

asn_roas_import.py — Technical Documentation

1. Purpose & Role in the Platform

`asn_roas_import.py` is an **auxiliary data collection script** responsible for importing all relevant **RPKI ROAs (Route Origin Authorizations)** into the platform's main database.

This script does **not** compute security metrics itself. Instead, it provides **essential raw data** used by higher-level analytics, especially:

- **asn_filters_strict.py** — requires accurate ROA information to evaluate whether an ASN's announced prefixes conform to RPKI expectations, and to detect suspicious / misconfigured origins.

Thus, the role of this script is simple but critical:

populate the `roas` table with authoritative RPKI origin information, ensuring downstream security logic has complete and correct data to work with.

2. High-Level Overview

The script:

1. Connects to the platform's `asn_data.db` database.
2. Determines which prefixes exist in the system (from `prefix_data` or `bgp_updates`).
3. For each prefix, queries RIPEstat's **RPKI ROAs API** to fetch all VRPs (Valid ROA Payloads).
4. Normalizes and validates the RPKI data.

5. Inserts all ROAs into the `roas` table in SQLite.
6. Optionally sleeps between API calls to control rate-limiting.

This data is then consumed by the strict filtering metric (`asn_filters_strict`), which needs full ROA coverage to determine:

- Which ASNs are allowed to announce which prefixes.
 - Whether a given prefix-length exceeds `maxLength`.
 - Whether an observed origin matches the ROA-authorized ASN(s).
 - If an update is suspicious.
-

3. Database Schema & Setup

The script ensures a dedicated table:

```
CREATE TABLE IF NOT EXISTS roas (  
    prefix      TEXT NOT NULL,  
    asn         INTEGER NOT NULL,  
    max_length  INTEGER NOT NULL  
)
```

This format exactly matches the structure of VRPs:

- **prefix** — the ROA-authorized IPv4/IPv6 prefix
- **asn** — the authorized origin ASN
- **max_length** — maximum allowed prefix length in announcements

This enables efficient downstream lookups for strict origin validation.

Optional behavior:

- `--keep-old` → preserves existing ROA rows
 - default (no flag) → clears table before importing
-

4. Prefix Discovery Logic

Before collecting ROAs, the script must determine **which prefixes to query**.

It does this dynamically:

Case 1: `prefix_data` table exists

→ Use all known prefixes from the platform's own collectors.

Case 2: `prefix_data` absent

→ Fall back to `bgp_updates` table containing raw RIS events.

This logic ensures flexibility:

The script adapts automatically to whichever dataset is available in the installation.

You can also limit the number of prefixes processed via `--limit`.

5. Fetching ROAs from RIPEstat

For each prefix, the script calls:

```
https://stat.ripe.net/data/rpki-roas/data.json?resource=<prefix>&include=more-specifics
```

It handles:

- HTTP errors

- connection failures
- JSON decode errors
- malformed responses

Each response is parsed to extract VRPs, using any of the possible fields:

- `prefix`, `route`, or `roa_prefix`
- `asn`, `origin_asn`, or `origin`
- `maxLength`, `maxlength`, or `max_length`

If `maxLength` is missing, the script falls back to using the prefix's own length (RFC-consistent behavior).

All parsing logic is defensive and robust.

6. Inserting ROAs into SQLite

For each VRP (prefix, ASN, maxLength), the script inserts:

```
INSERT INTO roas(prefix, asn, max_length) VALUES (?, ?, ?)
```

This is committed immediately to ensure durability.

The script maintains counters:

- **total_vrps** — number of VRPs successfully imported
- **no_vrps** — prefixes with no associated ROAs
- **api_errors** — failed API calls

At the end it prints:

```
[roas] total rows in roas = X
```

allowing the operator to verify completeness.

7. Operational Controls

The script supports safe-run options:

--limit N

Restricts how many prefixes are queried. Useful for testing.

--sleep SECONDS

Sleeps between API calls to avoid hitting rate limits.

--keep-old

Preserves existing rows in the `roas` table.

--verbose

Enables detailed logging for debugging or audits.

8. Why This Script Is Required by `asn_filters_strict.py`

`asn_filters_strict.py` computes advanced routing-security metrics, including:

- strict origin correctness
- authorized vs unauthorized ASN announcements

- incorrect prefix-lengths
- presence or absence of ROA coverage
- determining if an ASN is vulnerable or misconfigured

For this, it *must* know:

- All ROA-authorized origins for every prefix
- All maxLengths per ROA
- Whether an observation contradicts RPKI policy

Without the ROA data imported by this script,

asn_filters_strict.py cannot make correct security classifications and vulnerability risk assessments.

Therefore, `asn_roas_import.py` is a **foundational ingestion component** of the routing-security pipeline.

9. Why the ROA Data Source Is Reliable

The `asn_roas_import.py` script fetches ROA information via **RIPEstat's RPKI ROAs API**, which aggregates data from the global RPKI trust anchor ecosystem (RIRs' published ROA repositories). This means the VRPs (Valid ROA Payloads) it returns are derived from the same underlying cryptographic objects that operational networks use for RPKI Origin Validation (ROV). RIPEstat is a production-grade service operated by RIPE NCC and is widely trusted in both research and operational communities. As a result, the imported `roas` table reflects the actual, authoritative origin-authorization state of the Internet, providing a robust basis for strict origin checks and vulnerability analysis in `asn_filters_strict.py`.

10. Summary

`asn_roas_import.py` is a lightweight but essential auxiliary script. It:

- discovers prefixes in the database
- fetches authoritative ROAs from RIPEstat
- normalizes and validates VRP data
- populates the ROA table
- supplies crucial origin-authorization data to the platform's main security engines

It ensures the strict filtering logic, vulnerability scoring, and hijack-detection modules operate on a complete and accurate RPKI dataset.