0DATA LAB · PAPIER 013 · JUILLET 2026

Isolation Tenant

Architecture 9 Couches pour Infrastructure Mutualisée

Hadda TIKIJJA
0DATA Lab, France

RÉSUMÉ

L'isolation tenant dans une infrastructure mutualisée ne peut reposer sur une couche unique. Nous documentons l'architecture d'isolation à **neuf couches** déployée sur l'infrastructure 0DATA : PostgreSQL Row-Level Security, SPINA par tenant, NOVA par tenant, Cockpit par domaine, ALFA par niveau, PWA offline, plan de ports, NATS sujet namespacé, et DNS wildcard. Chaque couche fonctionne indépendamment ; la défaillance de l'une ne compromet pas les autres. Cette architecture s'inspire des **jonctions serrées** de l'épithélium biologique, où chaque cellule maintient sa propre membrane tout en participant au tissu commun. Nous décrivons chaque couche en détail, présentons le plan de ports documenté (aucun service n'écoute sur 0.0.0.0), et rapportons les résultats du test d'intrusion cross-tenant — toutes les tentatives de lecture inter-tenant ont échoué.

EN UNE PHRASE

Neuf couches d'isolation indépendantes garantissent qu'un tenant ne peut ni voir, ni atteindre, ni altérer les données d'un autre tenant — sur aucun plan de la pile.

1. Pourquoi Neuf Couches

1.1 La défense en profondeur n'est pas une métaphore

En cybersécurité classique, la défense en profondeur est un principe théorique. Dans une infrastructure mutualisée hébergeant des clients MSP aux intérêts antagonistes, elle devient une nécessité vitale. Un fournisseur de services managés peut héberger simultanément un cabinet d'avocats, une clinique, et un concurrent direct du premier — sur le même serveur physique.

L'approche traditionnelle du multi-tenant repose sur une séparation applicative : l'application filtre les requêtes selon un `tenant_id`. Cette approche échoue spectaculairement lorsqu'un bug, une faille d'injection, ou une escalade de privilège contourne cette couche unique.

1.2 Le modèle biologique : membrane cellulaire et jonctions serrées

L'épithélium intestinal humain illustre notre paradigme. Chaque cellule possède sa propre **membrane phospholipidique** — c'est la première barrière. Entre les cellules, des **jonctions serrées** (tight junctions) forment une seconde barrière, imperméable même aux petites molécules. Une troisième barrière — le **glycocalyx** — filtre les macromolécules. Aucune de ces barrières n'est suffisante seule ; leur superposition crée une étanchéité proche de l'absolu.

TRANSPOSITION BIOMIMÉTIQUE

Membrane cellulaire → Chaque tenant a ses propres instances SPINA, NOVA, ALFA
Jonctions serrées → PostgreSQL RLS empêche toute fuite au niveau de la base
Glycocalyx → Le plan de ports, le DNS wildcard, et NATS namespacé forment une couche de filtration réseau

1.3 Les neuf couches

COUCHE	NOM	DOMAINE	MÉCANISME
1	PostgreSQL RLS		Row-Level Security policies

		Base de données	
2	Schéma PostgreSQL	Base de données	Schéma logique par tenant
3	Policies applicatives	Base de données	Contraintes CHECK, triggers
4	SPINA v2	Signature	HMAC-SHA256 local + eIDAS RFC 3161
5	NOVA	Observation	Instance par tenant, bind 127.0.0.1
6	ALFA	Décisionnel	Moteur commun, mémoire isolée
7	Cockpit	Interface	Domaine propre, JWT signé par tenant
8	DNS / NATS	Communication	Wildcard DNS, sujets namespacés
9	Plan de ports	Réseau	Aucun service sur 0.0.0.0, ports documentés

2. Couches 1–3 : PostgreSQL, le Socle

2.1 Row-Level Security (Couche 1)

PostgreSQL Row-Level Security est le mécanisme le plus profond de l'isolation. Chaque ligne de chaque table porte un `tenant_id`. Une policy RLS appliquée à chaque table garantit que la session PostgreSQL connectée pour le tenant A ne peut physiquement pas lire les lignes du tenant B.

```
ALTER TABLE organisms ENABLE ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation ON organisms
    USING (tenant_id = current_setting('app.current_tenant_id')::uuid);
```

Figure 1 : Policy RLS PostgreSQL — la colonne tenant_id est comparée à la variable de session.

La fonction `current_setting` lit une variable de session PostgreSQL. Cette variable est positionnée par le pool de connexions au moment de l'authentification du tenant. Au-

cune requête applicative ne peut la modifier — elle est en lecture seule depuis l'application. Cette couche est inviolable sans accès super-utilisateur PostgreSQL.

2.2 Schéma par tenant (Couche 2)

Au-delà du RLS, chaque tenant dispose d'un **schéma logique dédié** — non pas un schema PostgreSQL au sens `CREATE SCHEMA`, mais une projection de données restreinte. Les vues matérialisées, les index partiels, et les séquences sont filtrés par `tenant_id`. Un tenant ne peut pas énumérer les autres tenants, car la table `tenants` elle-même est protégée par RLS.

Cette couche empêche les attaques par inférence : même si un tenant parvenait à mesurer le temps d'exécution d'une requête (timing attack), les index partiels garantissent que les données des autres tenants ne sont jamais parcourues.

2.3 Contraintes et triggers (Couche 3)

La troisième couche PostgreSQL utilise des contraintes `CHECK` et des triggers `BEFORE INSERT/UPDATE` pour empêcher l'écriture de données qui ne respecteraient pas l'isolation :

```
ALTER TABLE organisms ADD CONSTRAINT chk_tenant_owns
    CHECK (tenant_id IS NOT NULL
           AND tenant_id = current_setting('app.current_tenant_id')::uuid);
```

Figure 2 : Contrainte CHECK empêchant l'écriture cross-tenant.

Un trigger supplémentaire verrouille toute tentative de modification de la colonne `tenant_id` après insertion. Cette redondance est volontaire : si la policy RLS est désactivée par erreur (par exemple lors d'une maintenance), les contraintes et triggers maintiennent l'isolation.

3. Couches 4–6 : SPINA, NOVA, ALFA

3.1 SPINA v2 — Signature par tenant (Couche 4)

SPINA (Signature Protocol for Integrity and Non-repudiation of Assets) est le protocole de signature de 0DATA. En version 2, chaque tenant possède :

- Une **clé HMAC-SHA256 locale**, générée et stockée dans l'espace du tenant
- Un **certificat eIDAS RFC 3161** pour l'horodatage qualifié
- Un **keystore isolé**, chiffré avec une passphrase dérivée du secret du tenant

Deux tenants ne partagent jamais de matériel cryptographique. Si le tenant A compromise sa clé HMAC, les signatures du tenant B restent valides. L'horodatage eIDAS est fédéré au niveau de l'infrastructure mais le timestamp token est stocké dans le schéma du tenant.

La vérification de signature est exécutée dans le contexte PostgreSQL du tenant. La fonction `spina_verify()` utilise `SECURITY DEFINER` avec un `SET app.current_tenant_id` explicite — elle ne peut pas fuiter hors du tenant.

3.2 NOVA — Instance par tenant (Couche 5)

NOVA est le moteur d'observation du réseau. Chaque tenant MSP possède sa **propre instance NOVA**, exécutée comme un processus dédié :

- Bind exclusif sur `127.0.0.1`, port distinct pour chaque tenant
- Base de données de scan locale (SQLite ou schéma PostgreSQL isolé)
- Aucun partage de mémoire entre instances
- Fichier de configuration distinct (`/opt/nova/tenants/<tenant_id>/config.yaml`)

Un tenant qui contrôle son réseau local (et donc les cibles scannées par NOVA) ne peut pas influencer l'instance NOVA d'un autre tenant. Les processus sont isolés au niveau du système d'exploitation — `kill -9` sur le PID du tenant A n'affecte pas le tenant B.

3.3 ALFA — Moteur commun, mémoire isolée (Couche 6)

ALFA (Analytics Layer for Forensic Assessment) est le moteur décisionnel partagé. Contrairement à NOVA, ALFA utilise un **moteur commun** pour l'efficacité des ressources, mais avec une **isolation mémoire stricte** :

- Chaque niveau d'analyse (tenant, site, équipement) est cloisonné
- Les données sont chargées dans des segments mémoire étanches
- Les résultats d'analyse sont écrits dans le schéma PostgreSQL du tenant
- Aucune corrélation inter-tenant n'est effectuée sans consentement explicite

ALFA applique le principe du *besoin d'en connaître* au niveau algorithmique : un modèle d'analyse exécuté pour le tenant A ne reçoit que les données du tenant A. Les statistiques agrégées multi-tenant (utiles pour le benchmarking) utilisent des données anonymisées avec differential privacy (ε = 1.0).

4. Couches 7–9 : Cockpit, DNS, NATS

4.1 Cockpit — Domaine propre et JWT par tenant (Couche 7)

Chaque tenant accède à l'interface Cockpit via un **domaine dédié** : `client.odata.fr`. Le certificat SSL (Let's Encrypt) est émis pour ce domaine. Le JWT d'authentification est signé avec une clé secrète propre au tenant, et contient le `tenant_id` dans ses claims.

```
Authorization: Bearer eyJ...tenant_id:"a1b2c3d4-..."...
```

Figure 3 : En-tête d'authentification Cockpit — JWT contenant le tenant_id.

Le middleware nginx valide le JWT avant de router la requête. Une tentative d'accès à `client-b.odata.fr` avec un JWT du tenant A est rejetée au niveau du reverse proxy — avant même que la requête n'atteigne l'API. Le domaine lui-même constitue une frontière d'isolation : un tenant ne peut pas deviner le sous-domaine d'un autre tenant (les noms sont des UUID ou des identifiants opaques).

4.2 DNS wildcard et NATS namespacé (Couche 8)

La couche 8 opère au niveau de la communication inter-service :

DNS wildcard. L'enregistrement `*.odata.fr` pointe vers l'infrastructure. Chaque sous-domaine est résolu mais le serveur d'application filtre les domaines non enregistrés. Un tenant ne peut pas créer un sous-domaine arbitraire — l'enregistrement DNS est contrôlé par l'infrastructure.

NATS namespacé. Le bus de messages NATS utilise des sujets préfixés par le tenant :

```
tenant.a1b2c3d4.nova.scan.start
tenant.a1b2c3d4.spina.sign
tenant.e5f6g7h8.nova.scan.start
```

Figure 4 : Sujets NATS avec préfixe tenant. Les ACL restreignent l'accès par préfixe.

Les ACL NATS sont configurées pour que les credentials du tenant A ne puissent publier ou souscrire qu'aux sujets `tenant.a1b2c3d4.*`. Le serveur NATS applique ces ACL au niveau du protocole — un client ne peut pas contourner le namespacing.

4.3 Plan de ports — Aucun service sur 0.0.0.0 (Couche 9)

La couche 9 est la plus physique : le plan de ports documenté. Aucun service interne n'écoute sur `0.0.0.0`. Chaque service est bindé sur `127.0.0.1` avec un port explicite. Le firewall (UFW, politique default deny) bloque tout le trafic entrant qui n'est pas explicitement autorisé. Cette couche est documentée en détail dans la section 5.

5. Plan de Ports Documenté

Le tableau suivant documente l'intégralité des ports en écoute sur l'infrastructure. La colonne **Bind** indique l'interface d'écoute. **Aucun service n'écoute sur 0.0.0.0** à l'exception des ports publics délibérément exposés (HTTP/S, SSH, SIP).

5.1 Ports publics (exposés)

PORT	PROTOCOLE	SERVICE	JUSTIFICATION
22	TCP	SSH	Accès administratif
80	TCP	HTTP (nginx)	Redirection HTTPS + ACME
443	TCP/UDP	HTTPS (nginx)	Reverse proxy, HTTP/3 QUIC
5060	UDP	SIP (FreeSWITCH)	Téléphonie VoIP
5080	UDP	SIP (FreeSWITCH)	Téléphonie VoIP

5.2 Ports internes (bind 127.0.0.1)

PORT	SERVICE	TENANT	RÔLE
5432	PostgreSQL	Partagé (RLS)	Base de données principale
6379	Redis	Partagé (préfixé)	Cache, sessions
4222	NATS	Partagé (namespacé)	Bus de messages
5090–5099	NOVA instances	Par tenant	Scan réseau
5100–5109	NOVA API	Par tenant	API REST NOVA
5110–5119	SPINA workers	Par tenant	Signature/horodatage
8300	API principale	Partagé (RLS)	API FastAPI
8400–8409	Cockpit workers	Par tenant	Interface web Cockpit
9090	Cockpit système	Infrastructure	Administration système

5.3 Règles UFW

```
ufw default deny incoming
ufw default allow outgoing
ufw allow 22/tcp
ufw allow 80/tcp
ufw allow 443
ufw allow 5060/udp
ufw allow 5080/udp
```

Figure 5 : Règles UFW — seuls 5 ports sont exposés publiquement.

Tous les autres ports sont bloqués au niveau firewall. Le bind `127.0.0.1` ajoute une deuxième barrière : même si le firewall était désactivé, les services internes ne seraient pas accessibles depuis l'extérieur.

6. Vérification — Test d'Intrusion Cross-Tenant

6.1 Protocole de test

Le test d'intrusion cross-tenant a été conduit le 23 juillet 2026. Deux tenants de test ont été créés :

- **Tenant A** (`a1b2c3d4-...`) : données de test « clinique vétérinaire »
- **Tenant B** (`e5f6g7h8-...`) : données de test « cabinet comptable »

Les tests ont été exécutés depuis le contexte du tenant A, en tentant d'accéder aux données du tenant B. Tous les tests ont utilisé des credentials valides du tenant A.

6.2 Vecteurs testés

#	VECTEUR	MÉTHODE	RÉSULTAT
1	Lecture directe SQL	`SELECT * FROM organisms` (sans filtre tenant_id)	**Échec** — RLS retourne 0 ligne
2	Modification tenant_id	`UPDATE organisms SET tenant_id = 'e5f6...'`	**Échec** — Contrainte CHECK rejette

3	Accès API cross-tenant	`GET /api/organisms?tenant_id=e5f6...`	**Échec** — API ignore le paramètre, utilise le JWT
4	JWT forgé	JWT avec `tenant_id: "e5f6..."` signé clé A	**Échec** — Signature invalide
5	Accès Cockpit domaine B	Connexion à `tenant-b.odata.fr` avec JWT A	**Échec** — Domaine + JWT mismatch, 403
6	Souscription NATS	`nats.sub("tenant.e5f6.*")` avec creds A	**Échec** — ACL rejette la souscription
7	Process NOVA	Connexion au port NOVA du tenant B	**Échec** — Bind 127.0.0.1, firewall bloque
8	Clé SPINA	Tentative de vérification avec clé du tenant B	**Échec** — HMAC mismatch
9	Énumération DNS	Tentative de découverte de sous-domaines	**Échec** — UUID opaques, pas de zone transfer

6.3 Résultat

RÉSULTAT DU PENTEST CROSS-TENANT

Zéro succès sur neuf vecteurs. Chaque couche a bloqué les tentatives qui la concernaient. Les couches 1 (RLS) et 2 (contraintes) ont bloqué les tentatives SQL. La couche 4 (SPINA) a bloqué la falsification cryptographique. La couche 8 (NATS) a bloqué l'interception de messages. La couche 9 (firewall + bind) a bloqué l'accès réseau direct.

Aucune couche n'a été contournée. Aucune combinaison de vecteurs n'a permis d'inférer des données inter-tenant.

7. Pourquoi le Multi-Tenant Classique Est Insuffisant

7.1 Le modèle à couche unique

L'architecture multi-tenant classique — celle de la plupart des SaaS — repose sur une séparation applicative unique. L'application ajoute `WHERE tenant_id = ?` à chaque requête. Cette approche présente trois défaillances structurelles :

1. **Point de défaillance unique.** Un oubli de clause WHERE dans une jointure, une migration de schéma mal testée, ou un bug d'ORM expose les données de tous les tenants.
2. **Pas de défense contre l'élévation de privilège.** Un attaquant qui obtient un accès administrateur applicatif (via injection SQL, credential stuffing, ou vulnérabilité de dépendance) contourne l'intégralité de l'isolation.
3. **Pas d'isolation au niveau infrastructure.** Base de données, cache, bus de messages — tous ces composants sont partagés sans barrière interne. Une fuite mémoire dans Redis expose les sessions de tous les tenants.

7.2 Comparaison avec l'approche 0DATA

DIMENSION	MULTI-TENANT CLASSIQUE	ISOLATION 0DATA (9 COUCHES)
Base de données	`WHERE tenant_id = ?`	RLS + contraintes + triggers
Signature	Partagée	HMAC par tenant + eIDAS
Observation	Partagée	Instance NOVA par tenant
Analyse	Partagée	ALFA à mémoire isolée
Interface	Même domaine	Domaine par tenant, JWT dédié
Messagerie	Sujets partagés	NATS namespacé avec ACL
Réseau	Services sur 0.0.0.0	Bind 127.0.0.1, firewall
Profondeur	1 couche	9 couches indépendantes

7.3 Le coût de l'isolation

L'isolation à neuf couches a un coût : complexité de déploiement, consommation mémoire supplémentaire (une instance NOVA par tenant), et maintenance des ACL NATS. Ce coût est accepté comme un **investissement structurel**. Dans le contexte MSP où une fuite de données cross-tenant peut entraîner des poursuites judiciaires, des sanctions RGPD, et la perte de clients, le coût de l'isolation est inférieur au coût d'une violation.

L'architecture est documentée, automatisée (déploiement par Ansible), et testée (pentest cross-tenant à chaque déploiement). La complexité est maîtrisée par la reproductibilité.

8. Travaux Connexes et Perspectives

L'isolation tenant dans les infrastructures mutualisées est un sujet actif dans la littérature. Les architectures de type « cage » (Google Borg, AWS Nitro Enclaves) utilisent la virtualisation matérielle pour isoler les tenants — approche robuste mais coûteuse en ressources. Les architectures « shared-nothing » (chaque tenant sur son propre cluster) offrent une isolation parfaite au prix d'une inefficacité économique.

L'approche 0DATA occupe un point d'équilibre : **partage de l'infrastructure, isolation des données**. Le modèle biologique des jonctions serrées guide cette architecture — chaque couche est indépendante, redondante, et vérifiable.

Les travaux futurs incluent :

- **Rotation automatique des clés SPINA par tenant** (périodicité configurable)
- **Differential privacy multi-tenant** pour les agrégations statistiques
- **Isolation hardware** (SGX/TDX) pour les tenants à exigences réglementaires élevées
- **Audit continu cross-tenant** — test automatique des neuf couches à chaque déploiement

Références

[1] TIKIJJA, Hadda. « La Discipline ». 0DATA Lab, Papier 001, Juillet 2026.

[2] TIKIJJA, Hadda. « Le Système Nerveux ». 0DATA Lab, Papier 003, Juillet 2026.

[3] TIKIJJA, Hadda. « La Greffe Numérique ». 0DATA Lab, Papier 004, Juillet 2026.

[4] TIKIJJA, Hadda. « Le Système Immunitaire ». 0DATA Lab, Papier 005, Juillet 2026.

[5] TIKIJJA, Hadda. « Audit de Surface ». 0DATA Lab, Papier 011, Juillet 2026.

[6] PostgreSQL Documentation. « Row Security Policies ». PostgreSQL 16, 2024.

[7] NATS Documentation. « Subject-Based Messaging and ACLs ». Synadia Communications, 2024.

[8] eIDAS Regulation (EU) No 910/2014. « Electronic Identification and Trust Services ».

[9] RFC 3161. « Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP) ».

[10] Dwork, C., Roth, A. « The Algorithmic Foundations of Differential Privacy ». Foundations and Trends in Theoretical Computer Science, 2014.

Remerciements. À l'équipe PostgreSQL pour le Row-Level Security, aux mainteneurs de NATS pour le namespacing par ACL, et à l'infrastructure 0DATA qui a supporté le pentest cross-tenant sans broncher. L'isolation n'est pas une fonctionnalité — c'est la structure même du vivant.