PRE-LAUNCH SECURITY REVIEW · SAMPLE FINDINGS SECTION

Is this safe to put a competitor's brokerage data into?

Olha Shevchenko · Senior Backend Engineer · Node.js + AWS · 2026-08-26

Three findings from a multi-tenant review, in the format every finding in the register uses.

Sample — fictional client. Quillstone is not a real company. The findings, evidence, configuration and timeline are invented to show the deliverable. No client work is referenced. External technical claims are cited at the end.

SUBJECT	FINDINGS SHOWN	HIGHEST	BASIS
Multi-tenant SaaS Node.js · PostgreSQL · AWS	**3** of the register	**P0 — tenant isolation**	**Config and code** no runtime access

SUBJECT

Quillstone, a multi-tenant SaaS for independent insurance brokers. Pre-launch, onboarding a first enterprise customer: a brokerage group of roughly 40 branch offices, each of which must not see the others' book of business. The review question above is the client's own.

1 ASSESSMENT BASIS

Verified from material provided: read-only AWS console access (IAM, RDS, S3, VPC), read access to two application repositories, a PostgreSQL schema dump including role attributes and grants, and the deployment workflow definition.

Not verified: no penetration testing, no load or performance testing, no capture of live traffic, and no access to a running production environment. Findings describe what a read-only observer can establish from configuration and code. Whether any of this has been exploited is a separate question that this review cannot answer.

1 Tenant isolation is enforced in application code and nowhere else — P0

CLASSIFICATION

Not P1, because the exposure crosses the boundary the product exists to maintain and the first enterprise customer's data is the thing that crosses it. Not a design preference, because two live code paths already bypass the control.

OBSERVED CONFIGURATION

Every tenant-scoped table carries a `brokerage_id` column. Isolation is applied by a default query scope in the data-access layer, which appends `where brokerage_id = ?` from the request context. Row-level security is enabled on 3 of the 19 tenant-scoped tables. On the remaining 16, `rowsecurity` is false in `pg_tables`.

EVIDENCE

- Schema dump: `policies`, `claims`, `commission_statements`, `client_documents` and 12 further tenant-scoped tables have no policy defined and RLS not enabled.
- The application connects with a single role that owns the tables. A table owner is exempt from the table's own policies unless `FORCE ROW LEVEL SECURITY` is set, and it is not set on any of the three tables that do have policies. So the three tables that look protected are not protected against this connection either.
- Two paths do not go through the default scope: the reporting export builds its column list into a raw query, and the renewal-notice worker iterates every row by design and filters afterwards in application code.
- The `brokerage_id` in the request context is read from a JWT claim. Nothing re-checks that claim against current membership. Tokens carry a 24-hour expiry.

THE CHAIN — THESE COMPOSE INTO ONE PATH, NOT FOUR SEPARATE WEAKNESSES

1. A token is minted with `brokerage_id` fixed at sign-in.
2. Membership changes — a broker leaves the firm, an account is moved, access is revoked in the admin UI. The token keeps working until it expires.
3. Any request on that token is scoped by the application to a brokerage the caller no longer belongs to.
4. The database applies no independent check, because on 16 tables there is no policy and on the other 3 the connecting role owns the table.

The result is that revoking a user's access does not revoke their access for up to 24 hours, and the only thing standing between two competing brokerages is a `where` clause the reporting path already skips.

IMPACT

Cross-tenant read of the exact data the enterprise customer is being asked to trust the platform with: policies, claims history, and commission statements. Under the isolation model as built, an authorization bug anywhere in the application is a data-disclosure bug, because there is no second layer to fail into.

VERIFY OR FALSIFY

Query `pg_tables.rowsecurity` and `pg_policies` for the tenant-scoped schema — this confirms the 16-table count in one statement. Then check `rolbypassrls` and table ownership for the application role. To falsify: if the application connects as a non-owner role with `FORCE ROW LEVEL SECURITY` set and policies covering every tenant table, this finding does not stand.

REMEDIATION

Enable row-level security on every tenant-scoped table with a policy keyed on a session variable, set that variable at connection checkout, and connect as a role that does not own the tables. Then re-check membership on each request rather than trusting the token's claim, or shorten token lifetime to minutes and validate the claim server-side on refresh. The application scope stays where it is; it stops being the only control.

Effort: two to four days for the RLS retrofit including a test that asserts isolation per table, plus a day for the token change. The retrofit is mechanical; the test is the part worth the time.

2 Role is checked on the route, ownership is not checked on the record

P1

CLASSIFICATION

Below P0 because it requires a valid authenticated session, which F-01 does not.

OBSERVED CONFIGURATION

Route handlers are gated by a `requireRole` middleware. Inside the handler, the record is loaded by primary key from the path parameter. Of 34 handlers that load a single record by id, 11 apply no ownership predicate and rely on the default scope described in F-01 to supply one.

EVIDENCE

`GET /api/policies/:id`, `GET /api/claims/:id` and nine sibling routes resolve the identifier directly. The authorization decision answers *may this user read policies* rather than *may this user read this policy*. This is the object-level authorization failure OWASP lists first in its API Security Top 10, under `API1:2023`.

IMPACT

Any authenticated broker can read any record whose identifier they can guess or observe, subject only to the default scope — which F-01 establishes is absent on two paths and unenforced at the database.

VERIFY OR FALSIFY

Sign in as a broker in one brokerage, request a record identifier belonging to another, and observe the response. This is the one finding here that a functional test settles in minutes.

REMEDIATION

Make ownership part of the query rather than a filter applied after it. With F-01's RLS in place the database enforces this independently, which is why the two are sequenced together rather than fixed separately.

Effort: half a day to fix the 11 handlers once RLS is in place. Longer without it, because each handler then needs its own predicate and its own test.

3 The database and the document store have different recovery points

P1

CLASSIFICATION

A recovery gap rather than an exposure, and it becomes a P0 the first time it is needed.

OBSERVED CONFIGURATION

RDS automated backups are enabled with a seven-day retention window and point-in-time recovery available within it. Policy documents are written to an S3 bucket with versioning not enabled and a lifecycle rule that expires objects 30 days after creation.

EVIDENCE

The `client_documents` table stores S3 object keys, not the documents. So a restore of the database to a point four days ago produces rows referencing objects that may have been overwritten since — with versioning off, the previous version is gone — or expired, if the lifecycle rule has since passed over them.

IMPACT

The two stores cannot be restored to a consistent moment. A database restore yields a policy record whose signed document is either a later revision or absent, and nothing in the application distinguishes those two cases from a

healthy row. For an insurance product the document is frequently the artifact of record, so this is a correctness problem before it is an availability one.

VERIFY OR FALSIFY

Restore the most recent snapshot into an isolated environment and open ten policy records end to end, document included. This has not been done. Until it is, the recovery point is a setting rather than a measurement.

REMEDIATION

Enable versioning on the document bucket, and change the lifecycle rule in the same pass rather than after it: an expiration rule written for an unversioned bucket does not keep its meaning once versioning is on, and needs a noncurrent-version expiration alongside it to express the same intent. Set that window to match the database retention so the two stores can be reasoned about together. Then perform one restore and record how long it took and what came back. The number produced by that exercise is the real recovery objective; the current one is an assumption.

Effort: an hour for the bucket configuration. Half a day for the first restore, most of which is waiting.

2 ROOT CAUSES

Three findings, three patterns underneath them. Fixing the patterns is what stops the next instance.

1. Isolation is asserted in code and unverified in data. *(F-01, F-02)* Every control that separates one brokerage from another lives in the application. The database will return whatever it is asked for. This is why an ordinary authorization bug has data-disclosure consequences here and would not in a system with a second layer.

2. An identity claim is trusted for its whole lifetime. *(F-01)* The token is a snapshot of membership taken at sign-in and never re-examined. Access changes made in the admin interface are therefore advisory until the token expires, which is not what the interface implies.

3. Recovery is configured but never exercised. *(F-03)* Retention windows and lifecycle rules are set. No restore has been performed, so the recovery point is a value in a console rather than a measured property of the system.

3 REMEDIATION, IN ORDER

	ACTION	CLOSES	EFFORT
1	Enable RLS on all 19 tenant tables; connect as a non-owning role	**F-01**	2–4 days
2	Add ownership predicates to the 11 handlers	**F-02**	0.5 day
3	Re-validate membership per request, or shorten token lifetime	**F-01**	1 day
4	Enable bucket versioning; pair it with noncurrent-version expiration	**F-03**	1 hour
5	Perform one restore end to end and record the result	**F-03**	0.5 day

TRACK A

Internal

The engineering team runs the table above. Item 1 carries the judgment; the rest is mechanical. The test that asserts isolation per table is the deliverable that matters, not the migration.

TRACK B

Engagement

Five working days: the RLS retrofit with per-table isolation tests, the handler pass, and the first restore performed and documented. Fixed price. The isolation test suite is handed over and runs in the client's own pipeline.

SOURCES — EXTERNAL TECHNICAL CLAIMS ONLY; THE ENGAGEMENT AND ITS FINDINGS ARE INVENTED

PostgreSQL 18 documentation, Row Security Policies — table owners bypass row security unless `ALTER TABLE ... FORCE ROW LEVEL SECURITY` is set; superusers and roles with `BYPASSRLS` always bypass. `postgresql.org/docs/current/ddl-rowsecurity.html`

OWASP API Security Top 10 2023 — API1:2023 Broken Object Level Authorization: every endpoint that receives an object id should implement object-level authorization checks. `owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/`

Amazon RDS User Guide — automated backups; recovery to any point in time during the backup retention period. `docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_WorkingWithAutomatedBackups.html`

Amazon S3 User Guide — versioning is disabled by default; an expiration rule on an unversioned bucket needs a noncurrent-version expiration once versioning is enabled. `docs.aws.amazon.com/AmazonS3/latest/userguide/Versioning.html`

Olha Shevchenko — Senior Backend Engineer — Node.js + AWS — 2026-08-26

One section of a findings register, reproduced in full to show the format. A sample built on a fictional system, not a report on anyone's production environment.