CI path filter never matches this repository

`.github/workflows/contracts-ci.yml` filters on `paths: ["waldrop-contracts/**"]` and sets `working-directory: waldrop-contracts`. But this repository's root *is* `waldrop-contracts` - confirmed by `git rev-parse --show-toplevel` and by the remote `waldrop-labs/waldrop-contracts.git`. No file in the repository has a path beginning `waldrop-contracts/`, so the filter never matches and the workflow never runs. If it were triggered manually it would fail immediately, since the `working-directory` does not exist.

The build, test and coverage gates are therefore not enforced on any push or pull request. The workflow appears to have been copied from a monorepo layout. Remove the `paths` filter and the `working-directory` keys.

I-06 INFO Test coverage gaps on the subscription, provision and display modules

The suite is substantial - 124 tests, all passing, with genuinely adversarial cases around folders, shares and account transfer. Coverage is uneven, and the gaps line up with where findings were located:

MODULE	COVERAGE	UNTESTED AREAS RELEVANT TO THIS REPORT
`provision`	0.0%	The only `entry` user path. No test exercises it at all.
`display`	5.1%	Display registration; low risk, but `setup_display` is a one-shot deploy step worth a smoke test.
`subscription`	46.3%	Upgrade pro-rata maths (L-02), downgrade/renew interaction, Enterprise extension (L-03).
`storage`	74.0%	Repeated `extend_blob` (H-03), multi-store behaviour (M-02), share lifetime across release (M-04).
`migration`	75.1%	Concurrent job admission (H-01), double-credit interaction with storage (M-05).
`Overall: 65.9%`		

Every High and Medium finding in this report is a test that did not exist. The seven proofs in Appendix A are written against the public API and can be inverted into regression tests once the fixes land - each should flip from passing to aborting with the appropriate error code.

6 Centralization & Trust Assumptions

Waldrop is an administered service, not a trust-minimized protocol, and the code is honest about this. The disclosures below are not defects - they are the trust model, and users and integrators should be able to see it stated plainly.

6.1 What the `AdminCap` holder can do

POWER	FUNCTION	CONSTRAINT
Withdraw the entire treasury to any address	`treasury::withdraw`	Recipient must be non-zero. No cap, no timelock, no delay.
Send treasury funds to any address as a "refund"	`treasury::process_refund`	Functionally identical to `withdraw`; differs only in event name.
Rewrite any plan's limits and features	`subscription::update_plan_limits`	Immediate effect on existing subscribers. Setting a limit to `0` means *unlimited*, so a typo grants unlimited storage.
Set or remove the accepted coins and prices per tier	`subscription::set_plan_coin_price`	Removing every coin for a tier makes it unpurchasable and unrenewable.
Pause all user operations indefinitely	`admin::pause`	No maximum duration. Revocation paths intentionally remain available.
Change the billing period for everyone	`admin::update_storage_limits`	Applies at next renewal; existing cycles keep their stored period.
Mint and extend Enterprise subscriptions	`subscription::admin_activate_enterprise`	No payment. Enterprise storage is unlimited.

6.2 Key custody and recovery

- `AdminCap` declares `has key, store`, so it is freely transferable. There is no two-step handover, no `AdminCapTransferred` event, and no way to mint a replacement. Losing it permanently freezes the treasury, plan configuration and the pause switch. The README says as much; the contract offers no mitigation.
- No on-chain multisig or threshold requirement is enforced. "Hold in multisig on mainnet" (`admin.move:64`) is an operational instruction the code cannot check.
- The `UpgradeCap` created at publish is left at its default policy and transferred to the deployer implicitly. No module restricts, wraps or burns it, so the holder can replace all package logic. If immutability or a delay is intended, that must be established explicitly after publish - the deploy script does not do it.

- `Publisher` also has `store` and controls Display metadata for every Waldrop type. This is correctly called out at `display.move:20–22`.

6.3 Off-chain dependencies

Several on-chain values are declarations the contract cannot verify: the transform engine decides what was actually uploaded to Walrus and the user declares `verified_bytes`, `verified_files` and `blob_ids` to `complete_job`. The contract's only defence is the ceiling declared at `create_job`, which H-01 shows is not binding in aggregate. Similarly, `content_hash` is stored but never checked against anything on-chain. This is a reasonable design for an ETL service, but it means the on-chain record is an attestation by the user, not a proof.

6.4 Version gating

`assert_version` checks `config.version >= CURRENT_VERSION`, and `migrate_version` requires the new value to equal the compiled `CURRENT_VERSION` and to strictly increase. This is the correct pattern - after an upgrade every gated function halts until the admin migrates. Two operational consequences follow: the migration transaction is a mandatory, un-skippable step in any upgrade runbook, and the functions listed in L-01 that omit the gate will keep operating throughout the window when everything else is halted.

7 Test Suite Assessment

Verified against commit `d889b0c` with Sui CLI 1.76.0:

```
$ sui move build
BUILDING waldrop                                        ✓ clean, no warnings

$ sui move test
Test result: OK. Total tests: 124; passed: 124; failed: 0   ✓

$ sui move test --coverage && sui move coverage summary
% Move Coverage: 65.89
```

STRENGTHS

- Negative tests are used properly - `#[expected_failure]` cases cover the abort paths, not just the happy paths.
- The folder, share and marker-index logic is well exercised, including the subtle cases: moving a blob in and out of a shared folder grants and revokes access, repeated moves do not drift counters, and markers stay claimed while ungrouped.
- `transfer_account` has thorough coverage of its preconditions - uncleared shares, uncleared folder viewers, active jobs, self-transfer, and an already-registered recipient.
- Pause semantics are tested in both directions on the folder path, confirming that granting is blocked while revoking is not.

GAPS

- Every test operates on a single `BlobStore` and a single subscription per user. The multi-object cases - two stores, a borrowed subscription, an unrelated store at transfer time - are where M-01, M-02 and M-03 live.
- No test calls the same function twice to check that a limit holds in aggregate. H-01 and H-03 are both single-call-passes, repeated-call-fails.
- No test advances the epoch past expiry and then attempts a write, which is H-02.
- `provision::subscribe` - the actual production entry point - is never exercised.
- The suite tests each module against its own state; no test crosses the migration→storage boundary, where M-05's double credit occurs.

> **CONVENTION COMPLIANCE**
>
> The suite already follows several practices that are commonly missed: assertions are bare `assert!(cond)` with no abort codes that could collide with application errors, and `#[expected_failure]` attributes name a specific `abort_code` rather than accepting any failure. Section 8 lists the remaining convention gaps.

SUGGESTED ADDITION

Add a property-style test module asserting the system-wide invariants directly: after any sequence of operations, `account_total_bytes` never exceeds `effective_storage_limit`; every blob's `expiry_epoch − current_epoch` is within the plan's `max_walrus_epochs`; and the sum of `total_blobs` over a user's stores never exceeds `max_total_files`. Each of the three High findings violates one of these.

8 Code Quality & Convention Review

Assessed against the Move Book's *Code Quality Checklist* and the conventions in Sui's *Security Best Practices*. Nothing in this section is a vulnerability. It is included because convention drift is what allowed several findings above to hide: L-01's inconsistent guard sets and H-02's missing liveness check are both cases of two sibling functions that should have looked identical and did not.

8.1 Already compliant

The package is in good shape on most points, including several that are commonly missed:

- Move 2024 edition declared; framework dependencies left implicit (`[dependencies]` is correctly empty for Sui 1.45+).
- Module label syntax (`module waldrop::admin;`) throughout - no legacy braced modules.
- Error constants in `EPascalCase` , value constants in `ALL_CAPS` , capabilities suffixed `Cap` .
- All 30+ event structs named in the past tense (`BlobRegistered` , `SubscriptionRenewed` , `UserDeregistered`).
- No `public entry` anywhere. The three `entry` functions are correctly private, which matters most for `seal_approve` .
- Composable design: `subscription::subscribe` returns its objects for PTB composition, with `provision::subscribe` as the deliberately non-composable `entry` wrapper. This is exactly the recommended split.
- Doc comments use `///` ; `id.delete()` and `ctx.sender()` use method syntax; no redundant `{Self}` imports.
- All 88 error constants are unique within their module - no abort-code collisions, which keeps `#[expected_failure(abort_code = ...)]` meaningful.
- Tests assert with bare `assert!(cond)` - no numeric abort codes that could accidentally match application errors - and every `#[expected_failure]` names a specific `abort_code` .

8.2 Gaps

CONVENTION	SITES	DETAIL
Capabilities go second	17	Every admin function takes `_: &AdminCap` as the *first* parameter (`admin.move:`186 and 16 others). The convention places the object first and the capability second, so method syntax works: `config.pause(&cap, ctx)` rather than `admin::pause(&cap, &mut config, ctx)` . Changing this is a breaking API change - worth doing now, before mainnet, or not at all.

No `get_` prefix on getters	3	`admin::get_user_account_id`, `storage::get_blob`, `subscription::get_plan`. Should be `user_account_id`, `blob`, `plan`. Note the rest of the codebase already follows the convention (`account_owner`, `sub_tier`, `folder_name`), so these three are the outliers.
String literals over `string::utf8`	14	`string::utf8(b"Free")` at `subscription.move:254` and throughout `display.move`. Move 2024 checks `let name: String = "Free";` at compile time and reads better. The four `key_*()` helpers in `display.move` become unnecessary.
Loop macros over `while`	6	`storage.move:505, 522, 724, 783, 869, 1117`. The marker extraction at 724 is `vector::tabulate!(SEAL_MARKER_LEN, |i| id[prefix_len + i])`; the batch-size sum at 505 is `blob_refs.fold!(0, |acc, r| acc + r.size_bytes)`. Both are in security-relevant code, where the shorter form is easier to review.
`..` in struct unpack	2	`migration::release_job` (`migration.move:273–291`) names twelve fields as `_` to reach six it uses; `storage::delete_folder` (`storage.move:954`) names three to reach one. Both collapse to `let MigrationJob { id, owner, status, .. } = job;`.
`Option` macros	~40	The `if (option::is_some(&x)) { ... *option::borrow(&x) ... }` pattern recurs heavily in `storage.move`. Many collapse to `x.do!(|v| ...)` or `x.destroy_or!(default)`.
Merge test attributes	57	`#[test]` and `#[expected_failure(...)]` are on separate lines; the convention is `#[test, expected_failure(abort_code = ...)]`.
Drop `test_` prefix	124	Inside a module already named `*_tests`, the prefix is redundant: `test_transfer_account_to_self_fails` reads better as `transfer_account_to_self_fails`.
Use `assert_eq!`	all	No use of `std::unit_test::assert_eq`, which prints both values on failure. `assert!(a == b)` only reports that the test failed.

Most of the above is mechanical. Adopting the Move formatter as a CI check - once I-05 is fixed and CI actually runs - would keep the formatting subset from drifting again.

9 Remediation Checklist

ID	PRIORITY	ACTION
H-01	BEFORE DEPLOY	Add `reserved_bytes` to `UserAccount`; reserve at `create_job`, settle at `complete_job`, release at `release_job`.
H-02	BEFORE DEPLOY	Add `is_active` assertion to `register_blob` and `register_quilt`.

H-03	BEFORE DEPLOY	Bound `new_expiry - current_epoch` by `max_allowed`, not `additional_epochs`.
M-03	BEFORE DEPLOY	Assert `account_blob_store_id == object::id(blob_store)` in `transfer_account`.
M-04	BEFORE DEPLOY	Remove the `shares` entry and its `share_markers` mirror in `release_blob`.
M-07	BEFORE DEPLOY	Decide the upgrade policy: split out an immutable SEAL-policy package, or put the `UpgradeCap` behind a multisig + timelock. Publish the decision.
M-01	SAME RELEASE	Bind `sub` and `account` to the sender in `extend_blob`, `create_job`, `complete_job`, `release_job`.
M-02	SAME RELEASE	Check file caps against `account_total_blobs`, not `store.total_blobs`.
M-05	SAME RELEASE	Make blob registration the sole writer of `total_bytes_stored`.
M-06	SAME RELEASE	Drop `store` from `UserAccount` and `Subscription`; switch to `transfer::transfer`.
L-01	NEXT RELEASE	Define and apply a guard-set policy per function class; document it.
M-08	SAME RELEASE	Bound `file_count` at `create_job` with an explicit max, so "unlimited" never means an unbounded on-chain vector.
L-09	NEXT RELEASE	Emit summary events from `clear_blob_shares` and `clear_folder_viewers`.
L-10	IMMEDIATE	Raise the mainnet gas budget; make `deploy.sh` surface the `UpgradeCap` and `AdminCap`.
L-08	SAME RELEASE	Add an `admin_epoch` to `AdminCap` + `GlobalConfig` and a rotation function, so a leaked cap can be revoked without possessing it.
L-02–L-07	NEXT RELEASE	Address individually as described in section 5.3.
§8.2	NEXT RELEASE	Convention gaps. Do the capability-parameter reordering now if at all - it is a breaking API change and gets harder after mainnet.
I-05	IMMEDIATE	Fix the CI path filter - the test gate is currently not running on any change.
I-02, I-04	NEXT RELEASE	Remove loyalty dead code; correct the README.

Appendix A Proof-of-Concept Harness

Seven Move tests were written to confirm the findings empirically. They use only the public API and the standard `sui::test_scenario` framework, with no `#[test_only]` backdoors into module internals. Placed at `tests/zz_audit_poc.move` and run against commit `d889b0c`:

```
$ sui move test zz_audit_poc

[ PASS ] waldrop::zz_audit_poc::poc_concurrent_jobs_bypass_storage_quota          H-01
[ PASS ] waldrop::zz_audit_poc::poc_expired_subscription_still_registers_blobs    H-02
[ PASS ] waldrop::zz_audit_poc::poc_extend_blob_repeats_past_plan_ceiling         H-03
[ PASS ] waldrop::zz_audit_poc::poc_extend_blob_accepts_foreign_subscription      M-01
[ PASS ] waldrop::zz_audit_poc::poc_max_total_files_reset_by_second_store         M-02
[ PASS ] waldrop::zz_audit_poc::poc_transfer_account_with_unrelated_store         M-03
[ PASS ] waldrop::zz_audit_poc::poc_stale_share_survives_blob_release             M-04

Test result: OK. Total tests: 7; passed: 7; failed: 0
```

A *passing* test here means the exploit succeeded - each test asserts the violated post-condition. After remediation these should be inverted into `#[expected_failure]` regression tests carrying the specific abort code the fix introduces, so the suite proves the guard is present rather than merely that the happy path still works.

The full harness accompanies this report as `waldrop_audit_poc.move`. It was removed from the repository after verification; the working tree is unmodified.

Appendix B Module Inventory

MODULE	KEY OBJECTS	ENTRY / PUBLIC MUTATORS	FINDINGS
`admin`	`AdminCap` (owned) `GlobalConfig` (shared)	7 admin setters, pause / unpause, `migrate_version`, 3 package-internal registry mutators	I-01, I-02, I-03, L-08
`subscription`	`PlanRegistry` (shared) `UserAccount`, `Subscription` (owned)	`subscribe`, `upgrade`, `request_downgrade`, `cancel_downgrade`, `cancel`, `renew`, 5 admin functions	M-06, L-02, L-03
`storage`	`BlobStore` (shared) `BlobRef`, `Folder` (embedded)	`register_blob`, `register_quilt`, `extend_blob`, `release_blob`, viewer / share / folder management, `seal_approve`, `transfer_account`	H-02, H-03, M-01–M-04, M-07, L-04, L-06, L-07, L-09
`migration`	`MigrationJob` (owned)	`create_job`, `complete_job`, `release_job`	H-01, M-01, M-05, M-08, L-05

`treasury`	`Treasury` (shared, `Bag` of balances)	`deposit` (package), `withdraw`, `process_refund`	I-01, I-02
`provision`	-	`subscribe` (the only `entry` user path)	I-04, I-06
`display`	`Publisher`, 5 `DisplayCap<T>`	`create_display`, `setup_display`	I-06

ACCESS CONTROL MATRIX - STORAGE MODULE

FUNCTION	OWNER	SUB BOUND	SUB LIVE	PAUSE	VERSION
`register_blob`	✓	✓	✗	✓	✓
`register_quilt`	✓	✓	✗	✓	✓
`extend_blob`	✓	✗	✓	✓	✓
`release_blob`	✓	n/a	n/a	-	-
`add_viewer`	✓	n/a	n/a	✓	-
`remove_viewer`	✓	n/a	n/a	-	-
`add_blob_share`	✓	n/a	n/a	✓	-
`create_folder`	✓	n/a	n/a	✓	✓
`create_blob_store`	n/a	n/a	n/a	-	-
`transfer_account`	✓	✓	n/a	✓	✓

✓ enforced · ✗ missing and material · - absent, see L-01 · n/a not applicable. "Sub bound" = the `Subscription` argument is checked against `ctx.sender()`; "Sub live" = `is_active` is asserted.

Appendix C Fix Patches

Every patch below was applied to a copy of the package at commit `d889b0c`, compiled with Sui CLI 1.76.0, and re-tested against the Appendix A harness. The complete fix set is **72 lines added and 18 removed** across four files - the findings are concentrated in a handful of missing assertions rather than spread through the design.

> **VERIFICATION**
>
> After patching, the proof-of-concept suite inverts as intended. Five exploits now abort with the specific error code the fix introduces, and the sixth fails on its own assertion that the stale grant survives - which is what M-04 being closed looks like from the attacker's side. PoC 3 no longer compiles at all: with `store` removed, the subscription-rental primitive cannot be expressed in Move.

ID	FILE	CHANGE	LINES
H-01	`subscription`, `migration`	Add `reserved_bytes` to `UserAccount`; reserve at admission, release at completion and abandonment	+21
H-02	`storage`	`is_active` assertion in `register_blob` and `register_quilt`	+2
H-03	`storage`	Bound the resulting lifetime instead of the increment	+2 −2
M-01	`storage`	Bind `sub` to the sender in `extend_blob`	+1
M-02	`storage`	Count files against the account, not the store	+8 −2
M-03	`storage`	Assert the `BlobStore` is the one bound to the account	+3
M-04	`storage`	Drop the share allowlist when the blob is released	+6
M-06	`subscription`, `storage`, `provision`	Remove `store`; route transfers through package-internal senders	+12 −10
M-08	`migration`	Hard ceiling on `file_count` independent of the plan	+6

M-05 (single storage ledger) and M-07 (upgrade policy) are excluded. M-05 is a design decision about which module owns the counter, and the right shape depends on whether job-completed blobs are always subsequently registered - a question for the team, not a mechanical patch. M-07 is a deployment procedure, not code.

C.1 SUBSCRIPTION.MOVE - H-01, M-06

```
// Abilities: `store` removed so `storage::transfer_account` is the only
// path by which these objects change hands.
-public struct UserAccount has key, store {
+public struct UserAccount has key {
     id: UID,
     owner: address,
     /* ... */
     blob_store_id: Option<ID>,
+    /// Bytes committed by RUNNING jobs but not yet stored (H-01).
+    reserved_bytes: u64,
 }

-public struct Subscription has key, store {
+public struct Subscription has key {

// `transfer::transfer` only works inside the defining module, so `provision`
// and `storage` need these package-internal senders.
+public(package) fun send_account(account: UserAccount, to: address) {
+    transfer::transfer(account, to);
+}
+
+public(package) fun send_subscription(sub: Subscription, to: address) {
+    transfer::transfer(sub, to);
+}
+
+public(package) fun reserve_storage(account: &mut UserAccount, bytes: u64) {
+    account.reserved_bytes = account.reserved_bytes + bytes;
+}
+
+public(package) fun release_reservation(account: &mut UserAccount, bytes: u64) {
+    account.reserved_bytes = if (account.reserved_bytes > bytes) {
+        account.reserved_bytes - bytes
+    } else { 0 };
+}
+
+public fun account_reserved_bytes(account: &UserAccount): u64 { account.reserved_bytes }
```

C.2 MIGRATION.MOVE - H-01, M-08

```
+/// Hard ceiling on `blob_ids`, independent of the plan's `max_files_per_job`
+/// (which is 0 = unlimited on Pro). Keeps the job object inside Sui's
+/// 256 KB object limit and its inputs inside the tx size limit (M-08).
+const MAX_BLOB_IDS_PER_JOB: u64 = 2_000;

// create_job — admission now counts what other open jobs have committed.
 assert!(
     subscription::account_total_bytes(account)
+        + subscription::account_reserved_bytes(account)
         + total_size_bytes <= effective_limit,
     EStorageLimitExceeded,
 );

 assert!(file_count > 0, EZeroFiles);
+assert!(file_count <= MAX_BLOB_IDS_PER_JOB, EFileLimitExceeded);

 subscription::increment_active_jobs(account);
+subscription::reserve_storage(account, total_size_bytes);

// complete_job — settle the reservation before crediting actual usage.
+subscription::release_reservation(account, job.total_size_bytes);
 subscription::add_storage(account, verified_bytes, blob_count, job.encrypted);

// release_job — an abandoned job must give its budget back.
-        total_size_bytes: _,
+        total_size_bytes,
 /* ... */
 if (status == STATUS_RUNNING) {
     subscription::decrement_active_jobs(account);
+    subscription::release_reservation(account, total_size_bytes);
 };
```

C.3 STORAGE.MOVE - H-02, H-03, M-01, M-02, M-03, M-04

```
+const EWrongBlobStore: u64 = 439;
+const EStoreNotBound: u64 = 440;

// H-02 - in BOTH register_blob and register_quilt, beside the owner checks.
 assert!(subscription::account_owner(account) == ctx.sender(), ENotOwner);
+assert!(subscription::is_active(sub, ctx.epoch()), ESubscriptionNotActive);

// M-02 - the cap is per user, not per store. Same edit in register_quilt.
-assert!(store.total_blobs + 1 <= max_total, ETotalFilesExceeded);
+assert!(
+    subscription::account_total_blobs(account) + 1 <= max_total,
+    ETotalFilesExceeded,
+);

// M-01 - extend_blob must bind the subscription to the caller.
 assert!(ctx.sender() == store.owner, ENotOwner);
+assert!(subscription::sub_owner(sub) == ctx.sender(), ENotOwner);

// H-03 - bound the resulting lifetime, which is what the doc comment promised.
-assert!(additional_epochs <= max_allowed, EEpochsExceedPlanMax);
 let old_expiry = blob.expiry_epoch;
 let new_expiry = old_expiry + additional_epochs;
+assert!(new_expiry > current_epoch, EInvalidEpoch);
+assert!(new_expiry - current_epoch <= max_allowed, EEpochsExceedPlanMax);

// M-04 - in release_blob, beside the existing marker_folder cleanup.
 if (store.marker_folder.contains(marker)) {
     store.marker_folder.remove(marker);
 };
+if (table::contains(&store.shares, marker)) {
+    let allowlist = table::remove(&mut store.shares, marker);
+    let _ = vec_set::into_keys(allowlist);
+    let (found, idx) = vector::index_of(&store.share_markers, &marker);
+    if (found) { vector::remove(&mut store.share_markers, idx); };
+};

// M-03 - in transfer_account, before any mutation.
 assert!(blob_store.owner == sender, ENotOwner);
+let bound = subscription::account_blob_store_id(&user_account);
+assert!(option::is_some(&bound), EStoreNotBound);
+assert!(*option::borrow(&bound) == object::id(blob_store), EWrongBlobStore);

// M-06 - UserAccount / Subscription no longer have `store`.
-transfer::public_transfer(user_account, recipient);
-transfer::public_transfer(subscription, recipient);
+subscription::send_account(user_account, recipient);
+subscription::send_subscription(subscription, recipient);
```

C.4 NOTES ON APPLYING THESE

- **Do it before mainnet.** Adding `reserved_bytes` changes the layout of `UserAccount`, and removing `store` changes its abilities. Neither is possible on a live package - Sui upgrades cannot alter existing struct definitions.
- **M-03 rejects legacy accounts.** The binding assert fails with `EStoreNotBound` for any account whose `blob_store_id` is `None`. Accounts created through `provision::subscribe` always have it set; accounts

created by calling `subscription::subscribe` directly do not. Decide whether that path should remain reachable, or set the field there too.

- **M-06 has a deliberate side effect.** Once the returned objects lack `store`, a PTB can no longer take them from `subscription::subscribe` and transfer them itself - it must pass them to a Move function. That makes `provision::subscribe` the only practical entry point, which is what its own doc comment says the design intends.
- **Existing tests need updating.** The suite calls `transfer::public_transfer` on both types in several places; these become `subscription::send_account` / `send_subscription`. This is the only test change the fix set forces.
- **M-02 changes an abort's meaning.** `ETotalFilesExceeded` now reflects a per-user total rather than a per-store one, so a user with blobs spread across several stores may hit it sooner than before. That is the intended behaviour, but any client message explaining the error should be reworded.

Disclaimer

This assessment was performed by **The Blockchain Team** (https://theblockchain.team). This assessment covers the Move sources listed in section 2.1 at commit `d889b0c` only. A security review is a best-effort exercise conducted under time constraints and cannot prove the absence of vulnerabilities. It does not constitute an endorsement of the protocol, its business model, or its economic design, and it is not financial advice. Findings relating to off-chain components are inferred from their on-chain interfaces and were not verified against those components' implementations. Any modification to the code after the reviewed commit - including the remediations recommended here - invalidates these results and should be re-reviewed. The proof-of-concept code in Appendix A is provided for verification and regression-testing purposes only.

ATTESTATION

The Blockchain Team · https://theblockchain.team
Issued 17 August 2026 · Report version 1.0

- End of report -