Securing Your NATS Deployment

RethinkConn 2026 - Post-event Workshop

Seth Back, Synadia

John Weldon, Synadia

Three deployments, three security postures

All three are a single `nats-server`. What changes is the **auth model**, not the topology.

	Posture	Auth model	Where it bites
A	"we'll secure it later"	plaintext, **no auth**	anyone who can reach the port is a client
B	basic hardening	**TLS + static user/pass** in the config	no revocation primitive -- removing a user is a config edit + server reload
C	decentralized teams	**JWT operator-mode**, per-account signing	revocation exists but is connect-time; many teams can mint creds (audit surface)

Where on this spectrum are you? Where do you want to be?

What is your role?

Hands-on, built for a mixed room. Find your seat -- where you sit changes what to watch for:

You are...	today you're...	watch for
Operators / SREs	operating NATS in production	hardening, revocation, `$SYS` boundaries
Engineers/Architects	designing against NATS	the identity options: static -> JWT -> auth callout
Eng leaders	making identity & auth less risky	OSS vs Synadia Control Plane (SCP): who operates identity, and the cost of each

Everyone leaves with the runnable bundle and a hardening checklist to work through.

The five-layer NATS security stack

1. **Transport** -- is the wire encrypted? Is the peer who they say they are?
2. **Authentication** -- does the client prove identity?
3. **Authorization** -- what is the identity allowed to publish/subscribe?
4. **Isolation** -- accounts, leafnodes, $SYS boundaries.
5. **Audit** -- who did what, when. `$SYS` connect/auth events + logs/monitoring (the evidence stream); durable retention is your own tooling.

Unlike a typical HTTP stack -- transport, authn, and authz split across a gateway, a mesh, and policy services -- one NATS server enforces all three natively. The power and the risk: one surface to get right, not a chain of products to keep in sync.

What we'll cover (and what we won't)

Headlining: JWT, auth callout, and the NSC-into-Control-Plane path.

Lightly: NKey and password (high-level only).

Out of scope: Synadia Cloud as a product -- we treat it as a deployment target only.

> **Heads-up:** auth callout works with **any client credential** -- username/password, token, nkey, or TLS cert. The server forwards whatever the client presented (e.g. `client_opts.pass`) to the callout service, which decides and signs the user JWT -- and the server re-validates that signed JWT before honoring it. In our demo, clients use user/pass; the callout service runs as a dedicated `auth_user`. We'll show that wire when we get there.

Before we start: get the workshop labs running

Three host prerequisites. Everything else runs in Docker.

```
# nats CLI:  brew install nats-io/nats-tools/nats        (mac)
#            curl -sf https://binaries.nats.dev/nats-io/natscli/nats@latest | sh
# Docker:    docker.com/get-started

docker compose version     # expect v2 or later
nats --version             # expect 0.4.0+
python3 --version          # any 3.x (one lab script uses it)
git clone https://github.com/synadia-labs/nats-security-workshop && cd nats-security-workshop && make help
make prefetch              # pre-pull images (~350 MB) -- ideally done before today
```

Full step-by-step with expected output: **`LAB.md`** (linked in the pre-workshop email).

> Stuck on setup? Labs 1-3 are independent -- jump in at any `make 0N` (Lab 4 packages Lab 2's store). Separate ports (00=4222, 01=4322, 02=4422), so nothing collides.

Transport

TLS · mTLS · verify_and_map

Transport baseline

TLS, mTLS, `verify_and_map`.

Demo (presenter-only):

1. Plain server. Plaintext on the wire (tcpdump glance).
2. Add TLS server config. Wire is encrypted.
3. Add `tls.verify = true` and client certs. Server authenticates client by cert.
4. Add `verify_and_map`. Cert identity (SAN or CN) maps to a NATS user. Same identity model, PKI-backed.

What plain TLS/mTLS (steps 1-3) doesn't give you: per-subject permissions, revocation without restart, named identity in the audit trail. Step 4's `verify_and_map` is the bridge; the identity labs take it from there.

Labs 1-3 · Identity

static credentials -> JWT/nsc -> auth callout

Hands-on-keys: you run the `make` steps

Lab 1 · Static credentials

```
accounts {
  APP {
    users = [
      { user: alice, password: secret1 },
      { user: bob,   password: secret2,
        permissions: { publish: ["test.>"], subscribe: ["test.>"] }
      }
    ]
  }
}
```

Works when: small fixed deployments, embedded systems, single team.

Breaks when: humans rotate, multi-tenant, audit required.

Same model for NKey: replace `password` with cryptographic signature.

Demo: `make 00` (user/pass), then `make 00-nkey -- 00-baseline/`, ~6 min

Lab 2 · Decentralized JWT (nsc)

```
Operator --signs--> Account --signs--> User
    |                  |                 |
  NKey               NKey              NKey
```

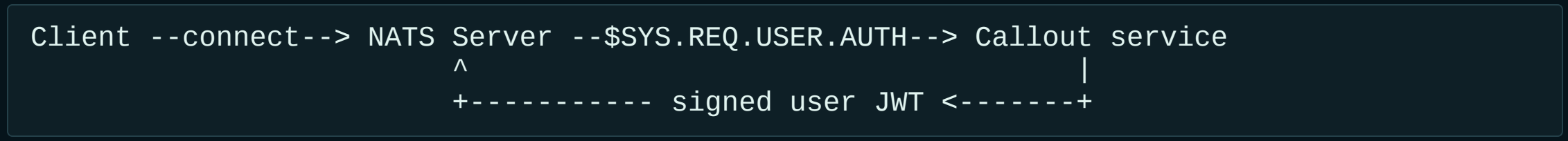
- Server holds only the **operator's** public key.
- Account JWTs distributed via the **NATS resolver itself**.
- Permissions, limits, expiry live in the user JWT.
- Rotation: `nsc revocations add-user` + repush.

Demo: `make 01`, then `make 01-decode` (live JWT claims) -- `01-jwt-nsc/`, ~8 min

Lab 3 · Auth callout

The why: identity comes from outside NATS -- your callout service consults any backend (OIDC, LDAP, a user table, a billing system) and decides.

The how (zero-trust, mutual): the server publishes the request on `$SYS.REQ.USER.AUTH`; the callout -- authenticated in its own account, and via xkey bound to *this* server -- returns a **signed** user JWT; the server then **verifies** that response before honoring it. Both sides validate the other.



Demo (hands-on-keys): run `make 02` then `make 02-flip-bob` -- watch *your* `bob` gain `app.>`, NATS server never restarting. `02-auth-callout/`, ~18 min

Auth callout: what it buys, what it costs

+ What callout buys

- Dynamic, per-connection permissions
- External IdP / IAM integration
- Per-connection limits from an external source
- No growing revocation lists -- revocation takes effect on next connection

- What it costs

- Another service to run (run it HA, queue-subscribed)
- Latency on every connection (connect-time, not per message)
- Callout is now critical-path security
- Tightening/revoking only bites on reconnect -- pair short JWT TTLs or force-disconnect for live cutoff

Managing it at scale: OSS vs SCP

Same scenario, two paths: onboard an app team -- account, three users, one 30-day rotation, audit trail.

- OSS: you own the machinery

- Cost: free
- Skill: `nsc` + git fluency
- Audit: build your own
- New admin: onboard in days
- CI/CD: native (it's git)
- Multi-region: custom resolver
- Self-service: you build it

+ SCP: managed workflow

- Cost: license
- Skill: UI literate
- Audit: built in
- New admin: onboard in hours
- CI/CD: API + UI
- Multi-region: built in
- Self-service: built in

Decision criterion: SCP when the people operating NATS identity are not the people who built the system. OSS when they are.

Lab 4 · Import into Control Plane

your nsc store, now managed

Bring-Your-Own-System import

Step 1: package the nsc store you built in Lab 2.

```
make 03          # tar 01-jwt-nsc/nsc-store -> 03-scp-import/nsc-store.tar.gz
make 03-decode   # print the operator/account JWTs + seeds for import
```

Step 2: import into the temp control plane. Same operator. Same accounts. Same users. Now with management UX, audit log, rotation buttons, and multi-region.

Talk-track payoff:

> Same primitives. Same security model. The difference is who can operate it without learning nsc.

Hardening checklist

1. **Deny-by-default permissions** -- open subs are the highest-leverage misconfig.
2. **Signing key rotation** -- issue user JWTs from dedicated signing keys; rotate them without touching account identity.
3. **$SYS account boundaries** -- treat $SYS like root; only a dedicated monitoring account may touch it.
4. **Leafnode isolation** -- pin each leaf to a scoped hub account; that account defines the trust boundary.
5. **Revocation hygiene** -- pick lists vs short-lived JWTs before you need to.
6. **Monitoring** -- `/healthz`, `$SYS.REQ.SERVER.PING`, log shipping.
7. **Connection naming** -- `name=app/ver (host=h) [purpose]` on every client.

Take it home

Everything you ran today is yours to keep and re-run:

- **Lab bundle:** `github.com/synadia-labs/nats-security-workshop`
- **Follow-along guide:** `LAB.md`, every step with its expected output
- **Hardening checklist:** the slide above; apply it Monday



scan to clone

Q & A + close

- Workshop demo bundle (clone, replay): `github.com/synadia-labs/nats-security-workshop`
- Open-source NATS: `nats.io`
- Auth-callout reference impls: Go `github.com/synadia-io/callout.go`, .NET `github.com/synadia-io/callout.net`
- Trial NATS / SCP: `synadia.com`

Thank you!