

OmniTWAG Architecture

Overview

OmniTWAG is a Trusted WLAN Access Gateway (TWAG) per 3GPP TS 23.402 Section 16. It bridges trusted WLAN access points into the Evolved Packet Core (EPC), providing authentication, session management, and data plane tunnelling.

Table of Contents

- [Component and Process Architecture](#)
- [Supervision Tree](#)
- [Component Reference](#)
- [Operating Modes](#)
- [AP Integration \(GRE Tunnel\)](#)
- [DHCP Behaviour](#)
- [Authentication](#)
- [Session Lifecycle \(Home-Routed\)](#)
- [Diameter Interfaces](#)
- [Configuration Summary](#)
- [Key Derivation](#)
- [Related Pages](#)

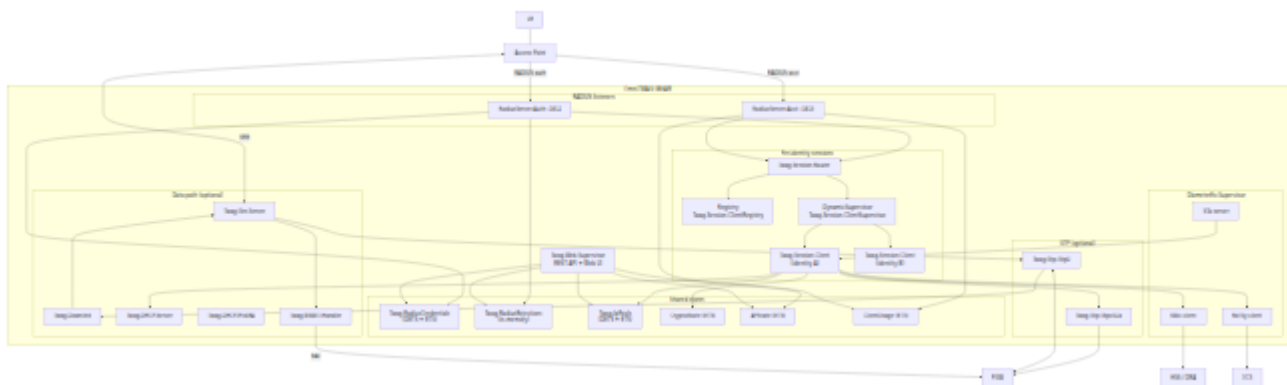
Component and Process Architecture

OmniTWAG runs as a single BEAM (Erlang VM) application. The top-level supervisor starts a fixed set of shared services and, based on config, a set of

optional data-path and interface children. Per-subscriber state lives in one GenServer for each identity, created on demand.

The main groups are:

- **Web and OAM:** the REST API and web interface (`Twag.Web.Supervisor`).
- **Per-identity sessions:** one `Twag.Session.Client` GenServer for each subscriber, found through a `Registry` and started by a `DynamicSupervisor`. `Twag.Session.Router` does the lookup-or-create.
- **Shared stores:** `Twag.RadiusCredentials` (durable shared secrets), `Twag.RadiusRejections` (in-memory rejected sources), `Twag.IpPools` (address pools), and the ETS-backed `CryptoState`, `APState`, and `ClientUsage` stores.
- **RADIUS listeners:** `RadiusServer.Auth` (port 1812) and `RadiusServer.Acct` (port 1813).
- **Diameter:** `DiameterEx.Supervisor` carries the SWx (auth vectors), STa (re-auth and abort), and Ro/Gy (online charging) peers.
- **GTP:** `Twag.Gtp.GtpcS2a` (control) and `Twag.Gtp.GtpU` (user plane), started only when `gtp.enabled` is true.
- **Data path:** `Twag.Gre.Server`, `Twag.Downlink`, `Twag.DHCP.Server`, `Twag.DHCP.IPv6RA`, and `Twag.NSW0.Handler`, each started only when its config block is enabled.



Supervision Tree

The root supervisor uses a `:one_for_one` strategy (max 10 restarts in 60 seconds). The children start in this order, so that per-source-IP secrets resolve before the first RADIUS packet arrives:



Children marked ☐ are started only when their config block is enabled (see [Configuration Summary](#)).

Component Reference

Component	Kind	Started when
<code>Twag.Web.Supervisor</code>	Supervisor	Always
<code>DiameterEx.Supervisor</code>	Supervisor	Always
<code>Twag.Session.ClientRegistry</code>	Registry	Always
<code>Twag.Session.ClientSupervisor</code>	DynamicSupervisor	Always
<code>Twag.Session.Router</code>	Module	On demand
<code>Twag.Session.Client</code>	GenServer	Per identity
<code>Twag.RadiusCredentials</code>	GenServer	Always

Component	Kind	Started when
<code>Twag.RadiusRejections</code>	GenServer	Always
<code>Twag.IpPools</code>	GenServer	Always
<code>RadiusServer.Auth</code>	GenServer	Always
<code>RadiusServer.Acct</code>	GenServer	Always
<code>CryptoState</code>	GenServer (ETS)	Always
<code>APState</code>	GenServer (ETS)	Always
<code>ClientUsage</code>	GenServer (ETS)	Always

Component	Kind	Started when
Twag.Gtp.GtpcS2a	GenServer	gtp.enabled
Twag.Gtp.GtpU	GenServer	gtp.enabled
Twag.Downlink	GenServer	downlink.enabled
Twag.Gre.Server	GenServer	gre.enabled
Twag.DHCP.Server	GenServer	dhcp.enabled
Twag.DHCP.IPv6RA	GenServer	ipv6_ra.enabled
Twag.NSW0.Handler	GenServer	nsw0.enabled

Operating Modes

OmniTWAG supports two data plane modes. The mode used for a session is resolved from the matching AP credential (see [RADIUS Shared Secrets](#)), with the global `gtp` / `nswo` config as the default.

Home-Routed Mode (GTP Tunnel)

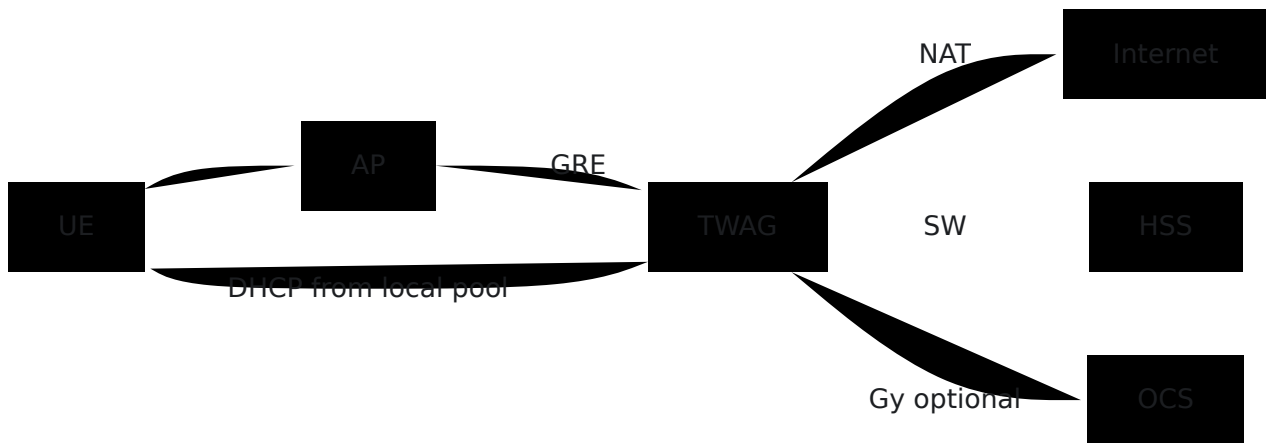
Traffic is tunnelled via GTP-C/GTP-U (S2a interface) to the PGW. The PGW allocates the UE IP address, applies policy/charging, and provides internet access.



- **UE IP:** Allocated by the PGW in Create Session Response (PAA IE).
- **IP delivery:** The TWAG serves DHCP to the UE over GRE, handing out the PGW-allocated IP.
- **Uplink:** The AP sends UE traffic via GRE. The TWAG encapsulates in GTP-U to the PGW.
- **Downlink:** The PGW sends via GTP-U. The TWAG decapsulates and sends via GRE to the AP, then the UE.
- **Charging:** Handled by the PGW (PCEF) via Gy to the OCS.
- **Config:** `gtp.enabled: true`, `gre.enabled: true`.

Local Breakout Mode (NSWO)

Traffic exits directly from the TWAG to the internet. No PGW is involved. The TWAG performs NAT/masquerade and optionally does its own online charging via Gy.



- **UE IP:** Allocated from a TWAG local IP pool (`Twag.IpPools`).
- **IP delivery:** The TWAG serves DHCP to the UE over GRE, from the pool.
- **Uplink:** The AP sends via GRE. The TWAG applies NAT to the internet.
- **Downlink:** The internet path returns through TWAG NAT, then GRE to the AP, then the UE.
- **Charging:** The TWAG acts as PCEF via Gy to the OCS (optional, disabled by default). See [Online Charging](#).
- **Config:** `nswo.enabled: true`, `gtp.enabled: false`.

Local Breakout is selected by setting `nswo.enabled: true` (with `gtp.enabled: false`), or per AP by the credential `breakout_type`. When enabled, the NSWO handler is started, sessions are given an address from a local pool instead of a PGW-assigned one, and uplink is NAT/masqueraded straight to the internet. See [User Plane and Data Path](#) for the data-path detail.

AP Integration (GRE Tunnel)

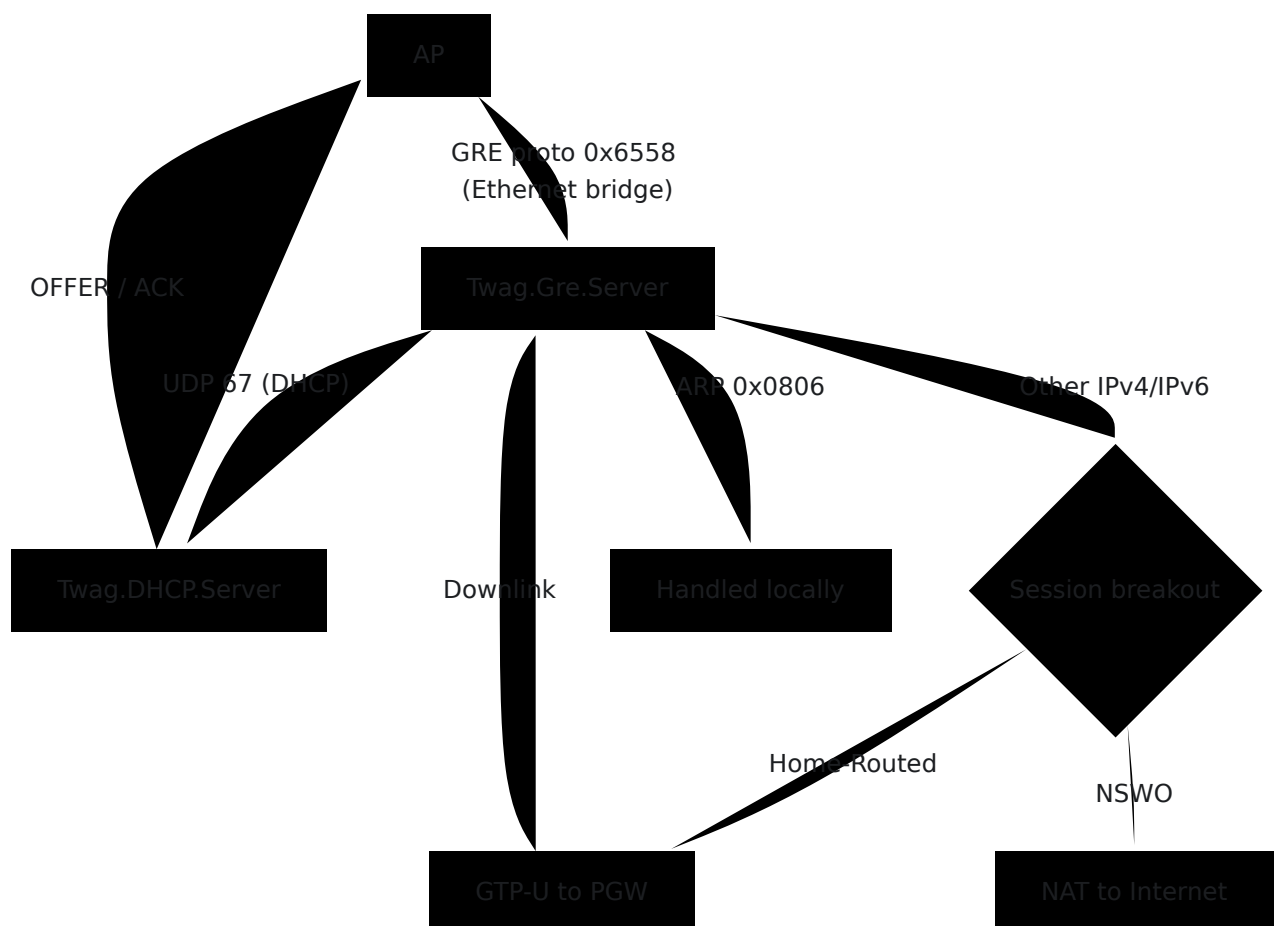
The AP tunnels UE L2 frames to the TWAG via GRE (IP protocol 47). This provides the required point-to-point link between UE and TWAG per TS 23.402 Section 16.1.2.

Cisco AP Configuration

```
dot11 ssid ons
  tunnel CWAG
  authentication open eap eap_methods
  authentication key-management wpa
  accounting acct_methods
```

```
dot11 tunnel CWAG
  type gre
  source {AP_IP}
  destination {TWAG_IP}
  mss 1410
  mtu 1500
```

GRE Data Flow



DHCP Behaviour

Mode	IP Source	DHCP Server	Notes
Home-Routed	PGW (PAA in Create Session Response)	TWAG pre-allocates the PGW IP, serves it via DHCP over GRE	DNS from PGW PCO
Local Breakout	TWAG local pool (<code>Twag.IpPools</code>)	TWAG allocates from the bound pool on DISCOVER (pool dish-out)	DNS from pool config

The TWAG DHCP server:

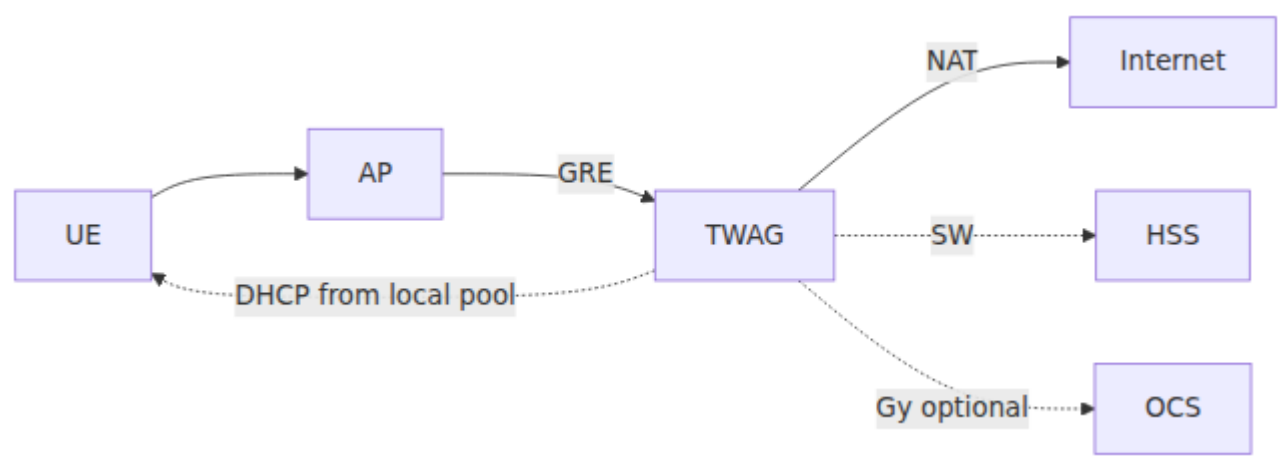
1. Receives a DHCP DISCOVER from the UE via the GRE tunnel.
2. Serves the OFFER. In home-routed mode this is the PGW-allocated IP that the session pre-allocated. In local-breakout mode, when the MAC is bound to a pool, the server dishes out a fresh address from the pool.
3. Handles REQUEST/ACK to complete the exchange.
4. Passes all DHCP traffic through the GRE tunnel to and from the AP.

Authentication

OmniTWAG acts as a combined TWAG and 3GPP AAA Server, performing EAP-AKA authentication locally and fetching auth vectors from the HSS via Diameter SWx. Each subscriber's EAP state machine runs inside its own

`Twag.Session.Client`.

Auth Flow



Before any packet is processed, the RADIUS listener authorizes the source IP against `Twag.RadiusCredentials`. A source with no matching credential subnet is dropped and recorded as `unauthorized_source`. A source that matches but fails the Message-Authenticator check is recorded as `bad_secret`. See [RADIUS Rejections](#).

Auth Modes

Mode	Config	Vector Source	EAP Processing
<code>:swx</code>	<code>auth_mode: :swx</code>	HSS via SWx MAR/MAA	Local (TWAG)
<code>:local</code>	<code>auth_mode: :local</code>	Local test keys (Milenage)	Local (TWAG)

EAP Method

Configurable via `eap_method`. Default `:eap_aka`. If the UE NAKs the offered method, the TWAG negotiates to the UE's preferred supported method. EAP-SIM and EAP re-authentication (ERP) are also supported. See [Authentication](#).

Session Lifecycle (Home-Routed)

EAP Auth -> Access-
Accept (MSK to AP)

AP sends Accounting-
Start

Post-accept on
Twag.Session.Client

GTP-C Create Session
Request -> PGW

PGW allocates IP,
Create Session
Response

Register GTP-U tunnel
(local + remote TEIDs)

Pre-allocate PGW IP in
DHCP for UE MAC

UE DHCP Discover ->

OFFER WITH PGW IP

Data flows: UE -> GRE -
> GTP-U -> PGW ->
Internet

Teardown: GTP-C Delete
Session, GTP-U + DHCP
cleanup

A session is torn down for one of several reasons, each recorded on the `twag.session.teardown` metric: `accounting_stop`, `admin_disconnect`, `network_disconnect` (STa ASR), `identity_timeout` (auth started but never completed), and `liveness_timeout`. See [Metrics Reference](#) and [Sessions](#).

Diameter Interfaces

Interface	App ID	Direction	Purpose
SWx	16777265	TWAG -> HSS	Auth vector fetch (MAR/MAA)
STa	16777250	Inbound to TWAG	RAR (re-auth), ASR (abort), STR (terminate)
Ro/Gy	4	TWAG -> OCS	Online charging (NSWO only)

DRA Routing

The DRA must route SWx to the HSS:

```
%{rule_name: "swx_to_hss", match: :all,  
  filters: [{:application_id, 16777265}, {:avp, {296, "  
{realm}"}}]},  
  route: %{peers: [{"hss_host"}]}}
```

Configuration Summary

Every parameter is documented in the table that follows the block.

```
config :omnitwag,  
  auth_mode: :swx,  
  eap_method: :eap_aka,  
  identity_timeout_ms: 600_000,  
  
  gtp: %{  
    enabled: true,  
    pgw_ip: "10.179.1.21",  
    local_ip: "10.179.1.41",  
    pgw_port: 2123,  
    gtpu_port: 2152,  
    echo_enabled: true  
  },  
  
  gre: %{  
    enabled: true,  
    local_ip: "10.179.1.41"  
  },  
  
  downlink: %{  
    enabled: true  
  },  
  
  dhcp: %{  
    enabled: true,  
    interface: "gre"  
  },  
  
  ipv6_ra: %{  
    enabled: false  
  },  
  
  nswo: %{  
    enabled: false,  
    local_pool: "10.100.0.0/16",  
    nat_interface: "eth0"  
  },  
  
  online_charging: %{  
    mode: :disabled  
  },  
  
  rate_limit: %{
```

```
    max_failures: 10,  
    window_ms: 60_000  
  },  
  
  prometheus: %{\br/>    port: 9568  
  }
```

Parameter	Default	Description
<code>auth_mode</code>	<code>:swx</code>	<code>:swx</code> fetches vectors from the HSS; <code>:local</code> uses local Milenage test keys.
<code>eap_method</code>	<code>:eap_aka</code>	Offered EAP method (<code>:eap_aka</code> or <code>:eap_aka_prime</code>).
<code>identity_timeout_ms</code>	<code>600000</code>	Time a session may stay at the identity phase before it is torn down (teardown reason <code>identity_timeout</code>).
<code>gtp.enabled</code>	<code>true</code>	Start the GTP-C/GTP-U children (home-routed mode).
<code>gtp.pgw_ip</code>	<code>127.0.0.1</code>	PGW address for GTP-C/GTP-U.
<code>gtp.local_ip</code>	<code>none</code>	Local TWAG address for GTP.
<code>gtp.pgw_port</code>	<code>2123</code>	PGW GTP-C port.
<code>gtp.gtpu_port</code>	<code>2152</code>	Local GTP-U port.
<code>gtp.echo_enabled</code>	<code>true</code>	Send GTP echo keepalives to the PGW.

Parameter	Default	Description
<code>gre.enabled</code>	<code>false</code>	Start the GRE server for AP tunnels.
<code>gre.local_ip</code>	<code>0.0.0.0</code>	Local address the GRE server binds.
<code>downlink.enabled</code>	<code>true</code>	Start the downlink forwarder.
<code>dhcp.enabled</code>	<code>false</code>	Start the DHCPv4 server.
<code>dhcp.interface</code>	<code>none</code>	<code>"gre"</code> for tunnel mode, or a NIC name for direct delivery.
<code>ipv6_ra.enabled</code>	<code>false</code>	Start IPv6 Router Advertisements and DHCPv6.
<code>nsw.enabled</code>	<code>false</code>	Start the NSWO local-breakout handler.
<code>nsw.local_pool</code>	<code>none</code>	Fallback CIDR for local-breakout addressing.
<code>nsw.nat_interface</code>	<code>none</code>	Egress NIC for NAT/masquerade.
<code>online_charging.mode</code>	<code>:disabled</code>	Gy mode: <code>:disabled</code> ,

Parameter	Default	Description
		:authorize_only, or :online_charging.
rate_limit.max_failures	10	Auth failures per NAS in the window before Access-Requests are dropped.
rate_limit.window_ms	60000	Rate-limit window in milliseconds.
prometheus.port	9568	Prometheus metrics port.
radius_config.acct_interim_interval	300	Interim accounting interval (seconds) advertised to the AP in Access-Accept.

Key Derivation

For EAP-AKA' (per TS 33.402 Annex A.2 and RFC 5448):

```
HSS provides: RAND, AUTN, XRES, CK, IK
TWAG derives: CK' = KDF(CK||IK, FC=0x20, network_name, SQN(+)AK)
               IK' = (second half of KDF output)
               MK  = PRF'(IK'||CK', "EAP-AKA'" || Identity)
               MSK, EMSK, K_aut from MK
```

For EAP-AKA (RFC 4187):

```
HSS provides: RAND, AUTN, XRES, CK, IK
TWAG derives: MK = SHA1(Identity || IK || CK)
               K_encr, K_aut, MSK, EMSK from PRF(MK)
```

Network name format per TS 24.302:

```
WLAN:mnc{MNC}.mcc{MCC}.3gppnetwork.org
```

Related Pages

- [Authentication](#)
- [RADIUS Shared Secrets](#)
- [RADIUS Rejections](#)
- [IP Pools](#)
- [Sessions](#)
- [Per-AP Accounting](#)
- [User Plane and Data Path](#)
- [Online Charging](#)
- [Metrics Reference](#)

[Overview](#)

Authentication (EAP-AKA / EAP-AKA' / EAP-SIM)

Overview

OmniTWAG authenticates each subscriber with an EAP method that runs over RADIUS between the access point and the TWAG. The TWAG acts as a combined TWAG and 3GPP AAA Server. It runs the EAP state machine, and it gets the authentication vectors from the HSS over Diameter SWx, or from local test keys. On success, the TWAG delivers the Master Session Key (MSK) to the access point as MPPE keys inside the RADIUS Access-Accept. The access point then completes the 802.11 4-way handshake with the subscriber.

For the wider system context, see [Architecture](#). For how the RADIUS shared secret is resolved per access point, see [RADIUS Shared Secrets](#). Authentication runs first. The per-AP breakout and charging policy is applied after authentication succeeds. For the authenticated session and its status lifecycle, see [Sessions](#). For the S2a GTP-C session that the TWAG creates after authentication, see [S2a GTP-C](#). For charging that follows a successful authentication, see [Online Charging \(Gy\)](#).

Table of Contents

- [Overview](#)
- [Supported EAP Methods](#)
- [EAP State Machine and Phases](#)
- [Authentication Sequence](#)
- [Auth Vector Sourcing](#)
- [Identity and NAI Handling](#)
- [Key Derivation and Delivery](#)
- [Resynchronisation](#)

- Fast Re-Authentication (ERP)
- Failure Handling and Result Codes
- Configuration
- Observability
- Troubleshooting

Overview

The access point starts an IEEE 802.1X EAP exchange with the subscriber. The access point wraps each EAP message in a RADIUS `EAP-Message` attribute and sends it to the TWAG. The TWAG runs one authentication state machine per subscriber identity. The state machine drives the EAP exchange to a result, then returns one of three outcomes to the RADIUS layer.

Outcome	RADIUS reply	Meaning
Challenge	<code>Access-Challenge</code>	The exchange needs another EAP round.
Accept	<code>Access-Accept</code>	Authentication succeeded. The reply carries the MSK as MPPE keys.
Reject	<code>Access-Reject</code>	Authentication failed. The reply carries an EAP-Failure.



The TWAG holds crypto state per identity. It keeps the expected response (XRES), the authentication key (`Kaut`), the RAND, the MSK, and the extended MSK (EMSK) between EAP rounds. After the TWAG delivers the MPPE keys, the TWAG zeros the sensitive key material.

Supported EAP Methods

The TWAG supports three EAP methods. Each method uses the same TLV attribute format, but a different EAP type value and a different key hierarchy.

Method	EAP type	Specification	Vectors	K _{aut}	MAC	Key derivation
EAP-AKA'	50 (0x32)	RFC 5448	UMTS quintet	32 bytes	HMAC-SHA-256, truncated to 16 bytes	CK' / I PRF' (SHA-2
EAP-AKA	23 (0x17)	RFC 4187	UMTS quintet	16 bytes	HMAC-SHA1, truncated to 16 bytes	MK = SHA1(IK then f 2 PRF
EAP-SIM	18 (0x12)	RFC 4186	2 or 3 GSM triplets	16 bytes	HMAC-SHA1, truncated to 16 bytes	MK = SHA1(Kc. NONCE versi FIPS 1 PRF

Method selection and negotiation

The TWAG offers a default method. Set the method with the `eap_method` configuration key. The default is `:eap_aka_prime` (EAP-AKA').

The subscriber can reject the offered method with an EAP-NAK (EAP type 3). The NAK lists the EAP types that the subscriber prefers. The TWAG reads the list and selects the first type that it supports. The TWAG supports these types in the NAK: 50 (EAP-AKA'), 23 (EAP-AKA), and 18 (EAP-SIM).

- If the NAK lists a supported type, the TWAG restarts the exchange with that method. For EAP-AKA' and EAP-AKA, the TWAG sends an AKA-Identity request with `AT_PERMANENT_ID_REQ`. For EAP-SIM, the TWAG sends an EAP-SIM/Start request.
- If the NAK lists no supported type, the TWAG rejects the subscriber.

EAP State Machine and Phases

The TWAG runs one state machine per subscriber identity, inside a per-identity client process. The state machine tracks an EAP phase. The phase drives which EAP message the TWAG expects next and which telemetry it emits.

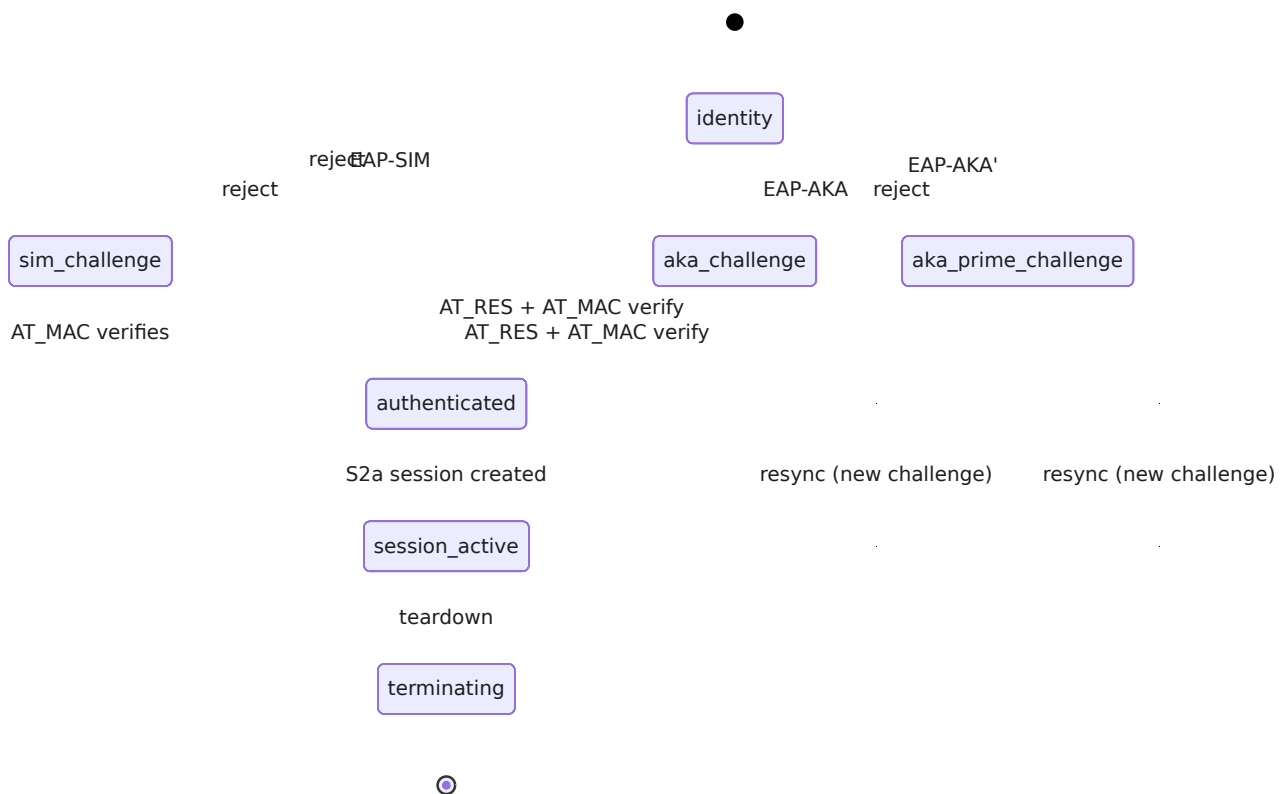
Phase (<code>eap_phase</code>)	Meaning
<code>identity</code>	The initial phase. The TWAG expects an EAP-Identity, or the first AKA-Identity or EAP-SIM/Start round. A failed round returns the state to <code>identity</code> .
<code>aka_challenge</code>	The TWAG sent an EAP-AKA challenge and waits for the AKA-Challenge response.
<code>aka_prime_challenge</code>	The TWAG sent an EAP-AKA' challenge and waits for the AKA-Challenge response.
<code>sim_challenge</code>	The TWAG sent an EAP-SIM challenge and waits for the SIM/Challenge response.
<code>authenticated</code>	The EAP exchange succeeded. The TWAG has the MSK and is about to establish the S2a session.
<code>session_active</code>	The S2a session is up. The subscriber can pass user data. RADIUS accounting can also promote a session to this phase.

Two further phase values exist in the code:

- `resyncing` is reserved as an in-progress authenticating phase. In practice a resynchronisation round rebuilds the challenge and returns the phase to `aka_challenge` or `aka_prime_challenge`, so the state does not settle on `resyncing`.
- `terminating` marks a session that is tearing down.

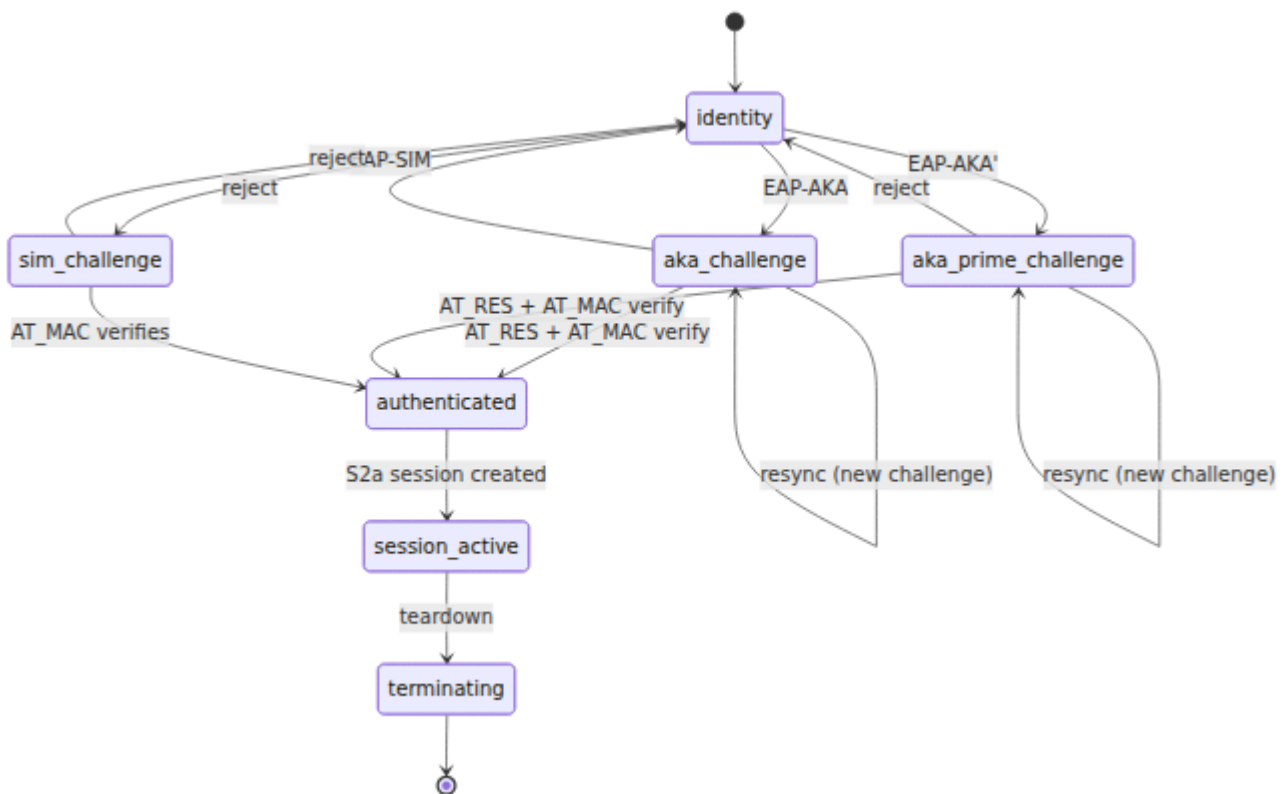
A session that stays in an in-progress authenticating phase (`identity`, `aka_challenge`, `aka_prime_challenge`, `sim_challenge`, or `resyncing`) past the identity timeout is reaped. Set the timeout with `identity_timeout_ms`. See [Configuration](#).

The [Sessions](#) page maps these phases to the session authentication status shown in the UI and the API.



Authentication Sequence

The full sequence has an identity phase and a challenge phase. The example below shows EAP-AKA'. EAP-AKA is the same shape. EAP-SIM uses a Start round in place of the AKA-Identity round, and it does not send `AT_RES`.



Identity phase

1. The subscriber sends an EAP-Identity response that carries the Network Access Identifier (NAI).
2. For EAP-AKA' or EAP-AKA, the TWAG replies with an AKA-Identity request that carries `AT_PERMANENT_ID_REQ`. This asks the subscriber for the permanent identity.
3. For EAP-SIM, the TWAG replies with an EAP-SIM/Start request that carries the `AT_VERSION_LIST` (version 1 only).
4. The subscriber returns the permanent identity (AKA-Identity with `AT_IDENTITY`), or the Start response with `AT_NONCE_MT` for EAP-SIM.

Challenge phase

1. The TWAG extracts the IMSI from the identity and gets the authentication vectors. See [Auth Vector Sourcing](#).
2. The TWAG derives the session keys, including the MSK. See [Key Derivation and Delivery](#).
3. The TWAG builds the challenge. For EAP-AKA' the challenge carries `AT_RAND`, `AT_AUTN`, `AT_KDF_INPUT` (the network name), `AT_KDF`,

`AT_BIDDING`, `AT_RESULT_IND`, and `AT_MAC`. For EAP-AKA the challenge carries `AT_RANDOM`, `AT_AUTN`, and `AT_MAC`. For EAP-SIM the challenge carries `AT_RANDOM` (2 or 3 RANDs) and `AT_MAC`.

4. The subscriber verifies the network (AUTN) and the MAC, then returns the response.
5. The TWAG verifies the response. See [Response verification](#).
6. On success, the TWAG returns an EAP-Success and the MSK. On failure, the TWAG returns an EAP-Failure.

Response verification

The TWAG verifies the challenge response in a fixed order. The order is the same for EAP-AKA and EAP-AKA'.

1. **State check.** If the stored `K_aut` or XRES is missing, the TWAG rejects.
2. **Attribute check.** If `AT_RES` or `AT_MAC` is missing, the TWAG rejects.
3. **MAC check.** The TWAG zeros the `AT_MAC` field, recomputes the MAC over the packet with `K_aut`, and compares. EAP-AKA uses HMAC-SHA1. EAP-AKA' uses HMAC-SHA-256. Both truncate to 16 bytes.
4. **Response check.** The TWAG compares `AT_RES` with the stored XRES. The comparison is **constant-time** to prevent timing attacks. The TWAG uses a length check and a constant-time byte compare.
5. **Result.** If both the MAC and the response match, the TWAG accepts. Otherwise the TWAG rejects.

For EAP-SIM, the subscriber does not send `AT_RES`. The `AT_MAC` check is the sole proof that the subscriber holds the correct keys. If the MAC verifies, the TWAG accepts.

Auth Vector Sourcing

The TWAG gets the authentication vectors from one of two sources. Set the source with the `auth_mode` configuration key.

Mode	auth_mode	Vector source	Use
SWx	:swx	HSS, over Diameter SWx (MAR/MAA)	Production
Local	any value other than :swx (default :local)	Local Milenage from provisioned test keys	Testing

In both modes, the TWAG runs the EAP state machine locally. Only the source of the vector differs. Any `auth_mode` value that is not `:swx` selects the local Milenage path.

STa pass-through is a separate path. A `Twag.Auth.StaAuth` provider can relay raw EAP payloads to a 3GPP AAA Server over Diameter STa (DER/DEA), so the AAA Server runs the EAP-AKA exchange and returns the MSK. This provider exists but is not on the live RADIUS authentication path. The live path fetches vectors with `:swx` or `:local` and runs the EAP state machine on the TWAG.

SWx to the HSS

In `:swx` mode, the TWAG sends a Multimedia-Auth-Request (MAR) to the HSS over the SWx interface (3GPP TS 29.273, Application ID 16777265). The MAR carries:

- `User-Name`: the IMSI.
- `SIP-Auth-Data-Item` with `SIP-Authentication-Scheme` set to `EAP-AKA`. For a resynchronisation, the item also carries the `SIP-Authorization` (the resync information: RAND concatenated with AUTS).
- `SIP-Number-Auth-Items`: 1.
- `ANID`: `WLAN`.
- `RAT-Type`: 0.
- `Auth-Session-State`: 1 (NO_STATE_MAINTAINED).
- `Vendor-Specific-Application-Id` with the SWx Auth-Application-Id 16777265 and Vendor-Id 10415.

The HSS returns a Multimedia-Auth-Answer (MAA). When the `Result-Code` is `2001` (success), the TWAG reads the first `SIP-Auth-Data-Item` and extracts the vector:

- `SIP-Authenticate` = RAND (16 bytes) concatenated with AUTN (16 bytes).
- `SIP-Authorization` = XRES.
- `Confidentiality-Key` = CK.
- `Integrity-Key` = IK.

If the `Result-Code` is not `2001`, or the vector fields are missing, the TWAG fails the fetch and treats it as an authentication denial (not an internal fault). The challenge phase rejects the subscriber cleanly with an EAP-Failure.

SWx error signalling (Experimental-Result)

SWx signals most errors in a grouped `Experimental-Result` AVP (Vendor-Id `10415`), not in the base `Result-Code`. A failed MAA can therefore carry no base `Result-Code` at all. The TWAG reads the `Experimental-Result-Code` as well, and it logs the human-readable name so the operator sees the real cause. See [Troubleshooting](#) for the codes.

A common case is `DIAMETER_ERROR_USER_UNKNOWN` (5001) or `DIAMETER_ERROR_USER_NO_NON_3GPP_SUBSCRIPTION` (5450): the subscriber is not enabled for non-3GPP access on SWx, so authentication fails even though the same subscriber may be known on S6a for EPC access.

The DRA must route the SWx application to the HSS. See [Architecture: DRA Routing](#).

Local Milenage

In `:local` mode, the TWAG generates the vector with the Milenage algorithm ([3GPP TS 35.206](#)) from the provisioned test keys (`test_keys`). The keys are `ki`, `opc`, `amf`, and `sqn`. The TWAG computes RAND, AUTN, XRES, CK, and IK in process. Use this mode for lab tests only.

Vectors for EAP-SIM

EAP-SIM needs 2 or 3 GSM triplets. The TWAG derives the triplets from the same Milenage vectors. For each vector the TWAG takes RAND, takes the first 4 bytes of XRES as SRES, and derives the GSM `Kc` from CK and IK with the c3 conversion function (3GPP TS 33.102 Annex C.3): `Kc = CK[0..7] XOR CK[8..15] XOR IK[0..7] XOR IK[8..15]`.

Identity and NAI Handling

The subscriber identity is a Network Access Identifier (NAI). The NAI has the form `{username}@{realm}`, for example

`0{IMSI}@wlan.mnc{MNC}.mcc{MCC}.3gppnetwork.org`.

IMSI extraction

The TWAG extracts the IMSI from the NAI in two steps:

1. The TWAG takes the username, that is the part before the `@`.
2. The TWAG removes a leading identity-type digit. A leading `0` (permanent identity, EAP-AKA/AKA') or a leading `6` maps to the bare IMSI. The remaining digits are the IMSI.

The TWAG accepts an IMSI of 14 or 15 digits. The IMSI is the `User-Name` in the SWx MAR. The TWAG also uses the IMSI to derive the MCC and the MNC for the GTP-C session: the MCC is the first 3 digits, and the MNC is the next 2 digits.

Network name (EAP-AKA')

EAP-AKA' binds the keys to the network name. The network name goes in `AT_KDF_INPUT`, and it is an input to the `CK'/IK'` derivation. Set the network name with the `network_name` configuration key. If the key is not set, the TWAG derives the network name from the Diameter realm per 3GPP TS 24.302:

- The TWAG removes a leading `epc.` from the realm.
- The TWAG prepends `WLAN:`.

For example, the realm `epc.mnc{MNC}.mcc{MCC}.3gppnetwork.org` gives the network name `WLAN:mnc{MNC}.mcc{MCC}.3gppnetwork.org`. If the realm has no `epc.` prefix, the TWAG uses `WLAN`.

Key Derivation and Delivery

Each method derives a Master Session Key (MSK) of 64 bytes and an Extended MSK (EMSK) of 64 bytes. The TWAG delivers the MSK to the access point. The TWAG keeps the EMSK for fast re-authentication.

EAP-AKA' key hierarchy

Per [RFC 5448](#) and [3GPP TS 33.402](#) Annex A:

1. `Key = CK || IK`.
2. `S = FC(0x20) || network_name || len(network_name) || (SQN XOR AK) || len(SQN XOR AK)`.
3. `CK' || IK' = HMAC-SHA-256(Key, S)`. `CK'` is the first 16 bytes. `IK'` is the last 16 bytes.
4. `MK = PRF'(IK' || CK', "EAP-AKA'" || Identity)`, 208 bytes. PRF' uses HMAC-SHA-256.
5. The TWAG splits `MK`: `K_encr` (16), `K_aut` (32), `K_re` (32), `MSK` (64), `EMSK` (64).

EAP-AKA key hierarchy

Per [RFC 4187](#):

1. `MK = SHA1(Identity || IK || CK)`.
2. The TWAG expands `MK` to 160 bytes with the FIPS 186-2 PRF.
3. The TWAG splits the output: `K_encr` (16), `K_aut` (16), `MSK` (64), `EMSK` (64).

EAP-SIM key hierarchy

Per [RFC 4186](#):

1. $MK = \text{SHA1}(\text{Identity} \parallel Kc1 \parallel Kc2 \parallel [Kc3] \parallel \text{NONCE_MT} \parallel \text{Version_List} \parallel \text{Selected_Version})$.
2. The TWAG expands `MK` to 160 bytes with the FIPS 186-2 PRF.
3. The TWAG splits the output: `K_encr` (16), `K_aut` (16), `MSK` (64), `EMSK` (64).

MSK to MPPE to the access point

The TWAG delivers the 64-byte MSK to the access point in the RADIUS Access-Accept. The TWAG splits the MSK and encodes each half as a Microsoft MPPE key ([RFC 2548](#), [RFC 3748](#)):

- `MS-MPPE-Recv-Key` = `MSK[0..31]` (first 32 bytes).
- `MS-MPPE-Send-Key` = `MSK[32..63]` (last 32 bytes).

The TWAG encrypts each key with the RADIUS shared secret and the Request Authenticator, per RFC 2548. The TWAG uses a fresh 2-octet salt with the high bit set for each key. The TWAG signs the Access-Accept with the shared secret. See [RADIUS Shared Secrets](#) for how the TWAG resolves the secret.

The access point decrypts the MPPE keys, recovers the MSK, and uses the MSK as the Pairwise Master Key for the 802.11 4-way handshake with the subscriber.

The Access-Accept requires a valid 64-byte MSK. If the MSK is missing or has the wrong size, the TWAG sends an Access-Reject instead.

Resynchronisation

The subscriber can report that its sequence number (SQN) is out of sync with the network. The subscriber sends an AKA-Synchronization-Failure (subtype `0x04`) with `AT_AUTS`. `AT_AUTS` is 14 bytes: 6 bytes of concealed SQN and 8 bytes of MAC-S.

The TWAG handles the resync as follows:

1. The TWAG builds the resync information: the stored RAND concatenated with the 14-byte AUTS.

2. The TWAG runs the resynchronisation. This recovers the subscriber SQN with the Milenage f_5^* function, verifies MAC-S, and returns an updated sequence number.
3. On success, the TWAG generates a new vector with the updated SQN, builds a new challenge, and sends a new Access-Challenge.
4. On failure, the TWAG rejects the subscriber with an EAP-Failure.

The TWAG handles resync for both EAP-AKA and EAP-AKA'. Resync uses the local Milenage keys. The `AT_AUTS` value must be exactly 14 bytes, otherwise the TWAG raises an error and the round fails.

Fast Re-Authentication (ERP)

The TWAG supports the EAP Re-authentication Protocol (ERP) per [RFC 6696](#). ERP lets a subscriber re-authenticate without a full EAP exchange. The saving comes from a key chain that the TWAG derives from the EMSK during the first full authentication.

After a full authentication, the TWAG derives and stores the re-authentication keys:

- `rRK` = re-authentication Root Key, from the EMSK.
- `rIK` = re-authentication Integrity Key, from the rRK.

All ERP key derivations use HMAC-SHA-256.

When the subscriber sends an EAP-Initiate/Re-auth message (EAP type 41), the TWAG does this:

1. If the TWAG has no stored `rRK` or `rIK`, the TWAG falls back to a full authentication. The TWAG sends a fresh AKA-Identity request.
2. If the TWAG has the keys, the TWAG parses the message and verifies the authentication tag with `rIK`.
3. If the tag verifies, the TWAG derives the re-authentication MSK (`rMSK`) from `rRK` and the message sequence number, builds an EAP-Finish/Re-auth response (EAP type 42), and accepts. The `rMSK` becomes the new MSK for the MPPE keys.

4. If the tag does not verify, or the parse fails, the TWAG falls back to a full authentication.

Failure Handling and Result Codes

EAP outcomes

Condition	TWAG action
SWx MAR fails, or vectors are incomplete	The vector fetch is denied. The TWAG rejects the subscriber cleanly with an EAP-Failure, and logs a denial rather than a crash.
AT_MAC verification fails	Access-Reject with EAP-Failure.
AT_RES does not match XRES	Access-Reject with EAP-Failure.
Missing K_aut or XRES state	Access-Reject with EAP-Failure.
Missing AT_RES or AT_MAC attribute	Access-Reject with EAP-Failure.
Subscriber sends AKA-Authentication-Reject (subtype 0x02)	Access-Reject. The subscriber rejected the network.
Subscriber sends AKA-Client-Error (subtype 0x0E)	Access-Reject.
EAP-NAK with no supported method	Access-Reject.
MSK missing or not 64 bytes at Accept	Access-Reject.
EAP message not handled	Access-Reject with EAP-Failure.

SWx Diameter result codes

The success code is a base `Result-Code`. The error codes below arrive in the `Experimental-Result` AVP, not the base `Result-Code`. See the [SWx error signalling](#) note and [Troubleshooting](#).

Code	Name	Meaning
2001	DIAMETER_SUCCESS	The MAA carries a valid vector
5001	DIAMETER_ERROR_USER_UNKNOWN	The HSS has no non-3GPP/SWx subscription for this IMSI. The subscriber may still be known on S6a/EPC.
5004	DIAMETER_ERROR_NOT_SUPPORTED_USER_DATA	The requested user data is not supported
5420	DIAMETER_ERROR_UNKNOWN_EPS_SUBSCRIPTION	No EPS subscription for this subscriber
5450	DIAMETER_ERROR_USER_NO_NON_3GPP_SUBSCRIPTION	The subscriber has no non-3GPP subscription
Other	(Diameter error)	The vector fetch fails.

Code	Name	Meaning
		The round rejects.

Reference: [RFC 6733 Section 7.1](#), [3GPP TS 29.273](#), [3GPP TS 29.229](#).

Network-initiated events (STa)

The 3GPP AAA Server can send events to the TWAG over the STa interface ([3GPP TS 29.273](#), Application ID 16777250). The TWAG answers each event with Result-Code 2001.

Message	Trigger	TWAG action
ASR (Abort-Session-Request)	The AAA Server wants a disconnect.	The TWAG disconnects the subscriber session, then answers ASA.
STR (Session-Termination-Request)	HSS-initiated detach.	The TWAG disconnects the subscriber session, then answers STA.
RAR (Re-Auth-Request)	The AAA Server wants a re-authentication.	The TWAG answers RAA with success and records the request. Full re-authentication of the live session is not yet triggered from RAR.

All three answers carry the Origin-Host and Origin-Realm from the Diameter identity. See [Configuration](#).

Configuration

Set the authentication parameters in the `:omnitwag` application configuration. The Diameter identity (`host` and `realm`) comes from the `:diameter_ex`

configuration, and the TWAG uses it as the Origin-Host and Origin-Realm on SWx and STa.

```
config :omnitwag,  
  # Vector source: :swx (production) or :local (testing)  
  auth_mode: :swx,  
  
  # Offered EAP method: :eap_aka_prime (default), :eap_aka, or  
  :eap_sim  
  eap_method: :eap_aka_prime,  
  
  # EAP-AKA' network name. If unset, derived from the Diameter  
  realm.  
  network_name: "WLAN:mnc001.mcc001.3gppnetwork.org",  
  
  # Reap a session still stuck in an in-progress auth phase after  
  this many  
  # milliseconds. Default 600000 (10 minutes).  
  identity_timeout_ms: 600_000,  
  
  # TWAN-Identifier fields, added to STa DER/AAR when set  
  twan_ssid: "ons",  
  twan_bssid: "00:11:22:33:44:55",  
  
  # Local test keys (used only when auth_mode: :local)  
  test_keys: %{\br/>    ki: "11111111111111111111111111111111",  
    opc: "22222222222222222222222222222222",  
    amf: "8000",  
    sqn: 1200  
  }
```

Parameter	Type	Required	Default	Description
<code>auth_mode</code>	Atom	No	<code>:local</code>	Vector source. <code>:swx</code> gets vectors from HSS over <code>:local</code> generates vectors from <code>test_key</code> <code>:swx</code> in production
<code>eap_method</code>	Atom	No	<code>:eap_aka_prime</code>	The offered method. <code>:eap_aka</code> <code>:eap_aka</code> <code>:eap_sim</code> subscriber negotiate different supported method with EAP-NAK.
<code>network_name</code>	String	No	derived from realm	The EAP-AKA network name that goes <code>AT_KDF_ID</code> unset, the derives it the Diameter realm per 24.302.
<code>identity_timeout_ms</code>	Integer	No	<code>600000</code>	Time in milliseconds

Parameter	Type	Required	Default	Description
				a session in an in-pr auth phase before the reaps it.
<code>twan_ssid</code>	String	No	-	The WLAN When set, TWAG add TWAN-Ide to STa DEI AAR mess
<code>twan_bssid</code>	String	No	-	The acces BSSID. Combined <code>twan_ssid</code> form the T Identifier.
<code>test_keys</code>	Map	No	built-in defaults	Local Mile keys. Used when <code>aut</code> is <code>:local</code> Test key paramete

Test key parameters

The TWAG reads `test_keys` only in `:local` mode. Do not use `test_keys` in production.

Parameter	Type	Required	Default	Description
<code>ki</code>	String (hex)	Yes (local mode)	<code>1111...</code>	Subscriber key <code>Ki</code> , 16 bytes as 32 hex characters.
<code>opc</code>	String (hex)	Yes (local mode)	<code>2222...</code>	Operator variant key <code>OPc</code> , 16 bytes as 32 hex characters.
<code>amf</code>	String (hex)	Yes (local mode)	<code>8000</code>	Authentication Management Field, 2 bytes as 4 hex characters.
<code>sqn</code>	Integer	Yes (local mode)	<code>1200</code>	The starting sequence number.

Observability

The TWAG emits telemetry counters through the EAP state machine and the STa handler. The Prometheus endpoint exposes the counters. See the [Metrics Reference](#) for the endpoint and query examples.

EAP metrics

Metric	Meaning
<code>eap_aka_identity_count</code>	EAP-AKA/AKA' Identity exchanges (Phase 1).
<code>eap_aka_challenge_count</code>	EAP-AKA/AKA' Challenge exchanges (Phase 2).
<code>eap_aka_sync_failure_count</code>	AKA synchronisation failures (resync events).
<code>eap_aka_auth_success_count</code>	Successful EAP-AKA/AKA' authentications.
<code>eap_aka_auth_reject_count</code>	Rejected EAP-AKA/AKA' authentications.

The TWAG also emits telemetry events for EAP-SIM (`start`, `auth_success`, `auth_reject`), for AKA `auth_reject` and `client_error`, and for ERP (`initiate`, `success`, `fallback`, `mac_failure`). See the [Metrics Reference](#).

RADIUS reply metrics

Metric	Meaning
<code>radius_access_challenge_count</code>	Access-Challenge replies (EAP round in progress).
<code>radius_access_accept_count</code>	Access-Accept replies (successful authentication).
<code>radius_access_reject_count</code>	Access-Reject replies (failed authentication).

STa event metrics

The STa handler emits a telemetry counter for each network-initiated event:

`[ :sta, :asr, :received, :count ]`, `[ :sta, :str, :received, :count ]`, and `[ :sta, :rar, :received, :count ]`, plus session `disconnect` and `reauth` counters.

Log context

The TWAG sets the log metadata `identity` and `imsi` for each session. Use these fields to trace one subscriber through the authentication flow.

Troubleshooting

Symptom	Likely cause	
Auth fails and the log shows DIAMETER_ERROR_USER_UNKNOWN (5001)	The HSS has no non-3GPP/SWx subscription for this IMSI. The subscriber may be known on S6a/EPC but is not enabled for non-3GPP access.	Enable subscriber subscription in HSS.
Auth fails and the log shows DIAMETER_ERROR_USER_NO_NON_3GPP_SUBSCRIPTION (5450)	The subscriber profile has no non-3GPP subscription.	Provision non-3GPP subscription in the HSS.
Log shows Result-Code=nil on a failed MAA	SWx returned an error in the Experimental-Result AVP, not the base Result-Code.	Read the Experimental-Result from the TWAG human name.

Symptom	Likely cause	
Auth never completes and the session is reaped	The subscriber stalled in an in-progress auth phase past the identity timeout.	Check the expected identity if needed.
Subscriber sends AKA-Authentication-Reject	The UE failed to authenticate the network (AUTN check).	Check network OP/OPc match : local the HS correct
Repeated AKA-Synchronization-Failure	The subscriber SQN drifted from the network SQN.	The TV resync. If it per the HS handling. Resync
Access-Reject with a valid EAP-Success expected	The MSK was missing or not 64 bytes at Accept.	Confirm source complete the SW local key

For RADIUS-layer rejections that happen before or around the EAP exchange (bad shared secret, unknown access point), see **RADIUS Shared Secrets** and the **Sessions** status reference.

Call Flows

Overview

This page shows the end-to-end message flows for OmniTWAG. Each flow is a sequence diagram between the user equipment (UE), the WiFi access point (AP), the TWAG, and the core-network peers: the HSS (SWx), the PGW (S2a), and the OCS (Gy).

Read the per-feature pages for the detail behind each step: [Authentication](#), [S2a](#), [GTP-C](#), [User Plane](#), [IP Address Management](#), [IP Pools](#), [Online Charging](#), [RADIUS Credentials](#), and [Sessions](#).

Table of Contents

- [Actors and interfaces](#)
- [Successful attach - tunneled, charged](#)
- [Attach - NSWO with an IP pool](#)
- [Attach - NSWO with DHCP dish-out](#)
- [Attach - local breakout \(AP data plane\)](#)
- [Authentication failure - subscriber not provisioned for SWx](#)
- [Accounting and usage reporting](#)
- [Quota exhaustion and re-grant](#)
- [Teardown - subscriber disconnect](#)
- [Teardown - operator kick from the API](#)
- [Teardown - liveness timeout](#)
- [Charging-mode variants](#)

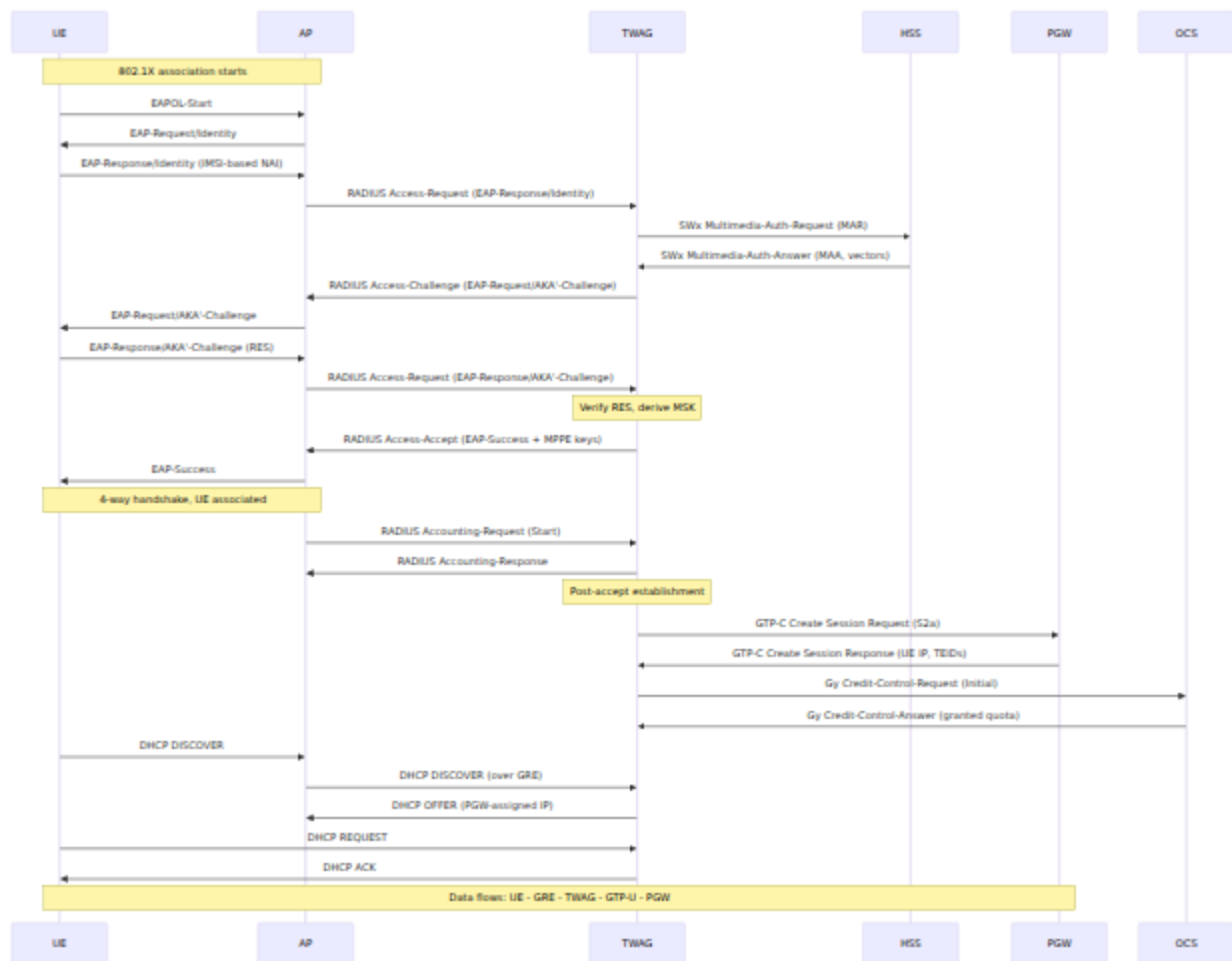
Actors and interfaces

Actor	Role	Interface to the TWAG
UE	Subscriber device with a SIM	802.11 / EAPOL (via the AP)
AP	WiFi access point and RADIUS client	RADIUS (UDP 1812 auth, 1813 accounting)
TWAG	Trusted WiFi access gateway and 3GPP AAA server	-
HSS	Home subscriber server	Diameter SWx (auth vectors)
PGW	Packet gateway	GTP-C/GTP-U over S2a (tunneled only)
OCS	Online charging system	Diameter Gy (charged only)

The breakout type and charging mode of a session come from the AP credential. See [RADIUS Credentials](#).

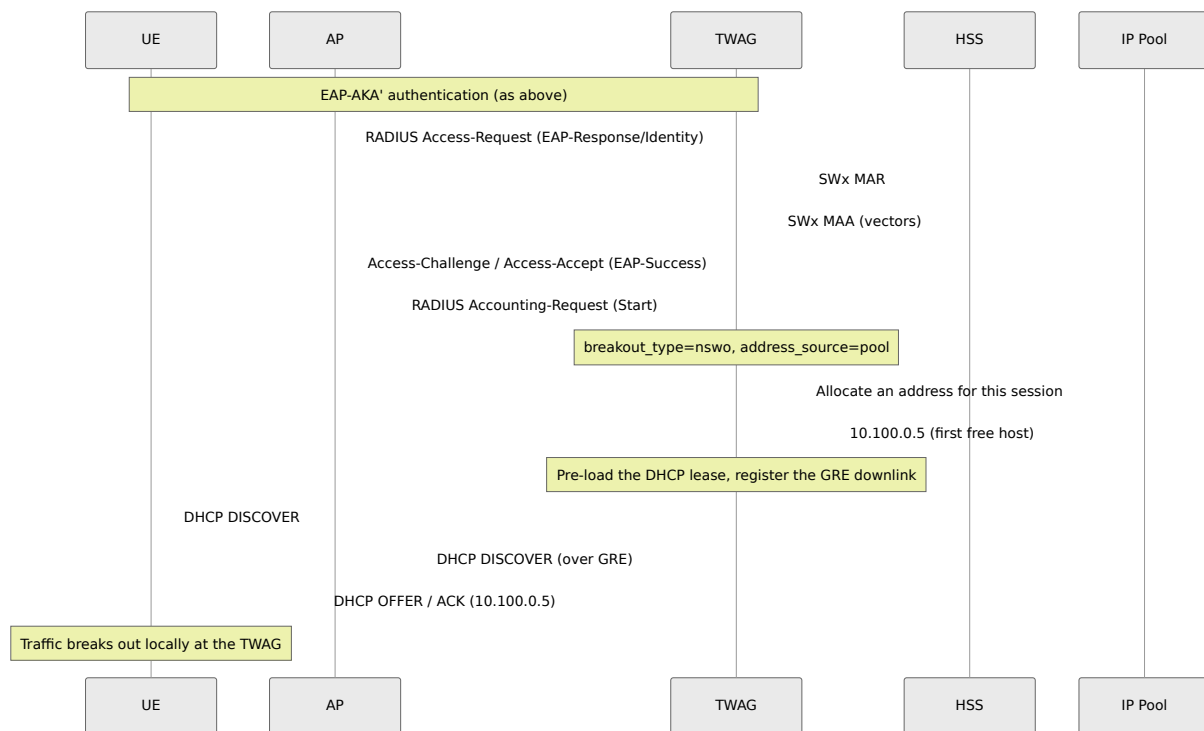
Successful attach - tunneled, charged

This is the full happy path for `breakout_type: tunneled` and `charging_mode: charged`. The TWAG authenticates the subscriber with EAP-AKA', establishes a GTP session to the PGW, delivers the PGW-assigned IP over DHCP, and opens a Gy charging session.



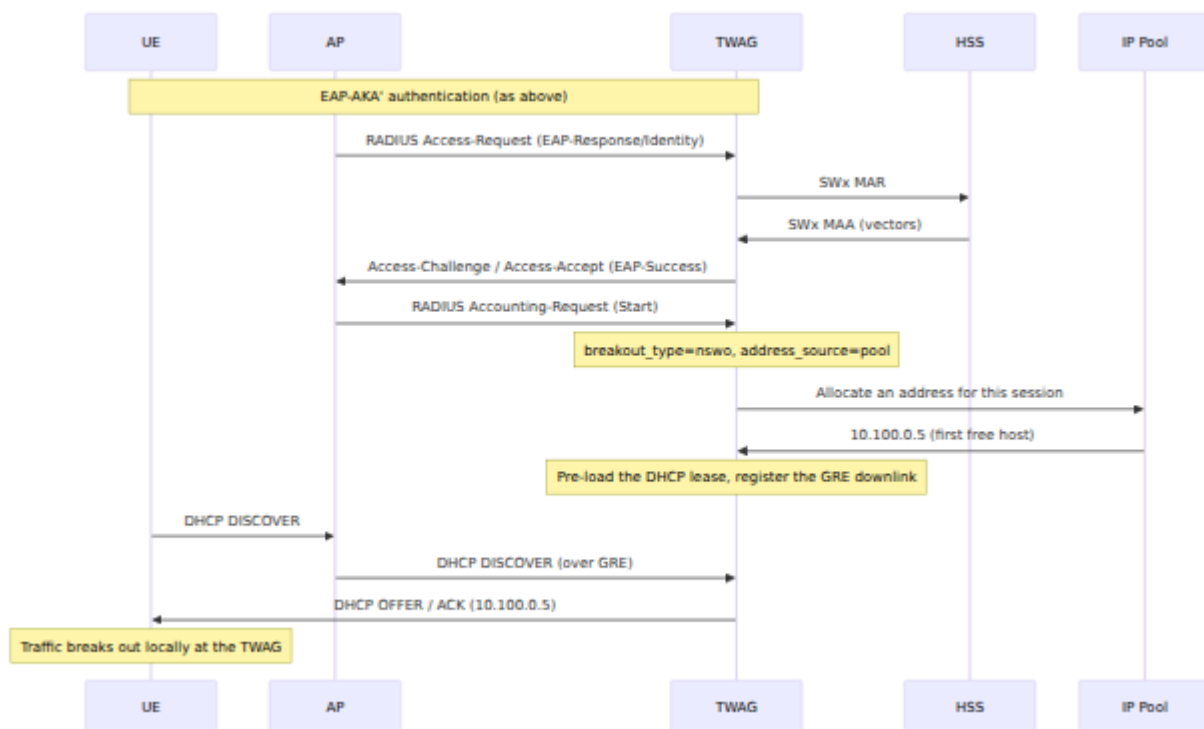
Attach - NSWO with an IP pool

For `breakout_type: nsw` and `address_source: pool`, the TWAG does not contact the PGW. The TWAG pre-allocates a fixed address from the associated IP pool and delivers it over DHCP. The subscriber traffic breaks out locally.



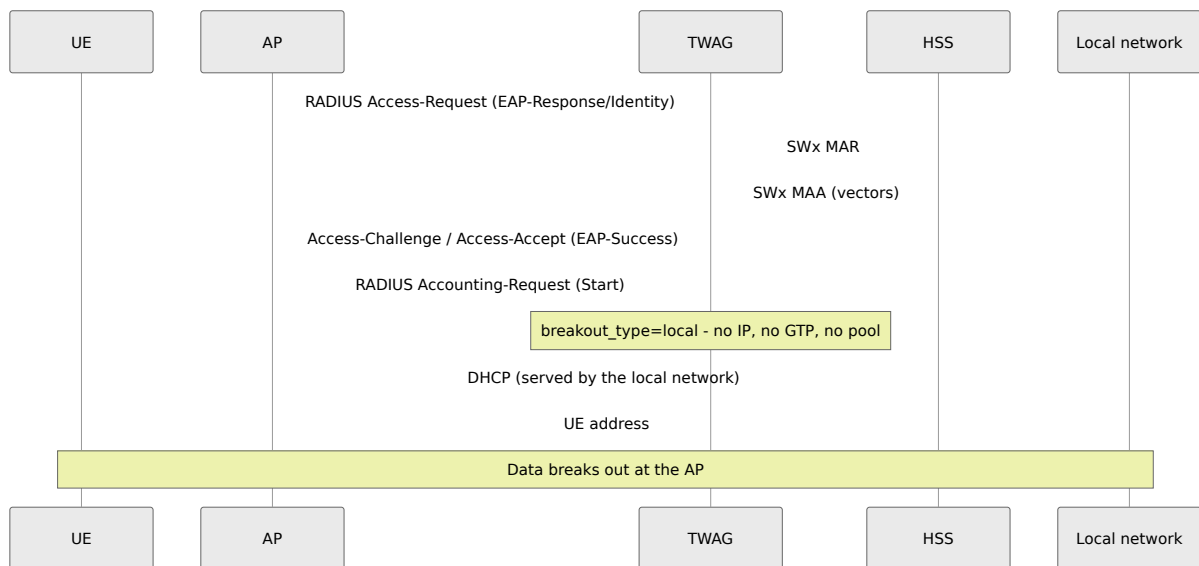
Attach - NSWO with DHCP dish-out

For `breakout_type: nswo` and `address_source: dhcp`, the TWAG binds the client MAC to the pool. The DHCP server dishes out the next free address from the pool range when the UE sends a DISCOVER. See [IP Pools](#).



Attach - local breakout (AP data plane)

For `breakout_type: local`, the AP owns the data plane. The TWAG authenticates the subscriber and assigns no IP. The UE gets its address from the local network. The TWAG establishes no GTP session and no DHCP lease.



Authentication failure - subscriber not provisioned for SWx

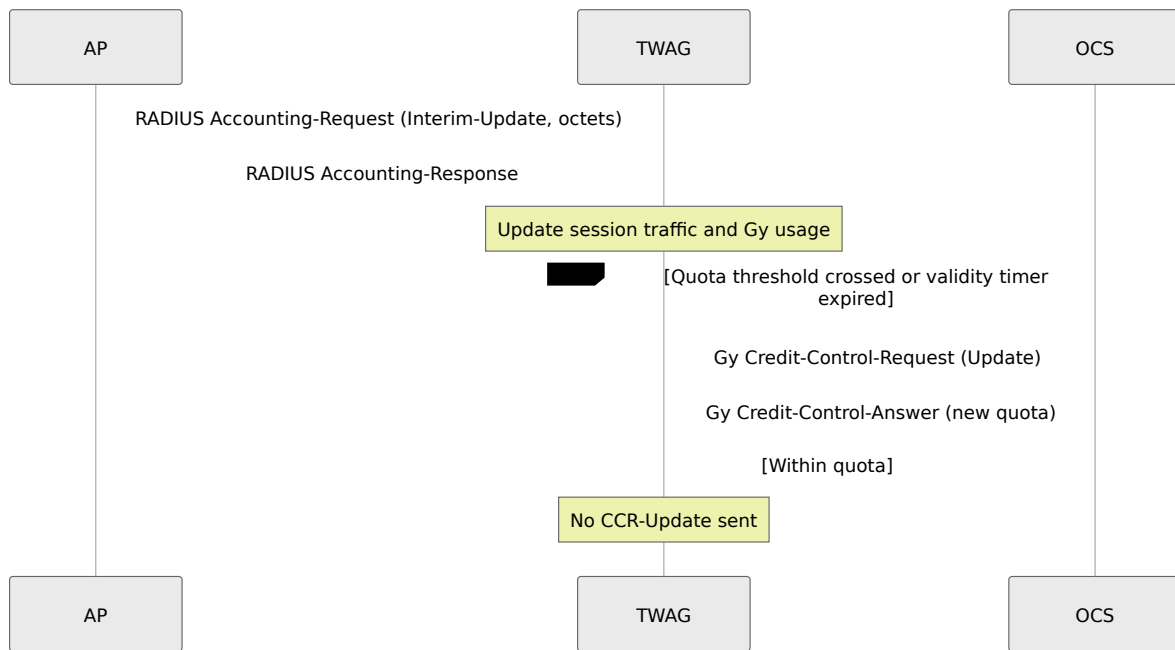
The HSS signals an SWx error in the grouped `Experimental-Result` AVP, not the base `Result-Code`. A subscriber known on S6a can still fail SWx if the HSS has no non-3GPP subscription for the IMSI. The TWAG rejects the session. See [Authentication](#).

```
Parse error on line 11: ... Experimental-Result TWAG->>AP: RADI -----
-----^ Expecting 'SOLID_OPEN_ARROW', 'DOTTED_OPEN_ARROW',
'SOLID_ARROW', 'BIDIRECTIONAL_SOLID_ARROW', 'DOTTED_ARROW',
'BIDIRECTIONAL_DOTTED_ARROW', 'SOLID_CROSS', 'DOTTED_CROSS',
'SOLID_POINT', 'DOTTED_POINT', got 'NEWLINE'
```

Try again

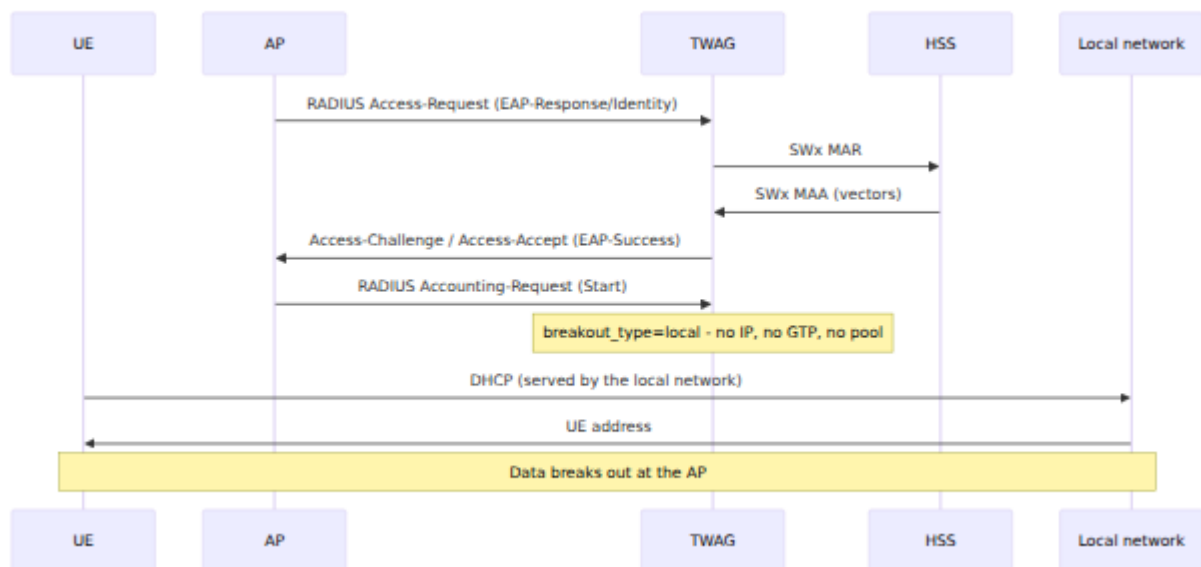
Accounting and usage reporting

The AP sends RADIUS accounting. The TWAG merges the traffic onto the session (see [Sessions](#)) and, in `charged` mode, reports usage to the OCS on a quota threshold or a validity-timer expiry, not on every interim update. See [Online Charging](#).



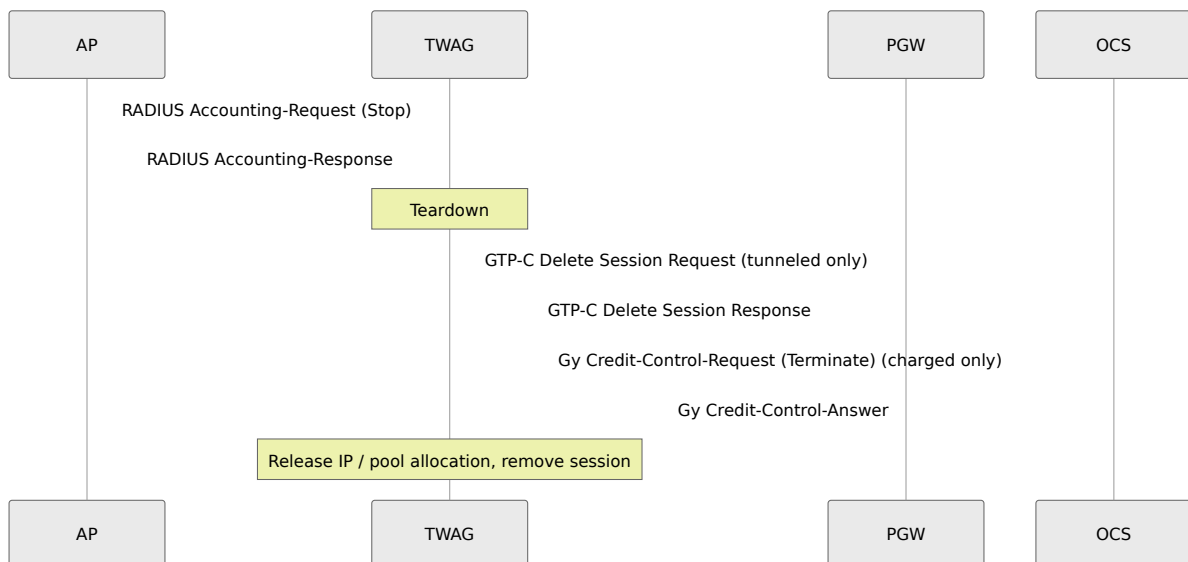
Quota exhaustion and re-grant

When the OCS grants no more quota, or sets a Final-Unit-Action, the TWAG blocks the UE data path. The TWAG keeps the session, the lease, and accounting so a top-up can unblock the UE. See [Online Charging](#).



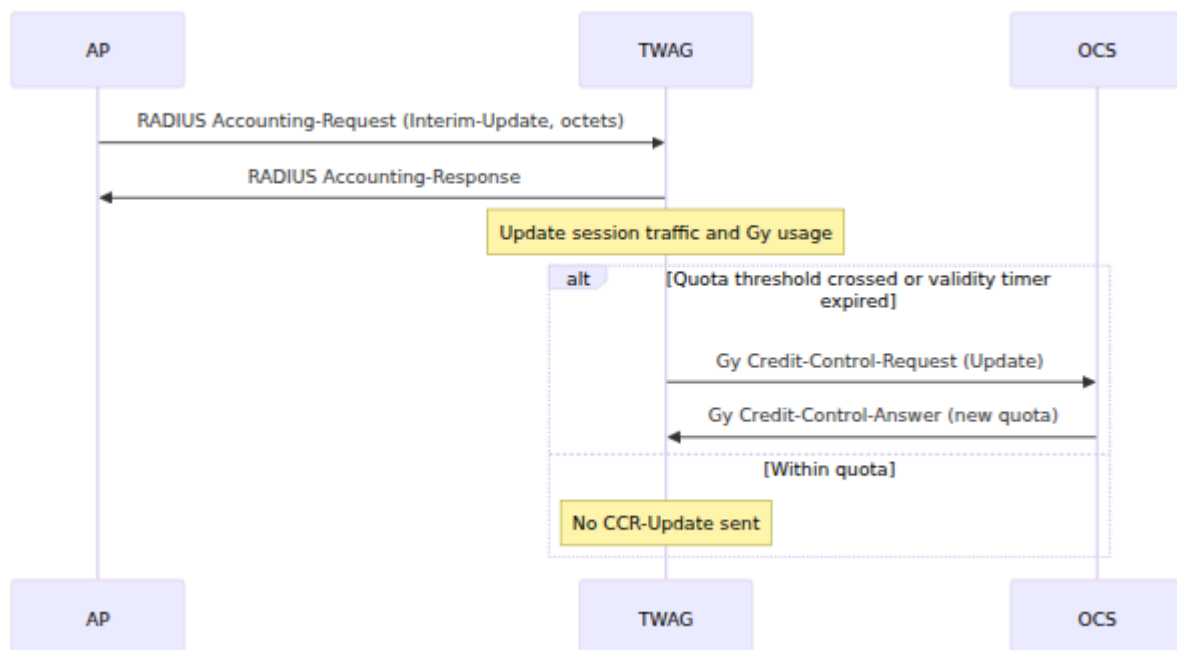
Teardown - subscriber disconnect

When the UE disconnects, the AP sends an Accounting-Stop. The TWAG tears the session down: it deletes the GTP session (tunneled), sends a Gy CCR-Terminate (charged), releases the address, and clears the session state.



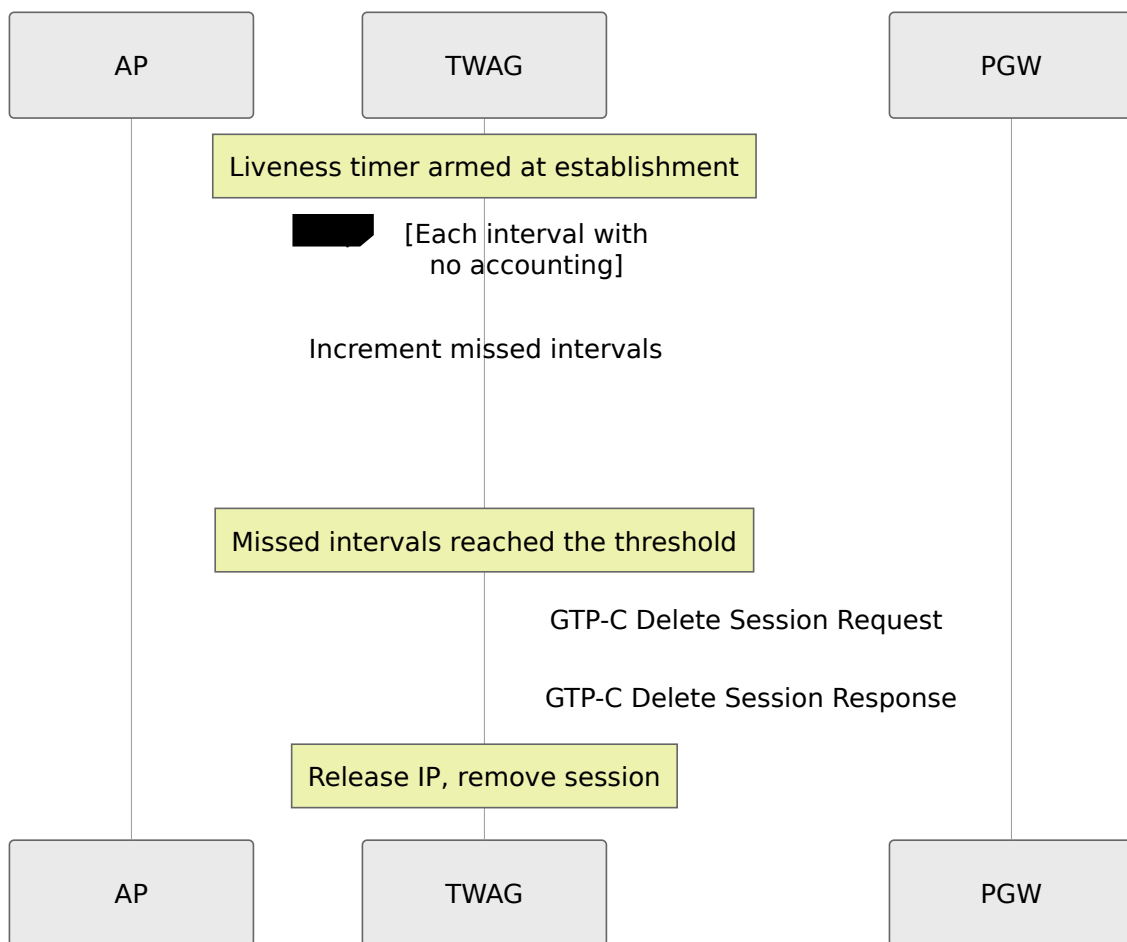
Teardown - operator kick from the API

An operator can remove a session through the API. The TWAG sends a RADIUS CoA-Disconnect (Disconnect-Message) to the AP, which drops the UE, then tears the session down. See [Sessions](#).



Teardown - liveness timeout

The TWAG monitors an established session with a liveness timer. Accounting interim updates reset the timer. If the AP sends no updates for the configured number of intervals, the TWAG declares the session dead and tears it down. A session that never completes authentication is reaped by a separate identity-timeout. See [Sessions](#).

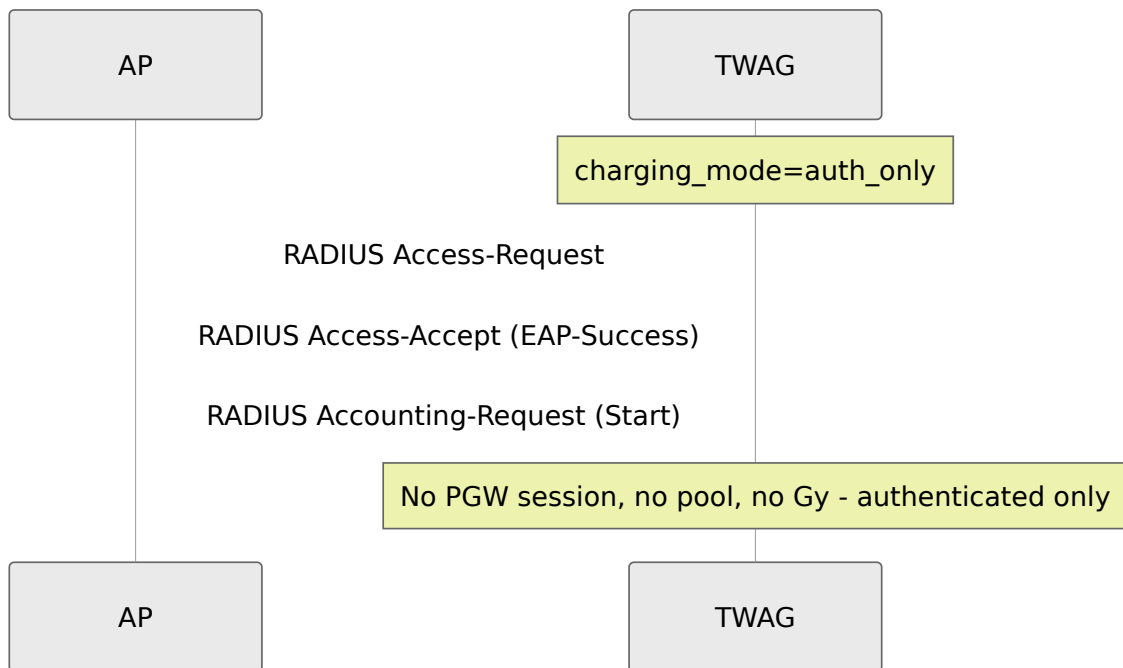


Charging-mode variants

The `charging_mode` on the AP credential changes the post-accept steps. See [RADIUS Credentials](#).

charging_mode	Data session	Gy
charged	Established (per <code>breakout_type</code>)	CCR-Initial / Update / Terminate
uncharged	Established (per <code>breakout_type</code>)	No Gy session
auth_only	Not established	No Gy session

For **uncharged**, the flow is the tunneled or NSWO attach above, but the TWAG sends no Gy messages. For **auth_only**, the flow stops after the Access-Accept and Accounting-Start: the TWAG establishes no data session.



Overview

Connection Modes, WLCP, and Mobility

Overview

This guide explains the TWAG connection modes, the WLAN Control Protocol (WLCP), and handover behaviour. It describes what the OmniTWAG software does today. It also marks each area that is partial or not yet integrated, so operators do not plan around behaviour that the code does not run.

The TWAG follows the Trusted WLAN model in [3GPP TS 23.402](#) Section 16. For the data plane, authentication, and S2a signalling that these modes build on, see:

- [Architecture](#): data plane, session lifecycle, and interfaces
- [Authentication](#): EAP-AKA / EAP-AKA' and key material
- [S2a GTP-C](#): Create Session, Delete Session, and bearer signalling
- [Sessions](#): the authenticated session and its status lifecycle
- [User Plane](#): GRE, GTP-U, and DHCP delivery
- [RADIUS Shared Secrets](#): per-AP breakout and charging applied after authentication

Table of Contents

- [Overview and Mode Comparison](#)
- [Mode Negotiation](#)
- [Transparent Single-Connection Mode \(TSCM\)](#)
- [Single-Connection Mode \(SCM\)](#)
- [Multi-Connection Mode \(MCM\) and WLCP](#)
- [Handover and Mobility](#)
- [Configuration](#)
- [Observability](#)
- [Implementation Status](#)

Overview and Mode Comparison

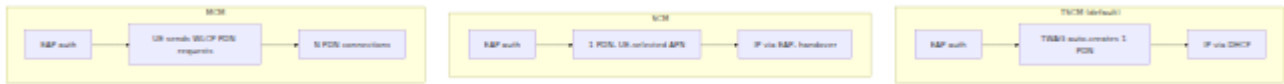
The TWAG supports three connection modes for a UE that accesses the EPC through a Trusted WLAN. The mode controls how the UE learns its IP address, whether the UE can signal an APN, and whether the UE can hold more than one PDN connection.

Property	TSCM	SCM	MCM
Mode code	0	1	2
Full name	Transparent Single-Connection Mode	Single-Connection Mode	Multi-Connection Mode
PDN connections per UE	One	One	Many
IP address delivery	DHCP	EAP	EAP / WLCP
UE-signalled APN selection	No	Yes	Yes (per connection)
Handover with IP preservation	No	Yes	Yes
Dedicated bearers	No	Yes	Yes
UE control protocol	None	None	WLCP (TS 24.244)
UE aware of S2a	No	Yes	Yes

TSCM is the default mode. TSCM matches pre-Release-12 behaviour, where the UE does not signal the network. The TWAG selects the APN, creates one PDN

connection, and delivers the PGW-allocated IP address through DHCP.

The mode codes 0, 1, and 2 follow the TWAN-Connection-Mode AVP in 3GPP TS 29.273.

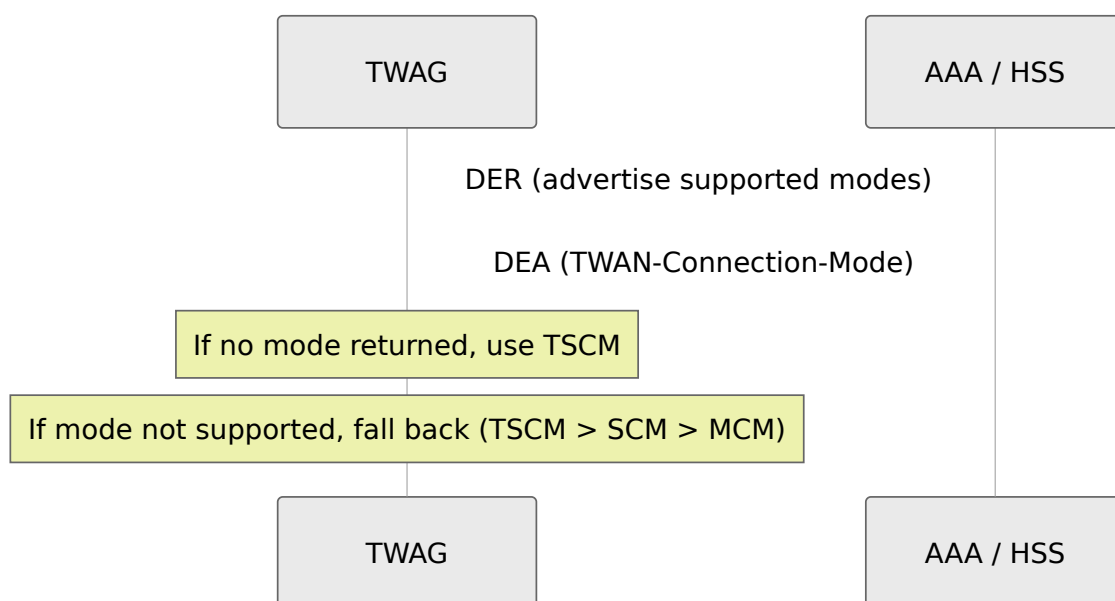


Mode Negotiation

The connection mode is negotiated during authentication, on the STa exchange that carries EAP. See [Authentication](#) for the EAP flow itself. The design is:

1. The TWAG advertises the modes it supports to the AAA server. The advertised set comes from the `connection_modes` configuration key.
2. The AAA server returns the selected mode in the STa DEA response, in the TWAN-Connection-Mode AVP.
3. If the AAA server does not indicate a mode, the TWAG uses TSCM as the default.

If the AAA server selects a mode that the TWAG does not support, the TWAG falls back. The fallback order prefers TSCM, then SCM, then MCM. The TWAG selects the first supported mode from this order.



The TWAG reads the selected mode from any of these keys in the AAA response, in order: `twan_connection_mode`, `TWAN-Connection-Mode`, `connection_mode`, or `Connection-Mode`. The value can be a numeric code (0/1/2) or an atom (`:tscm`/`:scm`/`:mcm`). An unrecognised value defaults to TSCM.

On the STa path, the STa DER advertises the TWAG preferred supported mode in the `TWAN-Connection-Mode` AVP. The STa DEA parse reads the negotiated mode back and defaults to TSCM when the DEA carries none.

Status: the negotiated mode is not enforced on the live session path. The STa client (`StaClient`) does advertise the supported mode in the DER and reads the negotiated mode from the DEA. The connection-mode module is present and unit-tested. However, the STa pass-through auth path is not the live RADIUS authentication path, and the live client never applies the negotiated mode: the `store_connection_mode` entry point has no caller on the attach path, and the attach and data path always behave as TSCM (IP through DHCP, no WLCP, no UE-signalled handover or APN). Today every live session follows the TSCM path. See [Implementation Status](#).

Transparent Single-Connection Mode (TSCM)

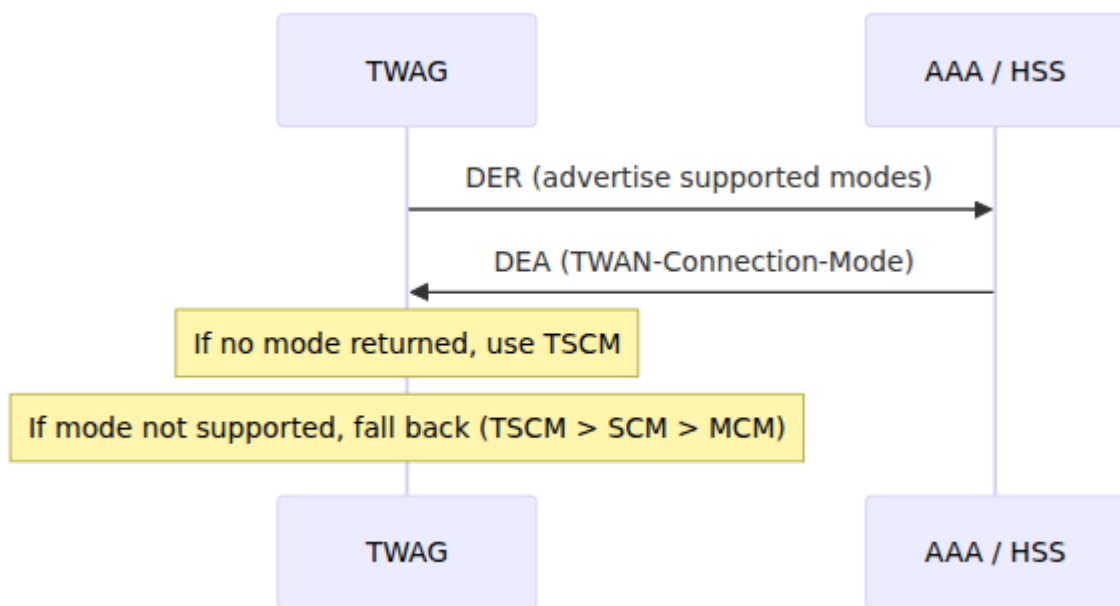
TSCM is the default mode and the behaviour that the live session path runs today. The UE does not signal the network. TSCM is described in the [Architecture](#) session lifecycle, and is summarised here.

In TSCM:

- The UE holds one PDN connection.
- The TWAG selects the APN. The UE cannot request an APN.
- The UE receives its IP address through DHCP, served by the TWAG over the GRE tunnel. See [User Plane](#).
- There is no WLCP signalling.

- Handover with IP preservation, dedicated bearers, and UE-signalled APN selection are not available.

The TWAG creates the S2a session automatically after authentication. It sends a Create Session Request to the PGW with the Handover Indication flag clear. It then pre-allocates the PGW-assigned IP address in its DHCP server for the UE MAC address. See [S2a GTP-C](#) for the Create Session detail and [Sessions](#) for the resulting session and its status lifecycle. Charging for the session, when enabled, follows through Gy. See [Online Charging \(Gy\)](#).



Single-Connection Mode (SCM)

SCM keeps a single PDN connection per UE, but the UE is aware of S2a. SCM adds these capabilities over TSCM:

- The UE receives its IP address through EAP, not DHCP.
- The UE can select a non-default APN.
- The UE keeps its IP address across 3GPP-to-WLAN and WLAN-to-3GPP handover.
- The PGW can create dedicated bearers.

The connection-mode module reports these SCM capabilities. It classifies SCM as a mode that delivers IP through EAP, supports handover, supports APN

selection, and supports dedicated bearers.

Status: SCM is not yet a distinct runtime path. The capability model for SCM exists, but the session-establishment code does not yet branch on the negotiated mode. A UE that the AAA server places in SCM is still served through the TSCM path (auto-created PDN, IP through DHCP). Do not rely on EAP-delivered IP or UE-signalled APN selection for SCM in this release.

Multi-Connection Mode (MCM) and WLCP

MCM lets a UE hold several PDN connections at the same time. The UE manages each connection with the WLAN Control Protocol (WLCP), defined in [3GPP TS 24.244](#). After authentication, the TWAG does not auto-create a PDN connection. Instead, the UE sends WLCP PDN Connection Requests. The TWAG maps each PDN connection to a separate S2a GTP-C session toward the PGW.

WLCP Transport

The TWAG runs a WLCP server on UDP port `2156`, the standard WLCP port. The server routes each packet to a per-UE WLCP session handler, keyed on the UE identity. The server maps the UE source IP and port to the identity after authentication.

In production, WLCP signalling runs over DTLS per [TS 23.402](#) Section 16.1.4A.3.1. The current server handles the plain UDP layer only. DTLS is not yet applied.

WLCP Message Format

Each WLCP message has a fixed header, followed by a list of Information Elements (IEs).

```
Message Type (1 byte) | Length (2 bytes) | Transaction ID (1 byte)
| IEs...
```

Each IE has this format:

IE Type (1 byte) | IE Length (2 bytes) | IE Value (variable)

The **Length** field counts the full message, including the 4-byte header.

WLCP Message Types

Code	Message	Direction	Purpose
0x01	PDN Connection Request	UE to TWAG	Request a new PDN connection
0x02	PDN Connection Accept	TWAG to UE	Connection created, with IP / DNS / QoS
0x03	PDN Connection Reject	TWAG to UE	Connection refused, with a cause
0x04	PDN Disconnect Request	UE to TWAG	Release a PDN connection
0x05	PDN Disconnect Accept	TWAG to UE	Connection released
0x06	Downlink Data Notification	TWAG to UE	Notify the UE of pending downlink data

WLCP Information Elements

Code	IE	Notes
0x01	APN	Access Point Name string
0x02	PDN Type	1 IPv4, 2 IPv6, 3 IPv4v6
0x03	PCO	Protocol Configuration Options
0x04	PDN Address	Allocated UE IP address (IPv4, IPv6, or IPv4v6)
0x05	Bearer QoS	QCI and MBR / GBR values
0x06	TFT	Traffic Flow Template
0x07	Cause	Result cause value
0x08	Bearer ID	EPS Bearer Identity (EBI)
0x09	DNS Address	One or more DNS server addresses

WLCP Cause Values

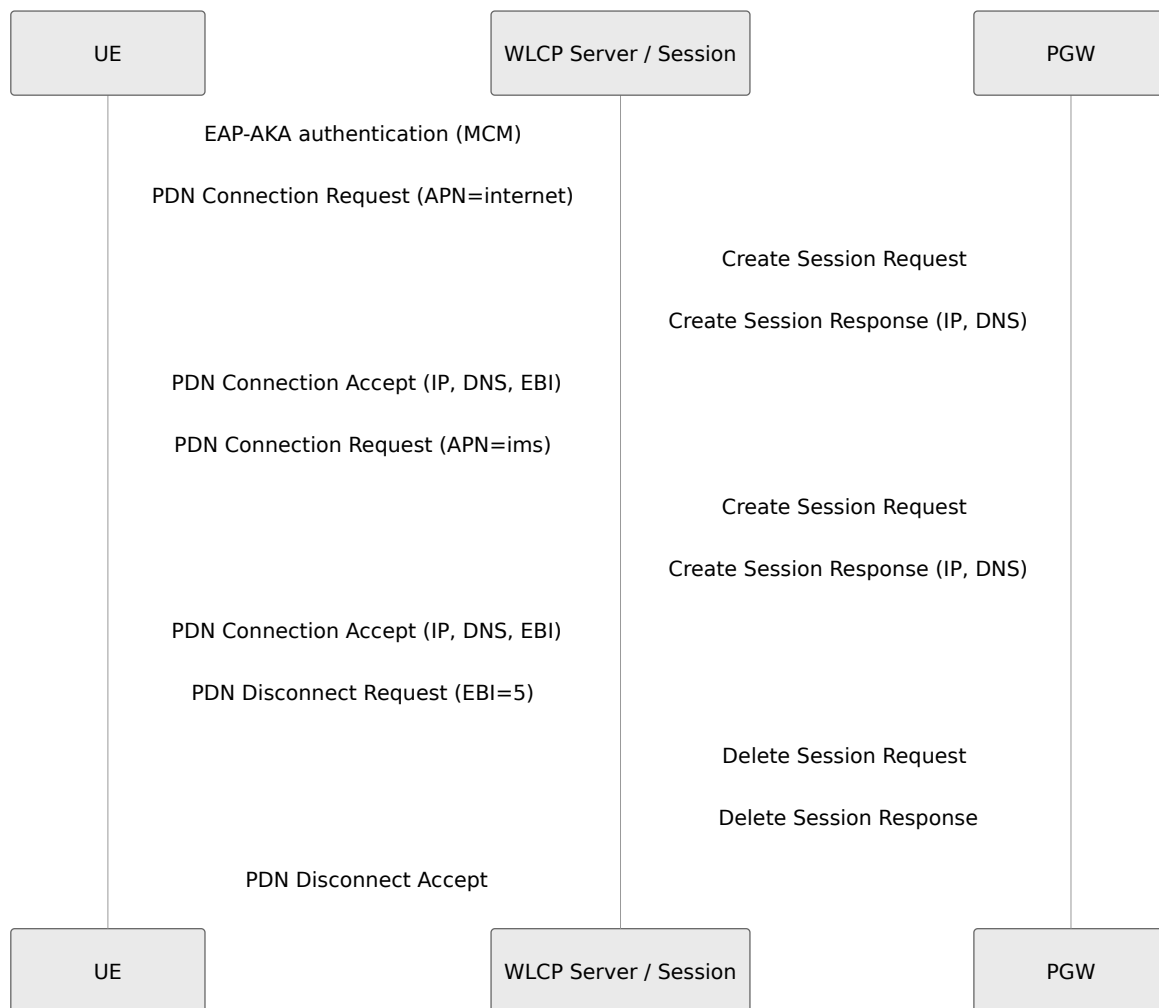
Code	Cause	Meaning
0x10	request_accepted	Request accepted
0x20	apn_not_allowed	APN not allowed for this UE
0x21	no_resources	No resources available
0x22	unknown_apn	APN not known
0x23	user_auth_failed	User authentication failed
0x24	network_failure	Network failure (for example S2a failure)
0x25	pdn_type_not_supported	Requested PDN type not supported
0x30	request_rejected	Request rejected

PDN Connection Setup

When a PDN Connection Request arrives, the WLCP session handler does these steps:

1. It reads the APN, PDN type, and PCO from the message. If the APN is absent, it uses `internet`. If the PDN type is absent, it uses IPv4.
2. It checks that the APN is allowed. See `allowed_apns` in [Configuration](#).
3. If the UE has no known IMSI, it rejects the request with cause `user_auth_failed`.
4. It creates an S2a GTP-C session toward the PGW for that APN, with RAT type `3` (WLAN) and the Handover Indication flag clear.
5. On success, it registers the GTP-U tunnel and stores the PDN connection, keyed on the EBI. It then sends a PDN Connection Accept with the UE IP address, and, when present, the DNS servers, bearer QoS, and EBI.

6. On an S2a failure, it sends a PDN Connection Reject with cause `network_failure`.



PDN Connection Teardown

When a PDN Disconnect Request arrives, the handler finds the PDN connection by EBI. If the EBI is unknown, it replies with a PDN Connection Reject and cause `request_rejected`. If the EBI is known, it deletes the S2a GTP-C session, unregisters the GTP-U tunnel, removes the connection, and replies with a PDN Disconnect Accept. When the UE has no PDN connections left, the WLCP session handler stops.

If the PGW deletes a bearer, the handler removes the matching PDN connection and frees the local GTP-U tunnel. It does not send a Delete Session in this case, because the PGW already released the bearer.

WLCP Key Derivation

The WLCP key `K_WLCP` protects WLCP signalling. It is derived from the Master Session Key (MSK) that EAP authentication produces. The derivation follows [3GPP TS 33.402](#) Section 6.2 and Annex A.

```
K_WLCP = HMAC-SHA-256(MSK, FC | P0 | L0)
```

```
FC = 0x21
```

```
P0 = TWAG IP address (4 bytes for IPv4, 16 bytes for IPv6)
```

```
L0 = length of P0 (2 bytes, big-endian)
```

The output is 32 bytes (256 bits). The TWAG binds the key to its own IP address through the `P0` parameter. A simplified variant derives the key from the MSK and a fixed label when the TWAG IP is not yet known.

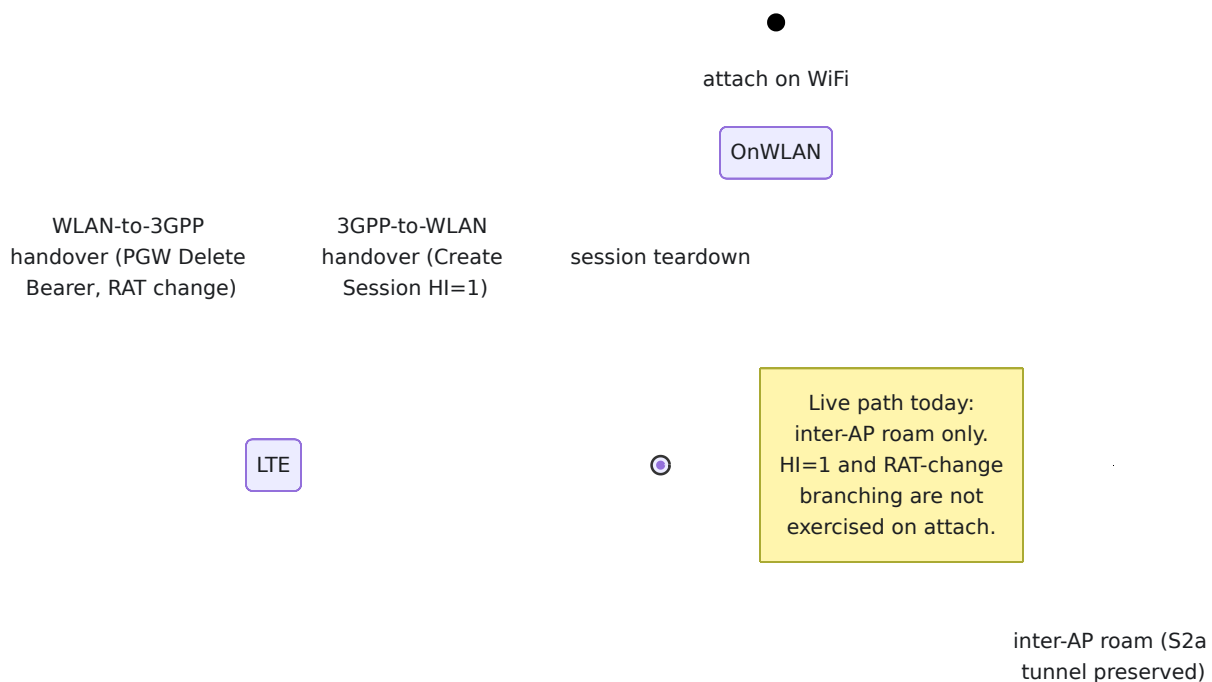
Status: MCM and WLCP are partial and not integrated. The WLCP codec, the WLCP UDP server, the per-UE session handler, and the key derivation all exist and are unit-tested. They are not started by the application and are not driven by the authentication flow. The following are not implemented:

- DTLS protection for WLCP. Only plain UDP is handled.
- Dedicated bearer management inside MCM. The codec carries QoS and TFT IEs, but the handler does not act on PGW-initiated dedicated-bearer creation.
- Handover inside MCM.
- The Downlink Data Notification message is defined in the codec but is not generated by the session handler.

Treat MCM as a preview. Do not enable it for production UEs.

Handover and Mobility

The TWAG models three mobility scenarios per [TS 23.402](#). Only inter-AP mobility is active on the live session path today.



3GPP-to-WLAN Handover

When a UE moves from LTE to WiFi, the TWAG must keep the existing IP session. To do this, the TWAG sends a Create Session Request with the Handover Indication (HI) flag set. The PGW then preserves the existing IP address and bearer context, instead of allocating a new address. See [S2a GTP-C](#) for the Create Session and Indication IE detail.

The S2a Create Session Request supports the Handover Indication flag. The handover handler can route a handover-in event to the UE client process with the handover flag set.

Status: the main session path does not set the handover flag. The automatic session establishment always sends Create Session with the Handover Indication flag clear. As a result, IP preservation on 3GPP-to-WLAN handover is not exercised by the normal attach path in this release.

WLAN-to-3GPP Handover

When a UE moves from WiFi to LTE, the PGW sends a Delete Bearer Request. The cause value indicates a RAT change. The TWAG treats these cause values as a RAT-change handover:

Cause code	Name
4	RAT changed (3GPP to non-3GPP)
5	ISR deactivation
10	Access changed from non-3GPP to 3GPP

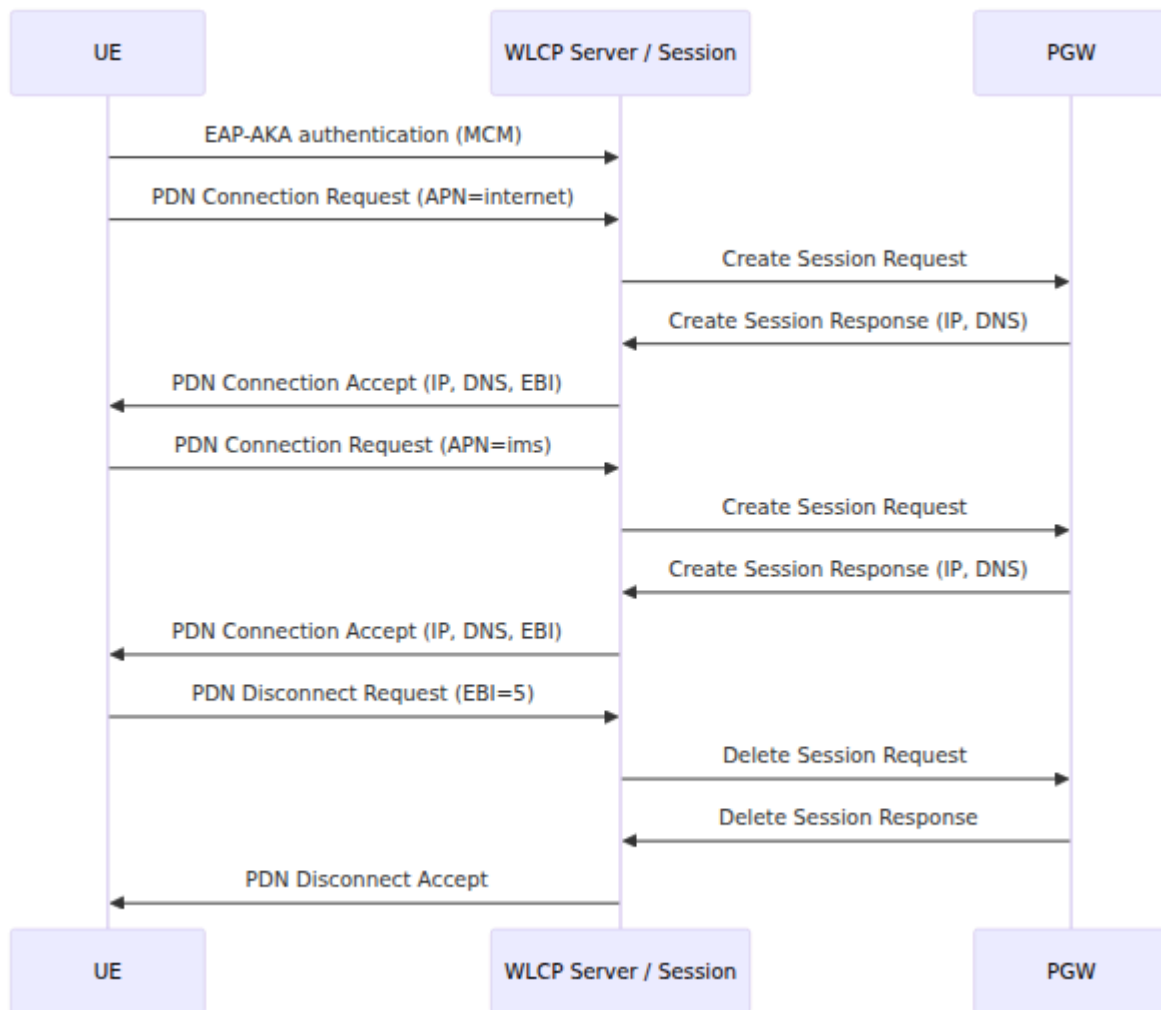
On a RAT-change Delete Bearer, the TWAG cleans up the WiFi session locally. It does not send a GTP-C Delete Session, because the PGW already released the bearer. The UE keeps its IP address on the LTE side.

Status: the delete-bearer handler does not branch on cause today.

The UE client process handles a PGW Delete Bearer by running local cleanup and then stopping, for any cause. The RAT-change classification exists as a helper, but the client does not use the cause to choose a different teardown path.

Inter-AP Mobility

When a UE roams between WiFi APs inside the same TWAN, the S2a GTP tunnel stays up. The UE keeps the same identity, but arrives with a different NAS-IP-Address and Called-Station-Id. The TWAG updates the AP tracking data on the UE client process and does not tear down the session. This keeps the UE IP address stable across the AP change.



Inter-AP mobility is the one mobility path that the live session code runs. It updates the UE client process fields for NAS-IP and Called-Station-Id, and records the new AP in AP tracking state. See [Per-AP Accounting](#) for the AP tracking data.

Configuration

Connection Modes

Set the modes that the TWAG advertises with the `connection_modes` key. The value is a list of mode atoms. The default is `[:tscm]`.

```

config :omnitwag,
  # Connection modes this TWAG advertises to the AAA server.
  connection_modes: [:tscm, :scm]

```

Parameter	Type	Required	Default	Description
<code>connection_modes</code>	List	No	<code>[:tscm]</code>	Modes the TWAG advertises. Use <code>:tscm</code> , <code>:scm</code> , and <code>:mcm</code> . The order does not matter. The advertised codes are sorted before use.

Note. The advertised set is read by the connection-mode module and the STa client sends the preferred mode in the STa DER. The live RADIUS session path does not act on the negotiated mode yet. See [Implementation Status](#). Keep the default `[ :tscm ]` for production until SCM and MCM are integrated.

WLCP Server (MCM)

The WLCP server settings live under the `:wlcp` key. The WLCP server is only relevant to MCM.

```
config :omnitwag,
  wlcp: %{
    enabled: true,
    port: 2156,
    allowed_apns: :all
  }
```

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>false</code>	Enable the WLCP server. Leave <code>false</code> unless you test MCM.
<code>port</code>	Integer	No	<code>2156</code>	UDP port for WLCP. <code>2156</code> is the standard WLCP port.
<code>allowed_apns</code>	Atom or List	No	<code>:all</code>	APN allow-list for WLCP PDN requests. Use <code>:all</code> to allow every APN, or a list of APN strings, for example <code>["internet", "ims"]</code> . A request for an APN not in the list is rejected with cause <code>apn_not_allowed</code> .

Note. The WLCP server is not started by the application supervisor in this release. The `enabled` key is defined for future use.

Related Configuration

The MCM PDN setup path reuses the GTP settings for its S2a sessions. See the [Architecture](#) configuration summary for the `gtp` block, including `pgw_ip`, `local_ip`, `mcc`, and `mnc`.

Observability

The WLCP server and session handlers emit telemetry events. These events feed the metrics pipeline. See [Metrics](#) for the metrics catalogue.

Telemetry event	Emitted when
<code>[:wlc, :packet, :received]</code>	A WLC packet arrives. Carries <code>bytes</code> .
<code>[:wlc, :packet, :malformed]</code>	A packet fails to decode.
<code>[:wlc, :packet, :unregistered]</code>	A packet arrives from an unmapped source address.
<code>[:wlc, :pdn, :request]</code>	A PDN Connection Request is received.
<code>[:wlc, :pdn, :accept]</code>	A PDN connection is established.
<code>[:wlc, :pdn, :reject]</code>	A PDN connection is rejected. Carries the cause.
<code>[:wlc, :pdn, :disconnect_request]</code>	A PDN Disconnect Request is received.
<code>[:wlc, :pdn, :disconnect_accept]</code>	A PDN connection is released.
<code>[:twag, :handover, :in]</code>	A 3GPP-to-WLAN handover-in event is routed. Carries <code>source</code> .
<code>[:twag, :handover, :out]</code>	A WLAN-to-3GPP RAT change is handled. Carries <code>cause</code> .
<code>[:twag, :handover, :ap_change]</code>	An inter-AP mobility event is handled.

The WLC server and session handlers also log at info level. They log each PDN connection request, accept, reject, and disconnect, with the UE identity, APN, and EBI. The handover handler logs each handover-in, RAT change, and AP change, with the UE identity.

Implementation Status

This table summarises what runs on the live session path, what exists but is not integrated, and what is not implemented. Use it to plan around real behaviour.

Feature	Status
TSCM (default, auto-created PDN, IP via DHCP)	Live
Inter-AP mobility (S2a tunnel preserved)	Live
S2a Create Session with Handover Indication flag	Supported in the S2a encoder, but the attach path always clears it
Connection-mode negotiation logic and fallback	Present as a module, unit-tested
STa DER advertises supported modes	Wired (STa DER carries the preferred TWAN-Connection-Mode, plus SSID / BSSID)
STa DEA reads selected mode	Wired (StaClient parses the negotiated mode, defaults to TSCM)
Negotiated mode applied to the live session	Not wired (store_connection_mode has no caller on the attach path)
SCM as a distinct runtime path (EAP IP, APN selection)	Not integrated (served as TSCM)
WLCP codec (all message types and IEs)	Present, unit-tested
WLCP UDP server and per-UE session handler	Present, not started by the application
WLCP key derivation (K_WLCP, FC=0x21)	Present, unit-tested

Feature	Status
MCM multiple concurrent PDN connections	Present in the handler, not integrated end to end
DTLS protection for WLCP	Not implemented (plain UDP only)
Dedicated bearer management in MCM	Not implemented
WLCP Downlink Data Notification generation	Not implemented (codec only)
Handover in MCM	Not implemented
3GPP-to-WLAN IP preservation on the attach path	Not exercised (attach clears the HI flag)
WLAN-to-3GPP teardown by RAT-change cause	Not branched (delete-bearer cleanup is cause-independent)

IP Address Management (DHCP / DHCPv6 / IPv6 RA / PCO)

Overview

OmniTWAG gives each UE an IP address and the network parameters that go with it. This subsystem covers DHCPv4, DHCPv6, IPv6 Router Advertisement (SLAAC), and the delivery of DNS and P-CSCF addresses. The subsystem does not invent an address at random. It delivers an address that another part of the system already chose. The source depends on the per-AP mode.

- In **tunneled (home-routed) mode** the PGW allocates the address over S2a GTP, and OmniTWAG delivers that same address to the UE. This design follows [3GPP TS 23.402 Section 16.2](#).
- In **NSWO (local breakout) mode** there is no PGW. OmniTWAG allocates the address itself from a local **IP pool** that the operator defines. See [IP Pools](#).

The mode, the address source, and the pool all come from the AP credential that matches the RADIUS source. See [RADIUS Credentials](#) for how an AP maps to a `breakout_type`, an `address_source`, and an `ip_pool_id`.

For the operating modes and the GRE data path, see the [User Plane](#) reference. For the GTP-C control plane that carries the PAA and PCO, see [S2a GTP-C](#). For the per-UE state machine that drives allocation, see [Sessions](#).

Table of Contents

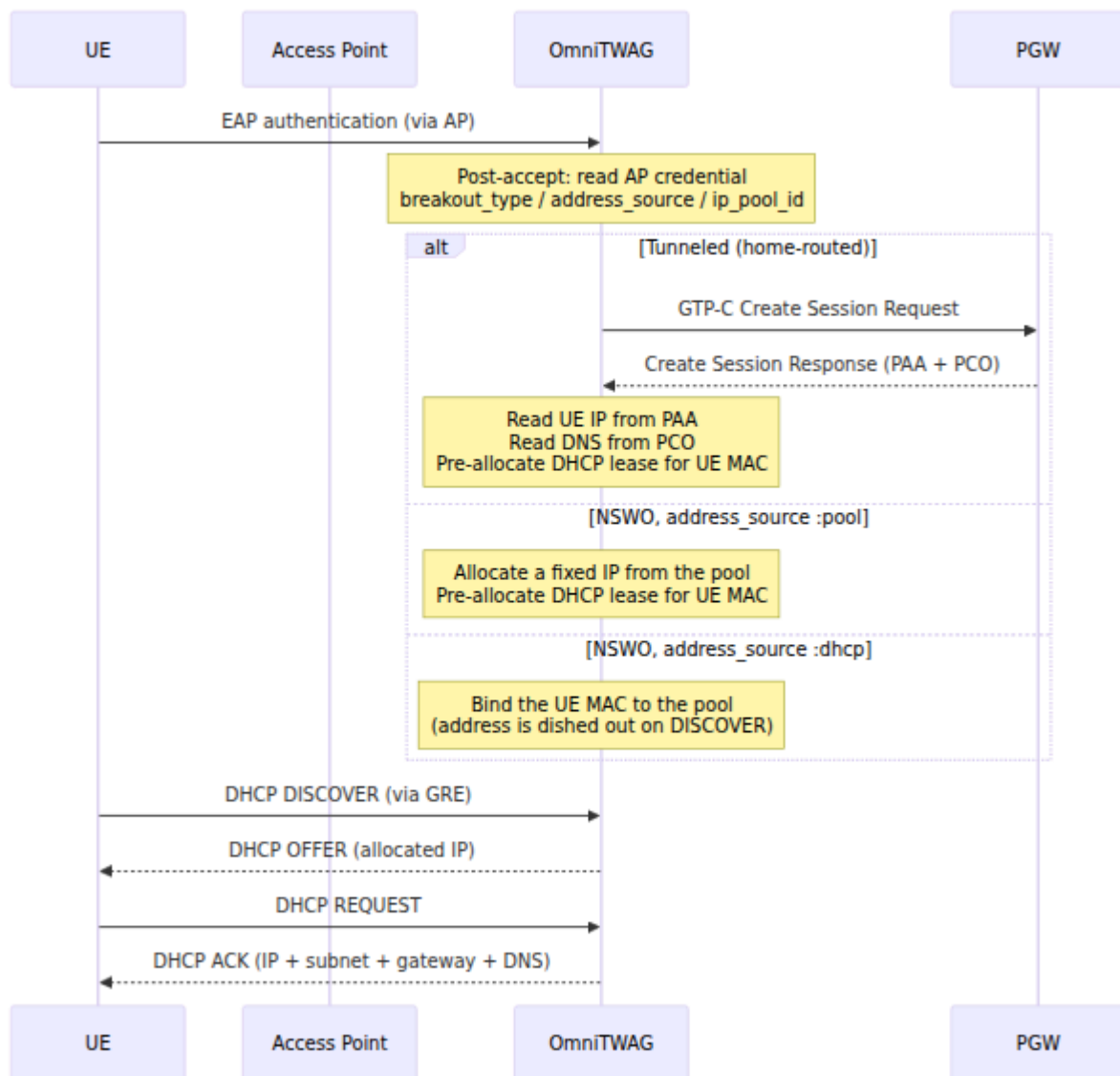
- [Overview: who allocates the address](#)
- [Address sources](#)

- Tunneled sessions: PGW-assigned address
- NSWO sessions: pool and dhcp
- DHCPv4
- DHCPv6
- IPv6 Router Advertisement (SLAAC)
- DNS and PCO delivery
- Address pools and exhaustion
- Configuration
- Observability
- Troubleshooting

Overview: who allocates the address

OmniTWAG runs a lightweight DHCPv4 server, a stateless and stateful DHCPv6 server, and an IPv6 Router Advertisement server. These servers do not scan a range for a free address in the tunneled case. They deliver the address that the PGW already allocated. In NSWO mode the same DHCPv4 server serves an address that OmniTWAG took from a local IP pool.

The address flow starts when the UE authenticates. After the Access-Accept, the per-UE session process runs a post-accept establishment step. That step reads the AP credential to decide the mode and then allocates the address by the right path.



Address sources

The AP credential sets the address source through two fields:

- `breakout_type`: `:tunneled`, `:nsw`, or `:local`.
- `address_source`: `:pool` or `:dhcp`. This applies only when `breakout_type` is `:nsw`.

Mode	Credential	Who chooses the UE address	How it reaches the UE
Tunneled (home-routed)	<code>breakout_type:</code> <code>:tunneled</code>	The PGW, in the GTP-C Create Session Response PAA.	Pre-allocated DHCPv4 lease, DHCPv6, or IPv6 RA.
NSWO, fixed	<code>breakout_type:</code> <code>:nsw</code> , <code>address_source:</code> <code>:pool</code>	OmniTWAG, from the <code>ip_pool_id</code> pool, at establishment.	Pre-allocated DHCPv4 lease.
NSWO, dish-out	<code>breakout_type:</code> <code>:nsw</code> , <code>address_source:</code> <code>:dhcp</code>	OmniTWAG, from the <code>ip_pool_id</code> pool, on the DHCP DISCOVER.	DHCPv4 pool dish-out on DISCOVER.
AP-local	<code>breakout_type:</code> <code>:local</code>	The local network behind the AP.	OmniTWAG assigns no address.

For the credential fields and how a RADIUS source maps to a credential, see [RADIUS Credentials](#). For the pool store, see [IP Pools](#).

Tunneled sessions: PGW-assigned address

For a tunneled session, OmniTWAG establishes an S2a session to the PGW with a GTP-C Create Session Request. The PGW answers with a Create Session Response. The response carries two things that this subsystem uses:

- The **PDN Address Allocation (PAA)**. This is the UE IP address.
- The **Protocol Configuration Options (PCO)**. This carries DNS servers, P-CSCF addresses, and the link MTU.

OmniTWAG reads the PAA and the PCO from the response. OmniTWAG then pre-allocates a DHCP lease for the UE MAC address. When the UE sends a DHCP request, OmniTWAG replies with this pre-allocated address. The UE never receives an address that the PGW did not allocate. See [S2a GTP-C](#) for the Create Session exchange.

The PAA type sets which address OmniTWAG delivers.

PAA type	Address delivered to the UE
ipv4	The IPv4 address from the PAA.
ipv6	The IPv6 address from the PAA.
ipv4v6	The IPv4 address from the PAA.

If the Create Session Response has no PAA, OmniTWAG does not pre-allocate a lease. In that case the UE receives no address.

NSWO sessions: pool and dhcp

An NSWO session has no PGW, so OmniTWAG owns the address space. The address comes from the IP pool named by the credential `ip_pool_id`. A pool holds an IPv4 CIDR, a gateway, a DNS server list, and a lease time. See [IP Pools](#).

The `address_source` field of the credential selects how the pool address reaches the UE.

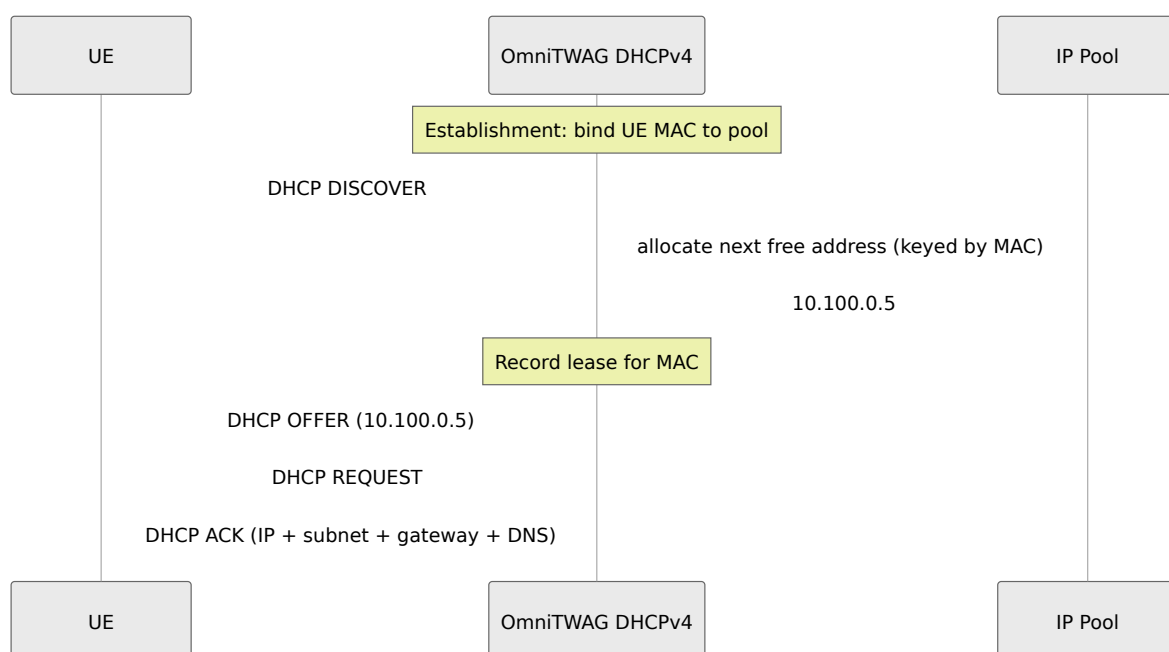
`address_source: :pool` (fixed)

At post-accept establishment OmniTWAG allocates one address from the pool, keyed by the subscriber identity. The allocation is idempotent, so the same identity keeps the same address across re-establishment. OmniTWAG pre-allocates a DHCPv4 lease for the UE MAC with that address, the pool gateway, the pool DNS servers, and a subnet mask derived from the pool CIDR. It also

registers the UE in the GRE data path at this point. When the UE sends DISCOVER, the server offers the pre-allocated address.

address_source: :dhcp (dish-out on DISCOVER)

At establishment OmniTWAG does not allocate an address yet. It **binds** the UE MAC to the pool. On the first DHCP DISCOVER from that MAC, the DHCPv4 server allocates the next free address from the pool range, records a lease, and offers it. This is the new dynamic dish-out path. The address is chosen at DISCOVER time, not at establishment.



On session teardown OmniTWAG releases the pool allocation and, for `:dhcp`, removes the pool binding and the dynamic lease. If the pool is missing or exhausted, or the AP has no `ip_pool_id`, OmniTWAG assigns no address and the UE self-addresses.

DHCPv4

The DHCPv4 server listens on UDP port `67`. It stores each lease in an in-memory ETS table, keyed by the UE MAC address. A lease reaches the table by one of two paths.

- **Pre-allocated.** The session process inserts a lease when the PGW allocates an address (tunneled) or when OmniTWAG allocates a fixed pool

address (NSWO `address_source: :pool`).

- **Dished out on DISCOVER.** For NSWO `address_source: :dhcp`, the session process instead binds the UE MAC to a pool. The server holds a per-MAC binding in memory. On the first DISCOVER from a bound MAC with no lease, the server allocates the next free address from the pool, records the lease, and offers it.

Lease lifecycle

For a pre-allocated lease, the server serves the address that already exists in the table for the requesting MAC. For a bound MAC, the server creates the lease on DISCOVER by dishing out from the pool.

Parse error on line 2: ...te (PGW PAA or pool :pool) [*] --> B -----^
Expecting 'SPACE', 'NL', 'HIDE_EMPTY', 'scale', 'COMPOSIT_STATE',
'STRUCT_STOP', 'STATE_DESCR', 'ID', 'FORK', 'JOIN', 'CHOICE', 'CONCURRENT',
'note', 'acc_title', 'acc_descr', 'acc_descr_multiline_value', 'CLICK', 'classDef',
'style', 'class', 'direction_tb', 'direction_bt', 'direction_rl', 'direction_lr',
'EDGE_STATE', got 'DESCR'

[Try again](#)

OmniTWAG handles these DHCP message types.

Message	Type code	OmniTWAG behaviour
DISCOVER	1	If a lease exists for the MAC, OmniTWAG sends an OFFER. If no lease exists but the MAC is bound to a pool, OmniTWAG dishes out the next free pool address, records a lease, and sends an OFFER. If no lease and no binding exist, OmniTWAG logs a warning and sends nothing.
REQUEST	3	If a lease exists and the requested IP (option 50) matches the lease, or the request has no requested IP, OmniTWAG sends an ACK. If the requested IP does not match, OmniTWAG sends a NAK. If no lease exists, OmniTWAG sends a NAK.
RELEASE	7	OmniTWAG marks the lease as <code>released</code> . OmniTWAG keeps the lease record. The PDN connection stays up, per TS 23.402 §16.1.5.2 NOTE 2 .
DECLINE	4	OmniTWAG logs a warning. OmniTWAG keeps the lease record.
INFORM	8	OmniTWAG sends an ACK with configuration only. The ACK carries the gateway, subnet mask, and DNS, but no address (<code>yiaddr</code> is <code>0.0.0.0</code>).

For INFORM, OmniTWAG uses the `default_gateway` and `default_subnet_mask` from configuration. OmniTWAG uses `8.8.8.8` as the DNS server in the INFORM reply.

Lease contents

Each lease holds the UE IP, the subnet mask, the gateway, the DNS servers, and the lease time. For a tunneled session, the session process sets the UE IP and the DNS servers from the PGW response, and the subnet mask, gateway, and lease time from the DHCP configuration. For an NSW0 session, these

values come from the IP pool: the IP from the pool range, the subnet mask from the pool CIDR prefix, and the gateway, DNS servers, and lease time from the pool record. See [IP Pools](#) and [Configuration](#).

A lease record also has an `allocated_at` time and an `expires_at` time. The `expires_at` time is `allocated_at` plus the lease time. OmniTWAG uses these values to report the lease status. A background reaper runs every 60 seconds and deletes any lease whose `expires_at` has passed. A lease with a lease time of zero has no `expires_at` and never expires. A lease is also removed when the session tears down.

DHCP options in a reply

OmniTWAG builds each OFFER and ACK with these DHCP options.

Option	Code	Content
Message Type	53	OFFER or ACK.
Server Identifier	54	The DHCP <code>server_ip</code> .
Subnet Mask	1	The lease subnet mask.
Router	3	The lease gateway.
Domain Name Server	6	The lease DNS servers. Omitted if the list is empty.
IP Address Lease Time	51	The lease time in seconds. Omitted if the lease time is zero.
Broadcast Address	28	Computed from the IP and the subnet mask. Sent only if the UE lists this option in its Parameter Request List (option 55).

DHCP transport over GRE

An access point that uses a GRE tunnel forwards the UE DHCP request inside a GRE-encapsulated Ethernet frame. The GRE server extracts the DHCP payload and passes it to the DHCPv4 server. OmniTWAG tags the packet with the source AP address. When OmniTWAG builds the reply, it wraps the DHCP payload in a UDP/IP/Ethernet frame and sends the frame back through the GRE tunnel to the same AP. The source is `10.0.0.1:67` and the destination is the broadcast address on the client port `68`.

A DHCP request that arrives directly on the UDP socket (not through GRE) is answered on the UDP socket. If the source port is not the standard client port `68`, OmniTWAG replies to the source address and port. This path supports test tools and relays. Otherwise OmniTWAG follows [RFC 2131](#): it uses the broadcast address if the broadcast flag is set or `ciaddr` is zero, and it uses `ciaddr` if that address is set.

DHCPv6

The DHCPv6 server listens on UDP port `547` over IPv6. It supports both a stateless mode and a stateful mode.

- **Stateless mode** answers an Information-Request (type 11) with an Information-Reply (type 12). The reply carries the DNS servers. This follows [RFC 3736](#) and [RFC 3646](#). A UE uses this mode when it sets the Other Configuration (O) flag from its RA processing.
- **Stateful mode** answers a Solicit (type 1) with an Advertise (type 2), and a Request (type 3) with a Reply (type 7). This mode delivers a pre-allocated IPv6 address in an IA_NA option. This follows [RFC 3315](#).

The server pre-allocates a stateful address per client DUID. The GTP control plane inserts the allocation. The server holds these allocations in an in-memory table, keyed by the client DUID.

DHCPv6 message handling

Message	Type	OmniTWAG behaviour
Information-Request	11	OmniTWAG sends an Information-Reply with the Server DUID, the echoed Client ID, and the DNS servers.
Solicit	1	OmniTWAG sends an Advertise. If the client DUID has an allocation, the Advertise carries the IA_NA address. If not, the Advertise carries a <code>NoAddrsAvail</code> status (code 2).
Request	3	OmniTWAG sends a Reply. The Reply confirms the IA_NA address for the client DUID.
Other types	-	OmniTWAG ignores the message.

Each reply echoes the client transaction ID, per [RFC 3315 §15](#). Each reply that includes an allocated address sets the DHCPv6 timers from the lease time. T1 is half the lease time. T2 is four fifths of the lease time.

Server DUID

OmniTWAG builds a Server DUID of type DUID-EN (Enterprise Number). It uses the 3GPP enterprise number `10415`. The identifier is derived from the node name, so it is stable for a given node.

DHCPv6 options in a reply

Option	Code	Content
Client Identifier	1	The Client ID echoed from the request, if present.
Server Identifier	2	The Server DUID.
Identity Association for Non-temporary Addresses (IA_NA)	3	The allocated IPv6 address, or a <code>NoAddrsAvail</code> status. Stateful mode only.
DNS Recursive Name Server	23	The configured DNS servers. Omitted if the list is empty.
Domain Search List	24	The configured search list. Sent only if <code>dhcpv6_domain_list</code> is set.

IPv6 Router Advertisement (SLAAC)

The IPv6 RA server delivers PGW-allocated IPv6 prefixes to UEs with Stateless Address Autoconfiguration (SLAAC). It implements ICMPv6 Router Advertisement per [RFC 4861](#).

The GTP control plane adds a prefix per interface. The server keeps the prefixes in memory, grouped by interface name. The server sends a Router Advertisement in two cases:

1. On a periodic timer. The default interval is `30` seconds.
2. On demand, when the control plane calls for an RA on an interface.

The server also parses Router Solicitation (ICMPv6 type 133) messages.

Router Advertisement contents

Each Router Advertisement carries these fields and options.

Field or option	Value
Current Hop Limit	From <code>hop_limit</code> . Default <code>64</code> .
Router Lifetime	From <code>router_lifetime</code> . Default <code>1800</code> seconds.
M and O flags	Both <code>0</code> . The RA does not signal managed address configuration or other configuration.
Source Link-Layer Address (option 1)	The source MAC, if it is a 6-byte value.
MTU (option 5)	Included only if an MTU is supplied.
Prefix Information (option 3)	One per prefix. The L (on-link) and A (autonomous) flags are both set.
Recursive DNS Server, RDNSS (option 25)	The DNS servers from the prefixes, per RFC 8106 . Omitted if there are no DNS servers.

Each Prefix Information option carries the prefix, the prefix length, the valid lifetime (default `86400` seconds), and the preferred lifetime (default `14400` seconds). Because the A flag is set, the UE forms its own IPv6 address from the prefix. The RDNSS lifetime is the largest valid lifetime across the prefixes.

DNS and PCO delivery

OmniTWAG delivers DNS and P-CSCF information in more than one way. The source is the PGW PCO where possible, and configuration otherwise.

PCO from the PGW

The PCO parser reads the Protocol Configuration Options from the Create Session Response, per [3GPP TS 24.008](#). The parser returns these fields.

Field	PCO protocol ID	Meaning
<code>dns_ipv4</code>	<code>0x000D</code>	IPv4 DNS server addresses.
<code>dns_ipv6</code>	<code>0x0003</code>	IPv6 DNS server addresses.
<code>pcscf_ipv4</code>	<code>0x000C</code>	IPv4 P-CSCF addresses.
<code>pcscf_ipv6</code>	<code>0x0001</code>	IPv6 P-CSCF addresses.
<code>mtu</code>	<code>0x0010</code>	IPv4 link MTU.
<code>ip_alloc_nas</code>	<code>0x000A</code>	A flag. It is <code>true</code> if the PGW signals IP allocation through NAS signalling.

The parser also keeps every parsed option in a `raw` list. If the PCO binary is malformed part way through, the parser returns the fields it read so far. It does not fail the whole parse.

OmniTWAG can also build a PCO request for the Create Session Request. The request lists the options that OmniTWAG wants the PGW to return, for example the IPv4 DNS servers and the link MTU.

DNS delivery to the UE

The DNS servers reach the UE by the transport that the UE uses.

Transport	DNS source	Notes
DHCPv4 (option 6)	The IPv4 DNS servers from the PCO.	If the PCO has no IPv4 DNS server, OmniTWAG uses <code>8.8.8.8</code> .
DHCPv6 (option 23)	The DHCPv6 DNS server list.	Set from the PGW PCO or from <code>dhcpv6_dns_servers</code> . Defaults to Google Public DNS IPv6 if not set.
IPv6 RA (RDNSS, option 25)	The IPv6 DNS servers attached to the advertised prefix.	Delivered with the prefix.

The DHCPv4 lease takes the IPv4 DNS servers from the PCO `dns_ipv4` list. If that list is empty, OmniTWAG falls back to `8.8.8.8` for the lease.

Note on P-CSCF. The PCO parser reads P-CSCF addresses from the PGW. The current DHCP, DHCPv6, and RA delivery paths do not add a P-CSCF DHCP option or RA option. The P-CSCF values are parsed and available, but they are not delivered to the UE through this subsystem in the audited code.

Address pools and exhaustion

Exhaustion behaviour depends on the mode.

- **Tunneled (home-routed).** There is no local address pool. The PGW owns the address space. OmniTWAG delivers exactly the address that the PGW allocates for each session. There is no local range to exhaust. If the PGW does not return a PAA, the UE receives no address, and OmniTWAG logs the condition.
- **NSWO.** OmniTWAG allocates from a local IP pool, so a pool can run out. A pool is an IPv4 CIDR of at most a `/30`. The allocator skips the network and broadcast addresses and hands out the first free host. When every host is

in use, the allocator returns `pool_exhausted`. For `address_source: :pool` the UE then self-addresses. For `address_source: :dhcp` the DISCOVER goes unanswered and OmniTWAG logs the exhaustion. Allocations are in-memory and rebuild as sessions re-establish, the same as DHCP leases. See [IP Pools](#) for pool definition and capacity.

The `Twag.Cidr` helper does the subnet math for the platform. It parses IPv4 and IPv6 CIDR strings and it tests whether an address is inside a subnet. The RADIUS subsystem uses this helper for source-IP authorization. The IP pool store uses it to derive the pool subnet mask, the usable host range, and the default gateway.

Configuration

The IP-management servers read their configuration from the `:omnitwag` application config. The main blocks are `dhcp` and `ipv6_ra`. DHCPv6 reads a small set of top-level keys.

```
config :omnitwag,  
  # DHCPv4 server  
  dhcp: %{  
    enabled: true,  
    interface: "wlan0",  
    server_ip: "10.0.0.1",  
    default_lease_time: 3600,  
    default_subnet_mask: "255.255.255.0",  
    default_gateway: "10.0.0.1"  
  },  
  
  # IPv6 Router Advertisement  
  ipv6_ra: %{  
    enabled: true,  
    interface: "wlan0",  
    ra_interval: 30,  
    hop_limit: 64,  
    router_lifetime: 1800  
  },  
  
  # DHCPv6 (optional top-level keys)  
  dhcpv6_dns_servers: ["2001:4860:4860::8888",  
    "2001:4860:4860::8844"],  
  dhcpv6_domain_list: ["example.com"]
```

DHCPv4 parameters (dhcp)

Parameter	Type	Required	Default	Description
enabled	Boolean	No	true	Enable the DHCPv4 server. Set false to disable it, example in test environment
interface	String	No	wlan0	Network interface for the DHCP service. Use gre for GRE tunnel mode
server_ip	String	No	10.0.0.1	The DHCP server identifier. OmniTWA sends this value in DHCP option 54 and as the source GRE-tunnelled replies.
default_lease_time	Integer	No	3600	Lease time in seconds. OmniTWA applies this

Parameter	Type	Required	Default	Description
				value to each pre-allocated lease.
default_subnet_mask	String	No	255.255.255.0	Subnet mask for the lease and for the RFC 2131 broadcast calculation.
default_gateway	String	No	10.0.0.1	Default gateway for the lease (DHCP option 3).

IPv6 RA parameters (`ipv6_ra`)

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>true</code>	Enable the IPv6 RA server. Set to <code>false</code> to disable it.
<code>interface</code>	String	No	<code>wlan0</code>	Network interface for Router Advertisements.
<code>ra_interval</code>	Integer	No	<code>30</code>	Interval in seconds between periodic Router Advertisements.
<code>hop_limit</code>	Integer	No	<code>64</code>	Current Hop Limit field in the Router Advertisement.
<code>router_lifetime</code>	Integer	No	<code>1800</code>	Router Lifetime in seconds in the Router Advertisement.

DHCPv6 parameters (top-level)

Parameter	Type	Required	Default	Description
<code>dhcpv6_dns_servers</code>	List of String	No	Google Public DNS IPv6 (<code>2001:4860:4860::8888</code> , <code>2001:4860:4860::8844</code>)	IPv6 servers that the DHCPv6 client can use to resolve domain names. This parameter is optional.
<code>dhcpv6_domain_list</code>	List of String	No	(unset)	DNS domain list for DHCPv6 clients. The DHCPv6 client will use the first domain in the list to resolve domain names. Omitting this parameter will result in the DHCPv6 client not resolving domain names.

Observability

Web interface

The control panel has a **DHCP Pool** page at `/dhcp`. The page lists every DHCPv4 lease with the UE MAC address, the IP address, the lease start time, the lease expiry time, and the status. The status is `Active` or `Expired`. The page shows a live countdown for each active lease. Select a lease row to see the full lease details: subnet mask, gateway, DNS servers, lease duration, and timestamps. The page refreshes every second. If the DHCPv4 server is not running, the page shows `Server Not Running`.

REST API

The REST API serves DHCPv4 lease data under the `DHCP` tag.

Method	Path	Description
GET	<code>/dhcp</code> (index)	List all DHCPv4 leases with a summary count.
GET	<code>/dhcp/{id}</code> (show)	Get one lease by MAC address. The <code>id</code> is the colon-separated MAC.

The index response has an `enabled` flag, the `leases` list, a total `count`, an `active` count, and an `expired` count. If the DHCPv4 server is not running, the index returns `enabled: false` with an empty list. The show endpoint returns `404` if no lease matches the MAC, and `503` if the server is not running.

Each lease object has these fields.

Field	Description
<code>mac</code>	UE MAC address, colon-separated.
<code>ip</code>	Assigned IPv4 address.
<code>subnet_mask</code>	Lease subnet mask.
<code>gateway</code>	Lease gateway.
<code>dns_servers</code>	List of DNS servers.
<code>lease_time</code>	Lease duration in seconds.
<code>allocated_at</code>	Lease start time, in seconds since the Unix epoch.
<code>expires_at</code>	Lease expiry time, in seconds since the Unix epoch.
<code>status</code>	<code>active</code> or <code>expired</code> .
<code>time_remaining</code>	Seconds until expiry, or <code>0</code> if expired.

Telemetry events

The IP-management servers emit telemetry events. An operator can attach these events to a metrics exporter. See the [Metrics Reference](#) for the Prometheus endpoint.

Event	Source	Meaning
<code>[:dhcp, :lease, :allocate]</code>	DHCPv4	A lease was pre-allocated for a MAC.
<code>[:dhcp, :lease, :deallocate]</code>	DHCPv4	A lease was removed on teardown.
<code>[:dhcp, :lease, :reap]</code>	DHCPv4	An expired lease was removed by the background reaper.
<code>[:dhcp, :discover]</code>	DHCPv4	A DISCOVER was answered with an OFFER from an existing lease.
<code>[:dhcp, :discover, :pool_allocated]</code>	DHCPv4	A DISCOVER from a pool-bound MAC was answered by dishing out a new pool address (NSWO <code>:dhcp</code>).
<code>[:dhcp, :discover, :no_allocation]</code>	DHCPv4	A DISCOVER arrived for a MAC with no lease and no pool binding.
<code>[:dhcp, :ack]</code>	DHCPv4	A REQUEST was answered with an ACK.
<code>[:dhcp, :nak]</code>	DHCPv4	A REQUEST was answered with a NAK.
<code>[:dhcp, :release]</code>	DHCPv4	A client released its lease.
<code>[:dhcp, :decline]</code>	DHCPv4	A client declined its lease.
<code>[:dhcp, :inform]</code>	DHCPv4	An INFORM was answered.
<code>[:dhcp, :packet, :malformed]</code>	DHCPv4	A malformed DHCP packet was received.

Event	Source	Meaning
[:dhcpv6, :info_request, :received]	DHCPv6	An Information-Request was received.
[:dhcpv6, :solicit, :received]	DHCPv6	A Solicit was received.
[:dhcpv6, :request, :received]	DHCPv6	A Request was received.
[:dhcpv6, :info_reply, :sent]	DHCPv6	An Information-Reply was sent.
[:dhcpv6, :advertise, :sent]	DHCPv6	An Advertise was sent.
[:dhcpv6, :reply, :sent]	DHCPv6	A Reply was sent.
[:ipv6_ra, :prefix, :add]	IPv6 RA	A prefix was added to an interface.
[:ipv6_ra, :prefix, :remove]	IPv6 RA	A prefix was removed from an interface.
[:ipv6_ra, :advertisement, :sent]	IPv6 RA	A Router Advertisement was sent.

Troubleshooting

The UE gets no IP address

Symptoms: The UE authenticates but does not get an IPv4 address. The DHCP Pool page shows no lease for the UE MAC. The `[ :dhcp, :discover, :no_allocation]` event fires.

Possible causes (tunneled):

- The GTP Create Session Response had no PAA, so OmniTWAG did not pre-allocate a lease.
- The GTP-C session did not establish, so no lease was created.

Possible causes (NSWO):

- The AP credential has no `ip_pool_id`, so OmniTWAG assigns no address.
- The pool is exhausted, so no free address could be dished out.
- For `address_source: :dhcp`, the DISCOVER arrived before the MAC was bound to the pool, or the bind did not run because the session had no MAC yet.

Common cause:

- The DHCPv4 server is not running.

Resolution:

1. Check the OmniTWAG logs for the establishment path (tunneled versus NSWO) and the UE IP.
2. For tunneled sessions, confirm that the PGW returns a PAA for the APN.
3. For NSWO sessions, confirm the AP credential names a valid `ip_pool_id` and that the pool has free capacity. See [IP Pools](#) and [RADIUS Credentials](#).
4. Confirm that the DHCP Pool page shows the DHCPv4 server as running.

The UE gets an address but no DNS

Symptoms: The UE has an IPv4 address but cannot resolve names.

Possible causes:

- The PGW PCO returned no IPv4 DNS server. In that case OmniTWAG uses 8.8.8.8.
- The UE cannot reach the DNS server that the PGW supplied.

Resolution:

1. Read the lease DNS servers on the DHCP Pool page or through the REST API.
2. Confirm that the PGW PCO returns the expected DNS server.
3. Confirm that the DNS server is reachable from the UE data path.

The UE receives a NAK

Symptoms: The UE sends a REQUEST and receives a NAK. The [:dhcp, :nak] event fires.

Possible causes:

- The UE requested an IP (option 50) that does not match its pre-allocated lease.
- The lease was removed, for example after a session teardown.

Resolution:

1. Confirm that a lease exists for the UE MAC.
2. Confirm that the UE requests the same IP that the PGW or the pool allocated.
3. If the session tore down, let the UE restart the DHCP exchange after it re-authenticates.

Related pages

- [Overview](#)
- [IP Pools](#)
- [RADIUS Credentials](#)

- Sessions
- S2a GTP-C
- User Plane
- Online Charging (Gy)

Overview

IP Pools

Overview

OmniTWAG assigns IPv4 addresses to NSW0 (Non-Seamless WLAN Offload) sessions from an **IP pool**. NSW0 is TWAG-anchored local breakout: OmniTWAG terminates the user traffic and breaks it out locally. An IP pool defines an IPv4 CIDR block, a gateway, DNS servers, and a lease time. The operator adds, changes, and removes pools on demand through the REST API or the control-panel web interface.

Pool definitions are persistent. OmniTWAG stores them in a DETS file, so pools survive a restart. Address allocations are in-memory. OmniTWAG rebuilds the allocations as sessions re-establish after a restart.

Table of Contents

- [Concepts](#)
- [How a session gets an address](#)
- [Address allocation](#)
- [Pool model](#)
- [REST API](#)
- [Web interface](#)
- [Troubleshooting](#)

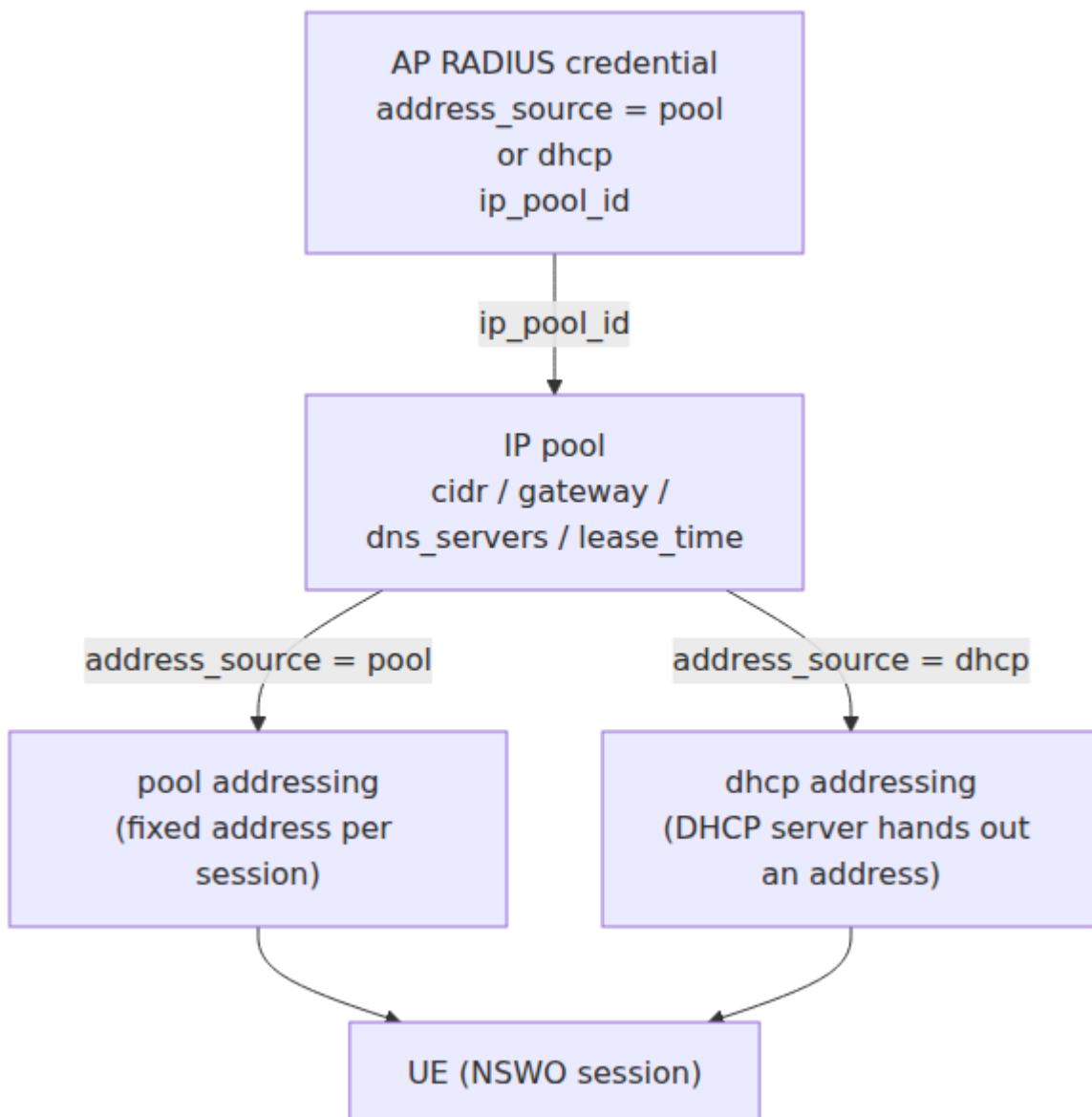
Concepts

A **pool** is a set of addresses from one IPv4 CIDR block. Each pool has a gateway, a list of DNS servers, and a lease time.

An IP pool is linked to an access point through the AP's RADIUS credential. The credential has an `ip_pool_id` field. This field holds the `id` of the pool. See [RADIUS Shared Secrets](#) for the credential model. NSW0 sessions from that access point are addressed from the linked pool.

Each credential also has an `address_source` field. This field selects how OmniTWAG gives the address to the UE (user equipment). There are two address sources:

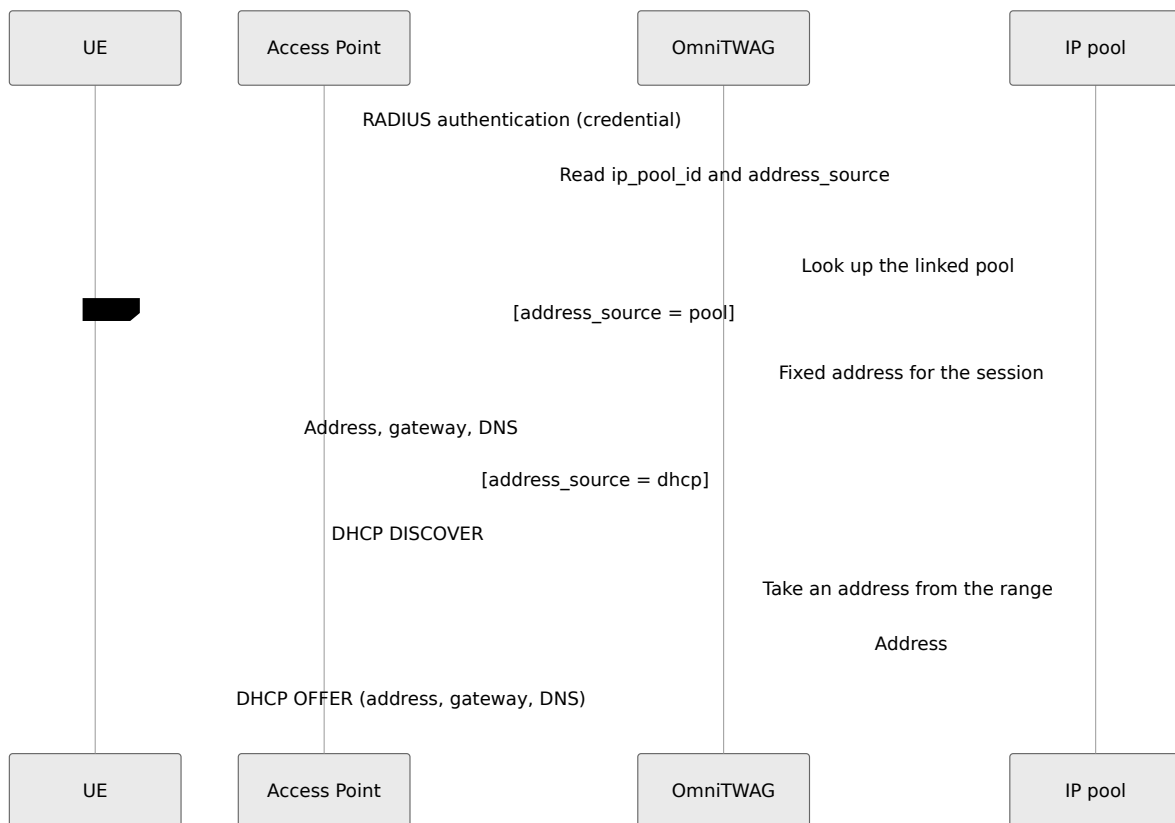
Address source	Behaviour
<code>pool</code>	OmniTWAG pre-allocates a fixed address per session from the pool. The address is chosen when the session is set up.
<code>dhcp</code>	The OmniTWAG DHCP server hands out an address from the pool range in reply to a DHCP DISCOVER from the UE.



How a session gets an address

OmniTWAG addresses each NSWO session from the pool that is linked to the AP credential. The steps are:

1. The access point authenticates through its RADIUS credential.
2. OmniTWAG reads the `ip_pool_id` field on the credential to find the pool.
3. OmniTWAG reads the `address_source` field on the credential.
4. If `address_source` is `pool`, OmniTWAG pre-allocates a fixed address for the session from the pool.
5. If `address_source` is `dhcp`, OmniTWAG waits for a DHCP DISCOVER from the UE. The DHCP server then hands out an address from the pool range.
6. OmniTWAG gives the UE the address, the gateway, and the DNS servers from the pool.



Address allocation

OmniTWAG allocates hosts from the pool CIDR. Allocation skips the network address and the broadcast address. The first usable host is the network address plus one.

The pool CIDR prefix must be `/30` or larger to have usable hosts. A `/30` block has 2 usable hosts. A smaller prefix has fewer usable hosts, so OmniTWAG rejects it.

When every usable host is allocated, OmniTWAG returns `pool_exhausted`. No further session from the linked access point can get an address until an allocation is freed.

The `allocated` field on the pool holds the current count of allocations. OmniTWAG computes this count from the in-memory allocations.

Pool model

Each pool has these fields.

Field	Type	Description
<code>id</code>	String	Unique identifier (UUID). The server sets this value.
<code>name</code>	String	Human-readable label, for example <code>Melbourne guest WiFi</code> .
<code>cidr</code>	String	IPv4 CIDR block for the pool, for example <code>10.60.0.0/24</code> . The prefix must be <code>/30</code> or larger to have usable hosts.
<code>gateway</code>	String	IPv4 gateway address given to the UE. Defaults to the first usable host of the CIDR if omitted.
<code>dns_servers</code>	List of String	IPv4 DNS server addresses given to the UE.
<code>lease_time</code>	Integer	Lease time in seconds. Defaults to <code>3600</code> .
<code>default</code>	Boolean	When <code>true</code> , this pool is the DHCP default range. The DHCP server dishes out from it for any client with no pre-allocated lease and no pool binding. Only one pool is the default at a time. Defaults to <code>false</code> .
<code>allocated</code>	Integer	Count of current allocations. The server computes this value.
<code>inserted_at</code>	Integer	Time the pool was created, in milliseconds since the Unix epoch. The server sets this value.
<code>updated_at</code>	Integer	Time the pool was last changed, in milliseconds since the Unix epoch. The server sets this value.

The operator sets `name`, `cidr`, `gateway`, `dns_servers`, and `lease_time`. The server sets `id`, `allocated`, `inserted_at`, and `updated_at`.

REST API

The OmniTWAG REST API serves these endpoints over HTTPS on the configured API port (8444 by default). The base path is `/ip_pools`.

Method	Path	Action	Description
GET	<code>/ip_pools</code>	index	List all pools. The result is paginated and searchable by <code>name</code> or <code>cidr</code> .
GET	<code>/ip_pools/{id}</code>	show	Get one pool by <code>id</code> .
POST	<code>/ip_pools</code>	create	Create a pool.
PUT	<code>/ip_pools/{id}</code>	update	Change a pool.
DELETE	<code>/ip_pools/{id}</code>	delete	Remove a pool.

Request body

`POST` and `PUT` accept a JSON body with these fields.

Field	Type	Required	Description
<code>name</code>	String	No	Human-readable label.
<code>cidr</code>	String	Yes	IPv4 CIDR block. The prefix must be <code>/30</code> or larger.
<code>gateway</code>	String	No	IPv4 gateway address. Defaults to the first usable host of the CIDR if omitted.
<code>dns_servers</code>	List of String	No	IPv4 DNS server addresses.
<code>lease_time</code>	Integer	No	Lease time in seconds. Defaults to <code>3600</code> .
<code>default</code>	Boolean	No	Mark this pool as the DHCP default range. Setting it clears the flag on any other pool.

`PUT` changes only the fields in the body. Fields that are not in the body keep their current value.

Responses

Status	Meaning
200 OK	The request was successful (index, show, update, delete).
201 Created	The pool was created.
404 Not Found	No pool has the given <code>id</code> .
422 Unprocessable Entity	The <code>cidr</code> is not a valid IPv4 CIDR block, or the prefix is too small to have usable hosts.

Example: list pools

Request:

```
GET /ip_pools
```

Response:

```
{
  "count": 1,
  "ip_pools": [
    {
      "id": "0d1e2f3a-4b5c-6d7e-8f90-1a2b3c4d5e6f",
      "name": "Melbourne guest WiFi",
      "cidr": "10.60.0.0/24",
      "gateway": "10.60.0.1",
      "dns_servers": ["8.8.8.8", "8.8.4.4"],
      "lease_time": 3600,
      "allocated": 12,
      "inserted_at": 1734512400000,
      "updated_at": 1734512400000
    }
  ]
}
```

Example: create a pool

Request:

```
POST /ip_pools
```

```
{
  "name": "Melbourne guest WiFi",
  "cidr": "10.60.0.0/24",
  "gateway": "10.60.0.1",
  "dns_servers": ["8.8.8.8", "8.8.4.4"],
  "lease_time": 3600
}
```

A successful request returns `201 Created` with the new pool, including its `id`. If `gateway` is omitted, the pool uses the first usable host of the CIDR (`10.60.0.1` in this example).

Example: a CIDR that is too small

Request:

```
{
  "name": "Broken pool",
  "cidr": "10.60.0.0/31"
}
```

Response (422 Unprocessable Entity):

```
{
  "error": "cidr too small: 10.60.0.0/31 has no usable hosts"
}
```

Default pool

One pool can be the DHCP default range. The DHCP server dishes out an address from the default pool for any client that has no pre-allocated lease and no explicit pool binding. Use the default pool to give an API-defined range to the base DHCP service, not only to NSW0 sessions.

Set the default with the `default` field, or the checkbox on the web page. Only one pool is the default at a time. When you mark a pool as the default, OmniTWAG clears the flag on the previous default pool.

Each active lease records its source. See the [IP Leases](#) view: a lease shows `PGW (tunneled)`, a named `NSW0 pool`, or the `default pool`.

Web interface

The control panel has an **IP Pools** admin page. The page lists every pool with its name, CIDR, gateway, and allocated count. The page has controls to add, edit, and delete a pool.

The former **DHCP Pool** page is now labelled **IP Leases**. The IP Leases page shows the active leases.

Troubleshooting

A pool is exhausted

Symptoms: New NSWO sessions from an access point cannot get an address. OmniTWAG returns `pool_exhausted`.

Possible causes:

- Every usable host in the pool CIDR is allocated.
- The pool CIDR is too small for the number of active sessions.

Resolution:

1. Check the `allocated` count against the number of usable hosts in the CIDR.
2. Widen the pool CIDR to add more usable hosts, or free stale allocations by ending idle sessions.
3. Confirm that the lease time is not so long that freed addresses stay allocated.

An access point does not use a pool

Symptoms: Sessions from an access point are not addressed from any IP pool.

Possible causes:

- The access point uses local breakout owned by the access point, not TWAG-anchored NSWO. A local, AP-owned-breakout access point addresses its own clients and does not use an OmniTWAG pool.
- The AP credential has no `ip_pool_id`, so no pool is linked.

Resolution:

1. Confirm the access point uses TWAG-anchored NSWO. Only NSWO sessions use a pool.
2. If the access point should use a pool, set `ip_pool_id` on its RADIUS credential to the `id` of the pool. See [RADIUS Shared Secrets](#).

The UE gets no address

Symptoms: The UE connects but receives no IPv4 address.

Possible causes:

- The credential `address_source` does not match how the UE requests an address. With `dhcp`, the UE must send a DHCP DISCOVER. With `pool`, OmniTWAG pre-allocates the address.
- The linked pool is exhausted.
- The `ip_pool_id` on the credential points to a pool that does not exist.

Resolution:

1. Confirm the `address_source` on the credential matches the UE behaviour.
2. Confirm the linked pool has free addresses. Check the `allocated` count.
3. Confirm the `ip_pool_id` on the credential points to a valid pool.

Overview

TWAG Metrics Reference

Overview

TWAG exposes Prometheus metrics on port **9568** (set by `prometheus.port`). Gauge metrics are refreshed by a poller every 5 seconds. Counter metrics are emitted from the code path that triggers them.

Table of Contents

- [Metrics Endpoint](#)
- [RADIUS Authentication Metrics](#)
- [RADIUS Authorization and Rate-Limit Metrics](#)
- [RADIUS Accounting Metrics](#)
- [EAP Metrics](#)
- [Client, AP, and Session Gauges](#)
- [Data Throughput Metrics](#)
- [TWAG Session and Data-path Metrics](#)
- [Gy Online Charging Metrics](#)
- [STa Diameter Metrics](#)
- [Diameter Message Metric](#)
- [NSWO Metrics](#)
- [DHCP Metrics](#)
- [DHCPv6 and IPv6 RA Metrics](#)
- [WLCP Metrics](#)
- [Erlang VM Metrics](#)
- [Example Grafana Queries](#)
- [Alerting Rules](#)
- [Related Pages](#)

Metrics Endpoint

```
http://<twag-host>:9568/metrics
```

Prometheus flattens each event name into a metric name by replacing dots with underscores. For example the event `[ :radius, :access, :request, :count]` is exposed as `radius_access_request_count`.

RADIUS Authentication Metrics

These counters track RADIUS Access-Request/Response flows per [RFC 2865](#).

Metric	Type	Labels	Description
<code>radius_access_request_count</code>	Counter	none	RADIUS Access-Request packets received.
<code>radius_access_accept_count</code>	Counter	none	RADIUS Access-Accept packets sent (successful auth).
<code>radius_access_reject_count</code>	Counter	none	RADIUS Access-Reject packets sent (failed auth).
<code>radius_access_challenge_count</code>	Counter	none	RADIUS Access-Challenge packets sent (EAP exchange in progress).

Example queries:

```
# Request rate per second
rate(radius_access_request_count[5m])

# Challenge rate (active EAP negotiations)
rate(radius_access_challenge_count[5m])

# Authentication success ratio
sum(rate(radius_access_accept_count[5m])) /
  (sum(rate(radius_access_accept_count[5m])) +
   sum(rate(radius_access_reject_count[5m])))
```

RADIUS Authorization and Rate-Limit Metrics

These counters track packets that are dropped or flagged before or during authentication. See [RADIUS Rejections](#) and [RADIUS Shared Secrets](#).

Metric	Type	Labels	
<code>radius_auth_unauthorized_source_count</code>	Counter	<code>source</code>	Pac tha cre (un is c
<code>radius_auth_bad_secret_count</code>	Counter	<code>source</code> , <code>credential_id</code>	Pac ma fail Aut (wr
<code>radius_access_rate_limited_count</code>	Counter	<code>nas_ip</code>	Acc bec exc ra in t

Example queries:

```
# Unknown APs appearing (add a credential for these)
increase(radius_auth_unauthorized_source_count[1h])

# APs with a wrong shared secret
sum by (source) (increase(radius_auth_bad_secret_count[1h]))

# Rate-limited NAS devices
sum by (nas_ip) (increase(radius_access_rate_limited_count[15m]))
```

RADIUS Accounting Metrics

Tracks RADIUS Accounting messages per [RFC 2866](#).

radius_accounting_request_count

Type: Counter

Label	Description
status_type	Accounting status type (see below).

Status type values: start, stop, interim_update, accounting_on, accounting_off.

Example queries:

```
# Session starts per minute
rate(radius_accounting_request_count{status_type="start"}[5m]) *
60

# Session stops per minute
rate(radius_accounting_request_count{status_type="stop"}[5m]) * 60
```

EAP Metrics

These counters track EAP-AKA phases per [RFC 4187](#), plus EAP-SIM, EAP re-authentication (ERP), and generic EAP failures.

Metric	Type	Description
<code>eap_aka_identity_count</code>	Counter	EAP-AKA Identity exchanges (Phase 1).
<code>eap_aka_challenge_count</code>	Counter	EAP-AKA Challenge exchanges (Phase 2).
<code>eap_aka_sync_failure_count</code>	Counter	EAP-AKA synchronization failures (SQN out of sync, triggers resync).
<code>eap_aka_auth_success_count</code>	Counter	Successful EAP-AKA authentications.
<code>eap_aka_auth_reject_count</code>	Counter	Rejected EAP-AKA authentications (UE failed to authenticate the network / AUTN failure).
<code>eap_aka_client_error_count</code>	Counter	EAP-AKA client-error notifications received from the UE.
<code>eap_auth_rejected_count</code>	Counter	Expected auth denials (for example HSS USER_UNKNOWN) that reject the UE cleanly.
<code>eap_process_crash_count</code>	Counter	Unexpected errors while processing an EAP message (UE rejected, session preserved).
<code>eap_sim_start_count</code>	Counter	EAP-SIM Start exchanges.
<code>eap_sim_auth_success_count</code>	Counter	Successful EAP-SIM authentications.

Metric	Type	Description
<code>eap_sim_auth_reject_count</code>	Counter	Rejected EAP-SIM authentications.
<code>eap_erp_initiate_count</code>	Counter	EAP ERP re-authentication initiations.
<code>eap_erp_success_count</code>	Counter	Successful EAP ERP re-authentications.
<code>eap_erp_mac_failure_count</code>	Counter	EAP ERP MAC verification failures.
<code>eap_erp_fallback_count</code>	Counter	EAP ERP fallbacks to full authentication. Label <code>reason</code> .

Example queries:

```
# Sync failures (should be rare - alert if > 0)
increase(eap_aka_sync_failure_count[1h])

# Successful AKA authentications per minute
rate(eap_aka_auth_success_count[5m]) * 60

# EAP processing crashes (should stay flat)
increase(eap_process_crash_count[1h])
```

Client, AP, and Session Gauges

These gauges are sampled every 5 seconds by the poller.

Metric	Type	Labels	Description
<code>radius_active_clients_count</code>	Gauge	none	Currently authenticated clients (live <code>Twag.Session.Client</code> processes).
<code>radius_access_points_count</code>	Gauge	none	Registered access points (<code>APState</code> entries).
<code>radius_clients_per_ap_count</code>	Gauge	<code>nas_ip</code> , <code>ssid</code>	Clients connected per access point.
<code>radius_active_sessions_count</code>	Gauge	<code>status</code>	Sessions grouped by accounting status (<code>start</code> , <code>stop</code> , <code>interim_update</code>).
<code>radius_active_sessions_total</code>	Gauge	none	Sessions in <code>start</code> or <code>interim_update</code> status.

Example queries:

```
# Top 5 busiest APs
topk(5, radius_clients_per_ap_count)

# Clients by SSID
sum by (ssid) (radius_clients_per_ap_count)

# Alert on a sudden drop in active sessions
delta(radius_active_sessions_total[5m]) < -10
```

Data Throughput Metrics

These gauges are summed across all tracked sessions from the RADIUS accounting counters.

Metric	Type	Unit	Description
<code>radius_data_bytes_in</code>	Gauge	bytes	Total bytes received across sessions (Acct-Input-Octets).
<code>radius_data_bytes_out</code>	Gauge	bytes	Total bytes sent across sessions (Acct-Output-Octets).
<code>radius_session_total_time</code>	Gauge	seconds	Total session time across sessions (Acct-Session-Time).

Example queries:

```
# Total throughput in MB
(radius_data_bytes_in + radius_data_bytes_out) / 1024 / 1024

# Average session duration
radius_session_total_time / radius_active_sessions_total
```

TWAG Session and Data-path Metrics

These counters track per-session lifecycle events emitted by `Twag.Session.Client`.

Metric	Type	Labels	Description
<code>twag_session_tearardown_count</code>	Counter	<code>reason</code>	Session tear down events. Reasons: <code>accounting</code> <code>admin_disconnect</code> <code>network_disconnect</code> <code>identity_disconnect</code> <code>liveness_disconnect</code>
<code>twag_bearer_modify_command_count</code>	Counter	none	PGW Modify Bearer Command applied to session.
<code>twag_bearer_update_count</code>	Counter	none	PGW Update Bearer events applied to session.
<code>twag_detach_cleanup_start_count</code>	Counter	none	Detach cleanup starts.
<code>twag_detach_cleanup_complete_count</code>	Counter	none	Detach cleanup completion.
<code>twag_detach_pgw_delete_count</code>	Counter	none	PGW Delete requests due to detach.
<code>twag_detach_liveness_timeout_count</code>	Counter	none	Detaches triggered by a liveness timeout.
<code>twag_liveness_arp_probe_success_count</code>	Counter	none	Successful ARP probes.
<code>twag_liveness_arp_probe_failure_count</code>	Counter	none	Failed liveness probes.

Metric	Type	Labels	Description
<code>twag_handover_in_count</code>	Counter	<code>source</code>	Handover-in
<code>twag_handover_out_count</code>	Counter	<code>cause</code>	Handover-out
<code>twag_handover_ap_change_count</code>	Counter	none	Intra-session change events

Example queries:

```
# Teardown reasons over the last hour
sum by (reason) (increase(twag_session_teardown_count[1h]))

# Sessions reaped for never completing auth
increase(twag_session_teardown_count{reason="identity_timeout"}[1h])

# Sessions dropped because the UE went silent
increase(twag_detach_liveness_timeout_count[1h])
```

Note: the Gy quota events `twag_gy_quota_blocked` and `twag_gy_quota_unblocked` are emitted by `Twag.Session.Client` but are not declared as Prometheus metrics today, so they do not appear on the `/metrics` endpoint. They are visible in the structured logs.

Gy Online Charging Metrics

These counters track Diameter Ro/Gy credit control, used only for NSWOL local breakout. See [Online Charging](#).

Metric	Type	Labels	Description
<code>twag_gy_ccr_sent_count</code>	Counter	<code>type</code>	Credit-Control-Requests sent to the OCS. <code>type</code> is <code>initial</code> , <code>update</code> , or <code>terminate</code> .
<code>twag_gy_ccr_error_count</code>	Counter	<code>type</code>	CCR send or build failures.
<code>twag_gy_cca_received_count</code>	Counter	<code>type</code> , <code>result</code>	Credit-Control-Answers received from the OCS.

Example queries:

```
# CCR send rate by type
sum by (type) (rate(twag_gy_ccr_sent_count[5m]))

# CCR failures
sum by (type) (increase(twag_gy_ccr_error_count[1h]))

# CCA results
sum by (result) (rate(twag_gy_cca_received_count[5m]))
```

STa Diameter Metrics

These counters track STa inbound requests handled by the TWAG.

Metric	Type	Labels	Description
<code>sta_asr_received_count</code>	Counter	none	Abort-Session-Requests received.
<code>sta_rar_received_count</code>	Counter	none	Re-Auth-Requests received.
<code>sta_str_received_count</code>	Counter	none	Session-Termination-Requests received.
<code>sta_session_disconnect_count</code>	Counter	<code>reason</code>	STa session disconnects.
<code>sta_session_reauth_count</code>	Counter	<code>re_auth_type</code>	STa session re-authentication

Diameter Message Metric

diameter_message_count

Type: Counter

Label	Description
<code>application</code>	Diameter application (for example <code>swx</code> , <code>sta</code>).
<code>command</code>	Command name (for example <code>MAR</code> , <code>MAA</code>).
<code>direction</code>	<code>inbound</code> or <code>outbound</code> .

Example queries:

```
# Diameter message rate by application
sum by (application) (rate(diameter_message_count[5m]))
```

NSWO Metrics

Metric	Type	Description
<code>nsw_allocate_count</code>	Counter	NSWO local IP allocations.
<code>nsw_release_count</code>	Counter	NSWO local IP releases.

DHCP Metrics

These counters track the DHCPv4 server. The `discover.pool_allocated` event covers the NSWO dish-out path, where the server allocates a fresh address from a bound pool on DISCOVER.

Metric	Type	Description
<code>dhcp_lease_allocate_count</code>	Counter	Lease allocations.
<code>dhcp_lease_deallocate_count</code>	Counter	Lease deallocations.
<code>dhcp_discover_count</code>	Counter	DISCOVER packets served from a pre-allocated lease.
<code>dhcp_discover_no_allocation_count</code>	Counter	DISCOVERs with no lease and no bound pool (no address to offer).
<code>dhcp_discover_pool_allocated_count</code>	Counter	DISCOVERs served by dishing out a fresh address from a bound pool (NSWO).
<code>dhcp_ack_count</code>	Counter	ACK packets sent.
<code>dhcp_nak_count</code>	Counter	NAK packets sent.
<code>dhcp_release_count</code>	Counter	RELEASE packets received.
<code>dhcp_decline_count</code>	Counter	DECLINE packets received.
<code>dhcp_inform_count</code>	Counter	INFORM packets received.
<code>dhcp_packet_malformed_count</code>	Counter	Malformed DHCP packets.

Example queries:

```
# Pool dish-out rate (NSW0 addressing)
rate(dhcp_discover_pool_allocated_count[5m])

# DISCOVERs that could not be served
increase(dhcp_discover_no_allocation_count[15m])
```

Note: the DHCP server also emits `dhcp_lease_reap` when it reaps an expired lease. This event is not declared as a Prometheus metric today and does not appear on the `/metrics` endpoint.

DHCPv6 and IPv6 RA Metrics

Metric	Type	Description
<code>dhcpv6_info_request_received_count</code>	Counter	DHCPv6 Information-Request packets received.
<code>dhcpv6_solicit_received_count</code>	Counter	DHCPv6 Solicit packets received.
<code>dhcpv6_request_received_count</code>	Counter	DHCPv6 Request packets received.
<code>dhcpv6_info_reply_sent_count</code>	Counter	DHCPv6 Information-Reply packets sent.
<code>dhcpv6_advertise_sent_count</code>	Counter	DHCPv6 Advertise packets sent.
<code>dhcpv6_reply_sent_count</code>	Counter	DHCPv6 Reply packets sent.
<code>ipv6_ra_prefix_add_count</code>	Counter	IPv6 RA prefix additions.
<code>ipv6_ra_prefix_remove_count</code>	Counter	IPv6 RA prefix removals.
<code>ipv6_ra_advertisement_sent_count</code>	Counter	IPv6 Router Advertisements sent.

WLCP Metrics

These counters track the WLAN Control Protocol used for multi-connection mode.

Metric	Type	Unit	Description
<code>wlcp_packet_received_count</code>	Counter	none	WLCP packets received.
<code>wlcp_packet_received_bytes</code>	Sum	bytes	Total bytes of WLCP packets received.
<code>wlcp_packet_malformed_count</code>	Counter	none	Malformed WLCP packets.
<code>wlcp_packet_unregistered_count</code>	Counter	none	WLCP packets from unregistered clients.
<code>wlcp_pdn_request_count</code>	Counter	none	PDN connection requests.
<code>wlcp_pdn_accept_count</code>	Counter	none	PDN connection accepts.
<code>wlcp_pdn_reject_count</code>	Counter	none	PDN connection rejects.
<code>wlcp_pdn_disconnect_request_count</code>	Counter	none	PDN disconnect requests.
<code>wlcp_pdn_disconnect_accept_count</code>	Counter	none	PDN disconnect accepts.

Erlang VM Metrics

These metrics give visibility into the BEAM runtime.

Memory

Metric	Description
<code>vm_memory_total</code>	Total allocated memory (bytes).
<code>vm_memory_processes</code>	Memory used by Erlang processes.
<code>vm_memory_processes_used</code>	Memory actively used by processes.
<code>vm_memory_system</code>	Memory used by the runtime.
<code>vm_memory_atom</code>	Memory used by atoms.
<code>vm_memory_atom_used</code>	Memory actively used by atoms.
<code>vm_memory_binary</code>	Memory used by binaries.
<code>vm_memory_code</code>	Memory used by loaded code.
<code>vm_memory_ets</code>	Memory used by ETS tables (session and AP state).

System

Metric	Description
<code>vm_system_info_process_count</code>	Current number of Erlang processes.
<code>vm_system_info_port_count</code>	Current number of ports.
<code>vm_system_info_atom_count</code>	Current number of atoms.
<code>vm_system_info_schedulers</code>	Total scheduler threads.
<code>vm_system_info_schedulers_online</code>	Online scheduler threads.

Schedulers

Metric	Description
<code>vm_statistics_run_queue</code>	Total run queue length.
<code>vm_total_run_queue_lengths_total</code>	Total scheduler run queue.
<code>vm_total_run_queue_lengths_cpu</code>	CPU scheduler run queue.
<code>vm_total_run_queue_lengths_io</code>	IO scheduler run queue.

Example queries:

```
# Total memory in MB
vm_memory_total / 1024 / 1024

# Run queue saturation (should be low)
avg_over_time(vm_statistics_run_queue[5m])
```

Example Grafana Queries

Authentication Overview

```
# Accepts per second
sum(rate(radius_access_accept_count[5m]))

# Rejects per second
sum(rate(radius_access_reject_count[5m]))

# Success ratio percentage
100 * sum(rate(radius_access_accept_count[5m])) /
  (sum(rate(radius_access_accept_count[5m])) +
   sum(rate(radius_access_reject_count[5m])))
```

Per-AP Statistics

```
# Requests by AP (Access-Request has no nas_ip label; use
accounting instead)
sum by (nas_ip) (rate(radius_accounting_request_count[5m]))

# Clients by AP
radius_clients_per_ap_count
```

Rejection and Rate-Limit Analysis

```
# Auth rejects by reason class
sum(increase(radius_access_reject_count[1h]))
sum(increase(radius_auth_unauthorized_source_count[1h]))
sum(increase(radius_auth_bad_secret_count[1h]))

# Teardown reasons
sum by (reason) (increase(twag_session_teardown_count[1h]))
```

Data Usage

```
# Total throughput in MB  
(radius_data_bytes_in + radius_data_bytes_out) / 1024 / 1024
```

Alerting Rules

Critical Alerts

```
groups:
  - name: twag_critical
    rules:
      - alert: HighAuthenticationFailureRate
        expr: >
          sum(rate(radius_access_reject_count[5m])) /
          (sum(rate(radius_access_accept_count[5m])) +
          sum(rate(radius_access_reject_count[5m]))) > 0.1
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "High authentication failure rate (> 10%)"

      - alert: EAPSyncFailures
        expr: increase(eap_aka_sync_failure_count[5m]) > 0
        for: 1m
        labels:
          severity: warning
        annotations:
          summary: "EAP-AKA synchronization failures detected"

      - alert: UnknownRadiusSources
        expr: increase(radius_auth_unauthorized_source_count[15m])
        > 0
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "RADIUS packets from sources with no matching
          credential"

      - alert: BadRadiusSecret
        expr: increase(radius_auth_bad_secret_count[15m]) > 0
        for: 5m
        labels:
          severity: warning
        annotations:
```

```
summary: "RADIUS packets rejected for a wrong shared secret"
```

- alert: NoActiveClients
expr: radius_active_clients_count == 0
for: 15m
labels:
severity: warning
annotations:
summary: "No active clients for 15 minutes"

Resource Alerts

```
groups:  
- name: twag_resources  
  rules:  
    - alert: HighMemoryUsage  
      expr: vm_memory_total > 1073741824  
      for: 10m  
      labels:  
        severity: warning  
      annotations:  
        summary: "TWAG memory usage exceeds 1GB"  
  
    - alert: HighRunQueue  
      expr: avg_over_time(vm_statistics_run_queue[5m]) > 10  
      for: 5m  
      labels:  
        severity: warning  
      annotations:  
        summary: "High scheduler run queue indicates CPU saturation"
```

Related Pages

- [Architecture](#)
- [RADIUS Rejections](#)
- [RADIUS Shared Secrets](#)

- Sessions
- Online Charging
- Per-AP Accounting

Overview

Online Charging (Gy)

Overview

OmniTWAG can charge data sessions in real time against an Online Charging System (OCS). OmniTWAG uses the Diameter Ro (Gy) interface to send Credit-Control Requests (CCR) and to receive Credit-Control Answers (CCA). This lets an operator apply prepaid credit control to WLAN traffic that the TWAG serves.

Online charging applies to sessions where the TWAG owns the data path, that is **NSWO (local breakout)** and TWAG-anchored traffic. In this case the TWAG is the point where traffic exits to the internet, so the TWAG acts as the charging enforcement point. In **home-routed (GTP) mode** the PGW performs charging, so an operator normally leaves the global charging mode `:disabled`. See the [User Plane](#) reference for the data-plane modes.

Two controls decide whether Gy runs for a given UE:

- The **global charging mode** (`online_charging.mode`). This picks which Credit-Control Requests OmniTWAG can send at all.
- The **per-AP charging mode** on the AP credential (`charging_mode`). This gates, per access point, whether OmniTWAG opens a Gy session for that UE. See [RADIUS Credentials](#).

Both controls must permit charging for a CCR-Initial to go out. See [Per-AP Charging Mode](#).

Table of Contents

- [Concepts](#)
- [Charging Modes](#)
- [Per-AP Charging Mode](#)
- [Credit-Control Flow](#)
- [Quota, Usage, and Reauthorization](#)
- [Quota Enforcement](#)

- Rating Group and Service Units
- Configuration
- Operational Behaviour and Failure Handling
- Observability
- Reference Tables

Concepts

The Gy interface is a Diameter application that carries credit-control messages between a network element and an OCS. OmniTWAG uses the 3GPP Ro dictionary with Auth-Application-Id 4, as defined in RFC 4006 and 3GPP TS 32.299. OmniTWAG uses the Service-Context-Id 32251@3gpp.org, which is the 3GPP PS charging context.

OmniTWAG sends three types of Credit-Control Request during a session:

- **CCR-Initial (CCR-I)**: at session start. OmniTWAG asks the OCS to start credit control for the subscriber.
- **CCR-Update (CCR-U)**: during the session. OmniTWAG reports used data and asks for more quota.
- **CCR-Termination (CCR-T)**: at session end. OmniTWAG reports the final used data and closes the credit-control session.

The OCS replies to each request with a Credit-Control Answer (CCA). A CCA carries a Result-Code, an optional grant of quota (Granted-Service-Unit), and an optional Validity-Time.

OmniTWAG keeps a separate Gy state for each subscriber session. The Gy state holds the Diameter Session-Id, the current CC-Request-Number, the cumulative usage counters, and the last quota that the OCS granted. OmniTWAG creates this state when it sends the CCR-I, and it discards the state when the session ends.

Charging Modes

The `online_charging.mode` parameter is the **global** charging mode. It selects one of three modes and controls which Credit-Control Requests OmniTWAG can send. The per- AP `charging_mode` (see [Per-AP Charging Mode](#)) then decides, for each UE, whether OmniTWAG opens a Gy session at all.

Mode	CCR-I at start	CCR-U during session	CCR-T at end
<code>:disabled</code>	No	No	No
<code>:authorize_only</code>	Yes	No	Yes
<code>:online_charging</code>	Yes	Yes	Yes

`:disabled`

OmniTWAG does not contact the OCS. This is the default. Use this mode for Home-Routed (GTP) operation, where the PGW performs charging.

`:authorize_only`

OmniTWAG sends a CCR-I at session start and a CCR-T at session end. OmniTWAG does not send interim CCR-U messages. Use this mode to check the subscriber balance at start and to report total usage at the end, without live quota management.

`:online_charging`

OmniTWAG sends a CCR-I at start, a CCR-U on each accounting interim-update, and a CCR-T at end. Use this mode for full quota management, where the OCS grants and refreshes quota during the session.

Per-AP Charging Mode

Each AP credential carries a `charging_mode` field. This is separate from the global `online_charging.mode`. OmniTWAG resolves the credential from the RADIUS source of the session, so every access point can have its own behaviour. See [RADIUS Credentials](#).

Per-AP <code>charging_mode</code>	Meaning	Effect on Gy
<code>:charged</code>	Charge this AP's traffic. This is the default.	OmniTWAG opens a Gy session (CCR-Initial) for the UE when the global mode also permits it.
<code>:uncharged</code>	Serve this AP but do not charge it.	OmniTWAG establishes the data session but opens no Gy session. No CCR is sent.
<code>:auth_only</code>	Authenticate only, no data session.	OmniTWAG authenticates the subscriber and stops. It builds no data session and opens no Gy session.

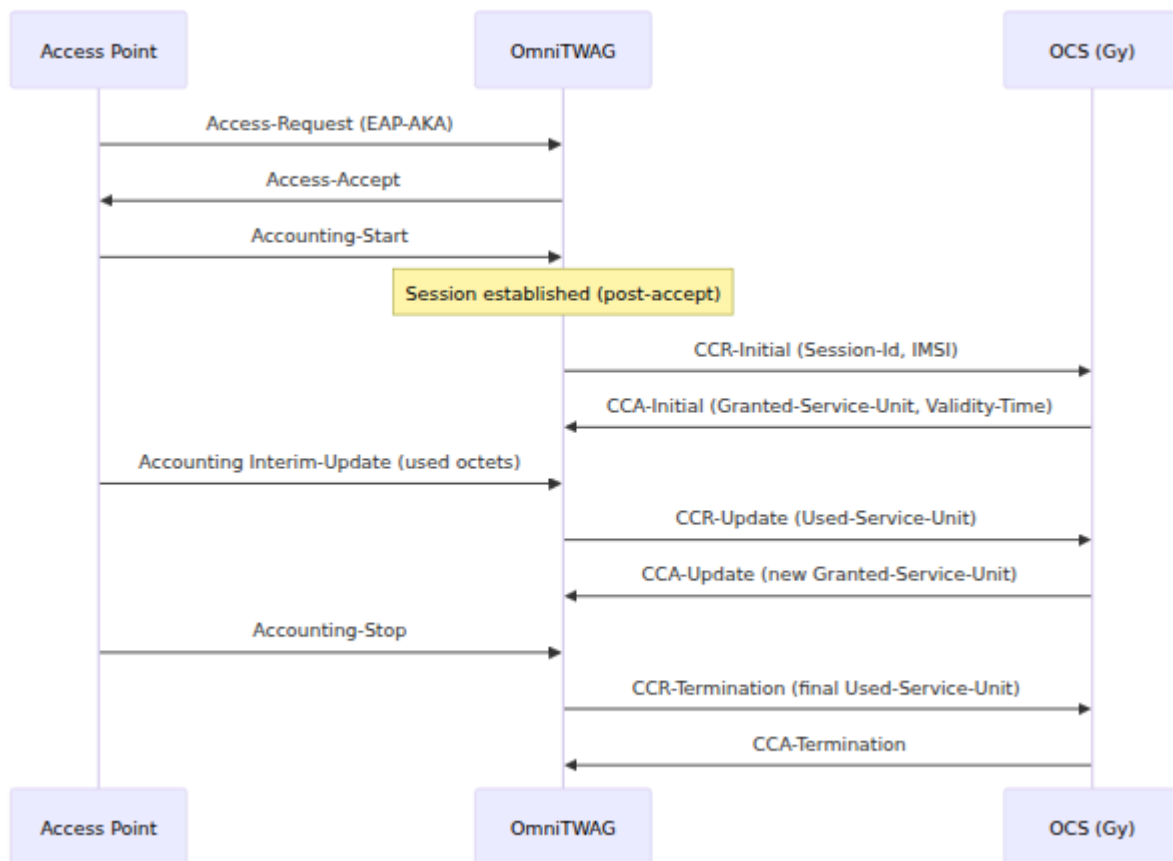
OmniTWAG sends a CCR-Initial only when **all** of these hold:

- The global `online_charging.mode` is not `:disabled` (the Gy client is enabled).
- The session has an IMSI.
- The AP credential `charging_mode` is `:charged`.

If any condition fails, OmniTWAG opens no Gy session for the UE, and no later CCR-U or CCR-T is sent. The per-AP mode therefore gates Gy independently of the global mode: even with the global mode set to `:online_charging`, an AP marked `:uncharged` produces no Gy traffic.

Credit-Control Flow

The credit-control session follows the subscriber session. OmniTWAG triggers each Credit-Control Request from a specific point in the RADIUS session lifecycle.



Session start (CCR-Initial)

OmniTWAG sends the CCR-I after it establishes the session. The session establishment runs on the post-accept step, which the RADIUS server triggers after the Access-Accept and the Accounting-Start. OmniTWAG sends the CCR-I only when the global charging mode is enabled, the session has an IMSI, and the AP credential `charging_mode` is `:charged`. See [Per-AP Charging Mode](#).

The CCR-I carries the subscriber IMSI in a `Subscription-ID` AVP with type `1` (`END_USER_IMSI`). The CCR-I sets `Multiple-Services-Indicator` to `1` and includes one `Multiple-Services-Credit-Control` (MSCC) group with an empty `Requested-Service-Unit` and the configured `Rating-Group`.

Interim updates (CCR-Update)

In `:online_charging` mode, each RADIUS accounting interim-update folds the reported octets and session time into the Gy usage counters. OmniTWAG does not send a CCR-U on every interim. It sends a CCR-U only when one of these triggers fires:

- The used volume has crossed the configured fraction of the granted volume (the `quota.update_threshold`, default `0.8`).
- The granted quota's `Validity-Time` has expired.

An interim that does not cross a trigger just records usage locally, so the OCS is not consulted on every interim. When a CCR-U does go out, OmniTWAG reads the used octets and the session time from the accounting attributes and reports them in a `Used-Service-Unit` AVP. The CCR-U also includes an empty `Requested-Service-Unit`, so the OCS can grant more quota.

A CCR-U is also sent off the accounting path when the quota-validity timer fires. See [Quota Enforcement](#).

OmniTWAG reads these RADIUS accounting attributes for the CCR-U:

RADIUS attribute	Gy field
<code>Acct-Input-Octets</code>	<code>CC-Input-Octets</code>
<code>Acct-Output-Octets</code>	<code>CC-Output-Octets</code>
<code>Acct-Session-Time</code>	<code>CC-Time</code>

OmniTWAG computes `CC-Total-Octets` as the sum of the input and output octets.

Session end (CCR-Termination)

OmniTWAG sends a CCR-T when the session tears down, for example on a RADIUS Accounting-Stop. OmniTWAG reports the final cumulative usage in a `Used-Service-Unit` AVP and sets the `Termination-Cause` to `1`

(DIAMETER_LOGOUT). OmniTWAG sends the CCR-T in both `:authorize_only` mode and `:online_charging` mode.

Quota, Usage, and Reauthorization

Granted quota

Each CCA can carry a `Granted-Service-Unit` inside the first MSCC group. OmniTWAG stores the granted values in the Gy state:

- `CC-Total-Octets`, `CC-Input-Octets`, `CC-Output-Octets`: the granted data volume.
- `CC-Time`: the granted time.
- `Validity-Time`: the time in seconds that the grant stays valid. OmniTWAG turns this into an absolute deadline and arms a timer against it.

OmniTWAG also reads the MSCC `Final-Unit-Indication` and the MSCC-level `Result-Code` from each CCA. A `Final-Unit-Action` of TERMINATE, REDIRECT, or RESTRICT_ACCESS, or an MSCC `Result-Code` of `4010` (END_USER_SERVICE_DENIED) or `4012` (CREDIT_LIMIT_REACHED), is stored as a credit-exhaustion signal. See [Quota Enforcement](#).

OmniTWAG updates the stored grant on every successful CCA-I and CCA-U.

Usage reporting

OmniTWAG keeps cumulative usage counters in the Gy state. The RADIUS accounting counters are cumulative for the session, so each interim-update reports the total so far. OmniTWAG records the reported counters and sends them as the `Used-Service-Unit`. The CCR-U sets the `Reporting-Reason` to `4` (VALIDITY_TIME). The CCR-T sets the `Reporting-Reason` to `2` (FINAL).

Reauthorization

OmniTWAG re-authorizes on two paths:

- **On traffic.** A RADIUS accounting interim-update that crosses the quota threshold or finds the validity expired triggers a CCR-U. The interim interval is a property of the access point, so a shorter interim interval gives finer metering.
- **On time.** OmniTWAG arms a quota-validity timer from the granted `Validity-Time`. If the timer fires with little or no traffic, OmniTWAG still sends a CCR-U to re-authorize. If the OCS sends no `Validity-Time`, OmniTWAG can fall back to the configured `quota.validity_time`. With neither present, no validity timer is armed.

The `CC-Request-Number` increases by one after each successful CCA. This value gives every request in the credit-control session a unique sequence number, as required by [RFC 4006](#).

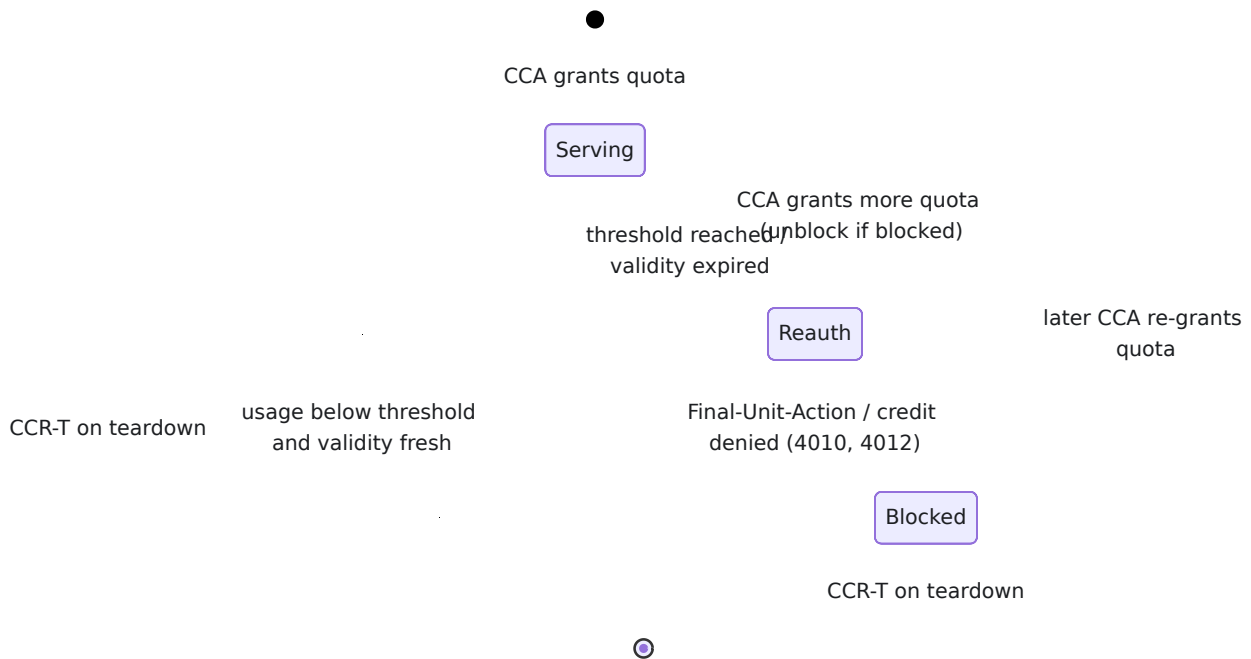
Quota Enforcement

OmniTWAG enforces credit exhaustion in `:online_charging` mode. The TWAG has no separate user-plane function, so its enforcement lever is the GRE data path: it unregisters the UE from the data path, which drops uplink and downlink for that UE.

A CCR-U response triggers enforcement when the Gy state shows credit exhausted, that is when the OCS:

- signals any `Final-Unit-Action` (TERMINATE, REDIRECT, or RESTRICT_ACCESS), or
- returns an MSCC `Result-Code` of `4010` or `4012`.

Because the TWAG cannot honour REDIRECT or RESTRICT_ACCESS on its data path, any `Final-Unit-Action` is treated as a stop. When enforcement fires, OmniTWAG blocks the UE's data path but keeps the session, the IP lease, and the accounting alive. A later re-authorization that grants usable quota restores the data path.



Enforcement is best-effort and fails open on transport errors: a CCR-U that fails at the Diameter layer never blocks the UE. Blocking happens only on an explicit OCS credit denial or exhaustion signal. See [Operational Behaviour and Failure Handling](#).

Rating Group and Service Units

OmniTWAG uses a single rating group for the whole session. The `rating_group` parameter sets the `Rating-Group` value in the MSCC of the CCR-I. The default is `1`. OmniTWAG does not split traffic into multiple rating groups or service identifiers. OmniTWAG requests a combined volume grant with an empty `Requested-Service-Unit`, so the OCS decides the grant.

Configuration

Set the online charging parameters under the `:omnitwag` application, in the `online_charging` block of `config/runtime.exs`.

```
config :omnitwag,
  online_charging: %{
    # Global charging mode: :disabled, :authorize_only, or
:online_charging
    mode: :disabled,

    # OCS Diameter realm (Destination-Realm for CCR)
    ocs_realm: "epc.mnc001.mcc001.3gppnetwork.org",

    # Rating group for the Multiple-Services-Credit-Control group
    rating_group: 1,

    # Quota management (used in :online_charging mode)
    quota: %{
      # Fraction of the granted volume at which a CCR-Update is
triggered
      update_threshold: 0.8,

      # Fallback validity time (seconds) when the OCS sends no
Validity-Time
      validity_time: 3600
    }
  }
}
```

Parameters

Parameter	Type	Required	Default	Description
<code>mode</code>	Atom	No	<code>:disabled</code>	Global charging mode. One of <code>:disabled</code> , <code>:authorize</code> , or <code>:online_charging</code> . See Charging Modes .
<code>ocs_realm</code>	String	No	Diameter realm	Diameter Realm for the OCS. OmniTWAG uses this as the <code>Destination Realm</code> in each request. If not set, OmniTWAG uses the local Diameter realm.
<code>rating_group</code>	Integer	No	<code>1</code>	Rating-Group value that OmniTWAG uses for the MSCC of CCR-Initial.
<code>quota.update_threshold</code>	Float	No	<code>0.8</code>	Fraction (0.0 to 1.0) of the guaranteed volume at which an interim-update triggers a CCR-Update.

Parameter	Type	Required	Default	Description
<code>quota.validity_time</code>	Integer	No	(unset)	Fallback quota validity in seconds used to arm validity timer the OCS sends <code>Validity-Timer</code> unset, no fallback timer is armed

The per-AP `charging_mode` is not part of this block. It lives on each AP credential. See [RADIUS Credentials](#).

The Diameter `Origin-Host` and `Origin-Realm` for the Gy session come from the Diameter identity in the `:diameter_ex` configuration (`host` and `realm`). The `Destination-Host` is not configured. OmniTWAG learns the `Destination-Host` from the `Origin-Host` of the first CCA, and it adds this AVP to later CCR-U and CCR-T messages.

Diameter transport and the Ro application

The Gy client sends through the shared Diameter stack. Configure the OCS as a Diameter peer, and enable the `:ro` application, in the `:diameter_ex` configuration.

```
config :diameter_ex,
  diameter: %{
    service_name: :omnitouch_twag,
    host: "amanaki",
    realm: "epc.mnc001.mcc001.3gppnetwork.org",
    request_timeout: 5000,
    applications: [
      %{
        application_name: :ro,
        application_dictionary: :diameter_gen_3gpp_ro,
        vendor_specific_application_ids: [
          %{vendor_id: 0, auth_application_id: 4,
acct_application_id: nil}
        ]
      }
    ],
    peers: [
      %{
        host: "ocs.epc.mnc001.mcc001.3gppnetwork.org",
        realm: "epc.mnc001.mcc001.3gppnetwork.org",
        ip: "10.179.2.233",
        port: 3868,
        transport: :diameter_tcp,
        initiate_connection: true
      }
    ]
  }
}
```

Parameter	Type	Required	Default	
<code>service_name</code>	Atom	Yes	-	Nam Dian Omn requ serv
<code>host</code>	String	Yes	-	Loca (hos builc Host <code>real</code>
<code>realm</code>	String	Yes	-	Loca Omn the (as th <code>Real</code> is no
<code>request_timeout</code>	Integer	No	5000	Time wait the r
<code>application_name</code>	Atom	Yes	-	Nam appl for G
<code>application_dictionary</code>	Atom	Yes	-	Dian the a <code>:dia</code> for G
<code>vendor_specific_application_ids</code>	List	Yes	-	Venc appl use (

Parameter	Type	Required	Default	
				auth 4.
peers	List	Yes	-	List o Add here Para

Peer Parameters

Parameter	Type	Required	Default	Description
<code>host</code>	String	Yes	-	Peer Diameter Identity (FQDN). Must match the identity.
<code>realm</code>	String	Yes	-	Peer Diameter Realm. See the OCS realm.
<code>ip</code>	String	Yes	-	OCS IP address for the TCP connection.
<code>port</code>	Integer	No	3868	OCS Diameter port. The standard port is 3868.
<code>transport</code>	Atom	No	<code>:diameter_tcp</code>	Transport protocol. Use <code>:diameter</code> for TCP.
<code>initiate_connection</code>	Boolean	No	false	When true, OmniTWA starts the connection to the peer.

Note: The `online_charging` block in the shipped `config/runtime.exs` also shows `ocs_host`, `service_identifier`, a top-level `request_timeout`, and the `quota` keys `initial_request` and `subsequent_request`. The current charging code does not read these keys. It reads `mode`, `ocs_realm`, `rating_group`, `quota.update_threshold`, and `quota.validity_time`. The

Diameter request timeout comes from the `:diameter_ex`
`request_timeout`. OmniTWAG requests quota with an empty `Requested-`
`Service-Unit`, so `initial_request` and `subsequent_request` do not shape
the CCR today. Do not depend on the unread keys until the code supports
them.

Operational Behaviour and Failure Handling

Fail-open behaviour

OmniTWAG treats a Gy transport failure as non-fatal. If a Credit-Control Request fails at the Diameter layer, the subscriber session continues and OmniTWAG does not block traffic.

- **OCS unreachable or timeout:** The CCR fails at the Diameter transport. OmniTWAG logs a warning. For a CCR-I, OmniTWAG still keeps the Gy state, so it can send a CCR-T later. For a CCR-U, OmniTWAG keeps the previous Gy state and does not block the UE. For a CCR-T, OmniTWAG logs the warning and completes the teardown.
- **CCA error result-code:** The OCS returns a command-level `Result-Code` that is not `2001` (`DIAMETER_SUCCESS`). OmniTWAG logs the error and treats the request as failed. OmniTWAG keeps the session alive.

Enforcement is separate from transport failure. In `:online_charging` mode, OmniTWAG does block the UE's data path when the OCS explicitly denies credit or signals a Final-Unit-Action in a CCA MSCC. See [Quota Enforcement](#). It never blocks on a transport failure, and it never denies access in `:authorize_only` mode. Plan the OCS integration with this split in mind.

Credit-control session lifecycle summary

```
Parse error on line 2: ...arging: global mode :disabled OR AP char... -----  
-----^ Expecting 'SPACE', 'NL', 'HIDE_EMPTY', 'scale', 'COMPOSIT_STATE',  
'STRUCT_STOP', 'STATE_DESCR', 'ID', 'FORK', 'JOIN', 'CHOICE', 'CONCURRENT',  
'note', 'acc_title', 'acc_descr', 'acc_descr_multiline_value', 'CLICK', 'classDef',
```

'style', 'class', 'direction_tb', 'direction_bt', 'direction_rl', 'direction_lr',
'EDGE_STATE', got 'DESCR'

Try again

Result-code handling

OmniTWAG reads the result code at two levels.

- **Command-level Result-Code**. OmniTWAG treats 2001 as success. It also treats a missing Result-Code as success, because some peer responses arrive as a raw record without a decoded result code. Any other command-level value is an error: OmniTWAG logs it, does not retry, and keeps the session alive.
- **MSCC-level Result-Code**. Inside the first MSCC group, a Result-Code of 4010 (END_USER_SERVICE_DENIED) or 4012 (CREDIT_LIMIT_REACHED) is read as a credit-denial signal. In :online_charging mode this drives enforcement and blocks the UE's data path. See [Quota Enforcement](#).

Observability

OmniTWAG emits Telemetry events for every Credit-Control Request and answer. The events carry a type metadata value of :initial, :update, or :terminate.

Telemetry event	When	Metadata
<code>[:twag, :gy, :ccr, :sent]</code>	OmniTWAG sends a CCR	<code>type</code>
<code>[:twag, :gy, :cca, :received]</code>	OmniTWAG receives a CCA	<code>type, result (:success or :error)</code>
<code>[:twag, :gy, :ccr, :error]</code>	A CCR fails at the transport	<code>type</code>
<code>[:twag, :gy, :quota, :blocked]</code>	The UE data path was blocked on credit exhaustion	<code>identity</code>
<code>[:twag, :gy, :quota, :unblocked]</code>	The UE data path was restored after a re-grant	<code>identity</code>

Note: These Gy Telemetry events are not registered as Prometheus metrics in the current build. The [Metrics Reference](#) does not yet export them. To watch Gy behaviour today, use the application logs. The Gy client logs each CCR-I, CCR-U, and CCR-T at INFO level, and each failure at WARN or ERROR level. The log lines carry the Diameter `Session-Id` and, for the CCR-I, the IMSI.

Reference Tables

CC-Request-Type values

Value	Name	Message
1	INITIAL_REQUEST	CCR-Initial
2	UPDATE_REQUEST	CCR-Update
3	TERMINATION_REQUEST	CCR-Termination

Values are defined in [RFC 4006 §8.3](#).

Reporting-Reason values used

Value	Name	Where OmniTWAG uses it
2	FINAL	CCR-Termination
4	VALIDITY_TIME	CCR-Update

Result-Code

Code	Level	Name	OmniTWAG behaviour
2001	Command	DIAMETER_SUCCESS	Treated as success. The Gy state advances.
(absent)	Command	-	Treated as success, for raw record responses.
Any other	Command	(error)	Logged as an error. The session continues.
4010	MSCC	END_USER_SERVICE_DENIED	Read as credit denied. Enforces (blocks the UE) in :online_charging mode.
4012	MSCC	CREDIT_LIMIT_REACHED	Read as credit denied. Enforces (blocks the UE) in :online_charging mode.

Final-Unit-Action values

The MSCC `Final-Unit-Indication` carries a `Final-Unit-Action`. OmniTWAG has no user-plane function, so it cannot honour REDIRECT or RESTRICT_ACCESS on the data path. It treats any Final-Unit-Action as a stop and blocks the UE.

Value	Name	OmniTWAG behaviour
0	TERMINATE	Blocks the UE data path (session retained).
1	REDIRECT	Blocks the UE data path (redirect not supported).
2	RESTRICT_ACCESS	Blocks the UE data path (restrict not supported).

Final-Unit-Indication is defined in [3GPP TS 32.299](#) and [RFC 4006 §5.6](#).

Diameter application

ID	Interface	Dictionary	Direction	Reference
4	Ro / Gy	diameter_gen_3gpp_ro	TWAG to OCS	RFC 4006 , 3GPP TS 32.299

Specification references

Topic	Specification
Diameter Credit-Control Application	RFC 4006
Diameter charging applications (Ro)	3GPP TS 32.299
Charging architecture	3GPP TS 32.240
TWAG / non-3GPP access (S2a, NSW0)	3GPP TS 23.402
Diameter Base Protocol	RFC 6733

Related pages

- [Overview](#)
- [RADIUS Credentials](#)
- [IP Pools](#)
- [IP Address Management](#)
- [Sessions](#)
- [S2a GTP-C](#)
- [User Plane](#)

[Overview](#)

Per-AP Accounting

Overview

OmniTWAG aggregates RADIUS accounting data for each access point. The operator sees the total traffic and the session counts for each access point. This data is separate from the per-session accounting records. OmniTWAG keys the per-AP totals by the `NAS-IP-Address` of the access point. When an access point omits `NAS-IP-Address` (for example some UBNT models), OmniTWAG falls back to the UDP source IP of the RADIUS packet, so accounting attaches to the same access-point entry that authentication created.

Table of Contents

- [Concepts](#)
- [How totals are counted](#)
- [Recorded fields](#)
- [Active client count](#)
- [Where to see the data](#)
- [Relation to per-session data](#)
- [Troubleshooting](#)
- [Related Pages](#)

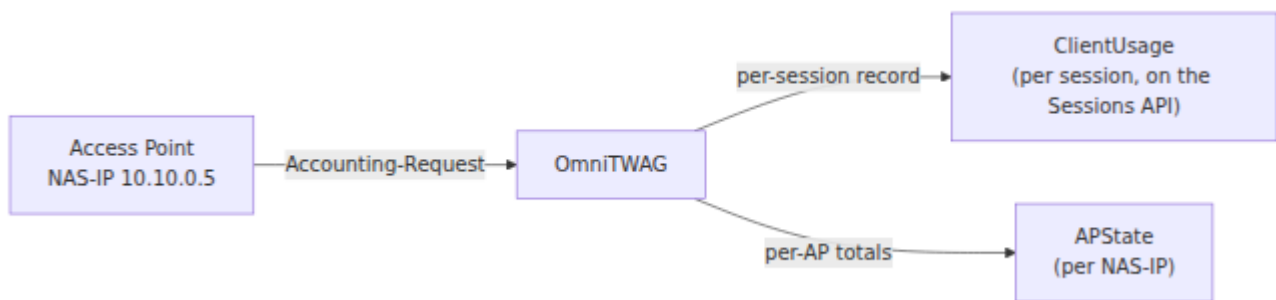
Concepts

An access point sends RADIUS Accounting-Request packets for each client session. A packet has an `Acct-Status-Type` that states the event:

- **Start** (1): a session started.
- **Interim-Update** (3): a session is active, with the counters so far.
- **Stop** (2): a session stopped, with the final counters.

OmniTWAG reads the octet and packet counters and the `NAS-IP-Address` from each packet. OmniTWAG adds these values to the running totals for that access

point. This gives the operator a per-AP view of traffic and sessions.



How totals are counted

The RADIUS octet and packet counters are **cumulative for each session**. An Interim-Update and a Stop both report the total for the session so far, not the change since the last packet. If OmniTWAG added the counters from every packet, it would count the same traffic more than once.

To avoid double counting, OmniTWAG adds the octet and packet counters to the per-AP totals **only on a Stop**. A Stop reports the final total for a completed session. The rules are:

- On **Start**: increase `acct_sessions_started` by one. Increase `acct_active_sessions` by one.
- On **Stop**: increase `acct_sessions_stopped` by one. Decrease `acct_active_sessions` by one, but not below zero. Add the final octet and packet counters to the per-AP totals.
- On **Interim-Update**: update the last-accounting time only. Do not change the totals.

If OmniTWAG receives a Stop for a session that it did not see start, the active session count does not go below zero. OmniTWAG still adds the final octets and packets. If OmniTWAG receives an Accounting-Request for an access point that it did not see before, OmniTWAG creates the per-AP entry. This means an accounting-only access point is still tracked.



Start (sessions_started
+1, active +1)

Active

Stop (sessions_stopped
+1, active -1, add
octets/packets)

Stopped

Interim-Update (time
only)



Recorded fields

OmniTWAG records these fields for each access point, in addition to the access-point metadata such as SSID and Called-Station-Id.

Field	Type	Description
<code>acct_input_octets</code>	Integer	Total input octets, summed over stopped sessions.
<code>acct_output_octets</code>	Integer	Total output octets, summed over stopped sessions.
<code>acct_input_packets</code>	Integer	Total input packets, summed over stopped sessions.
<code>acct_output_packets</code>	Integer	Total output packets, summed over stopped sessions.
<code>acct_sessions_started</code>	Integer	Number of sessions that started on this access point.
<code>acct_sessions_stopped</code>	Integer	Number of sessions that stopped on this access point.
<code>acct_active_sessions</code>	Integer	Accounting-derived open-session gauge: <code>started</code> minus <code>stopped</code> , never less than zero. It reflects only what RADIUS accounting reports, so it can over-count when an access point misses Stop packets (for example after a reboot) and stays stale until Stops arrive or the entry expires. This is not the "Active Clients" figure shown on the Access Points page — see Active client count .
<code>last_accounting_at</code>	Integer	Time of the last accounting packet, in milliseconds since the Unix epoch.

The direction of the octet and packet counters follows the RADIUS definition in [RFC 2866](#). Input is traffic from the client into the network. Output is traffic from

the network to the client.

Active client count

The **Active Clients** figure on the Access Points page, and the `client_count` field on the access-point API, report how many clients are **connected right now**. This value is not the accounting counter `acct_active_sessions`. It is derived from the live in-memory session processes that OmniTWAG runs for each authenticated client, grouped by the access point's `NAS-IP-Address`.

Each session process monitors liveness from the accounting interim-updates. When a client stops sending updates for a configured number of intervals, the process tears itself down, whether or not a RADIUS Accounting-Stop ever arrives. Because the active client count is a tally of these live processes, it drains to zero on its own when an access point drops off — it does not depend on the access point sending a Stop, and it does not drift upward after an access-point reboot.

Use `client_count` (the live figure) as the count of currently connected clients. Use `acct_active_sessions` when you specifically want the accounting-reported open-session gauge, keeping its caveats above in mind.

Where to see the data

The per-AP fields are part of the access-point record. The operator reads them in two places:

- **REST API:** the `GET /access_points` endpoint returns every access point with its per-AP accounting fields. The `GET /access_points/{id}` endpoint returns one access point by its `NAS-IP-Address`.
- **Web interface:** the **Access Points** page shows every access point. Expand an access point to see its full data, including the per-AP accounting fields.

Relation to per-session data

The per-AP totals are an aggregate. For each RADIUS Accounting-Request, OmniTWAG writes two records:

- The **per-AP totals** described here, held in `APState` and keyed by NAS-IP.
- A **per-session record**, held in `ClientUsage` and keyed by `Acct-Session-Id`. This is the live per-subscriber traffic and accounting view, and it is exposed on the Sessions API, not on the access-point record. See [Sessions](#).

The per-AP totals count octets and packets only on a Stop, so they lag the live per-session view during an active session. Use the Sessions API when you need the current usage of a single subscriber, and the per-AP totals when you need the lifetime traffic and session counts for an access point.

The access-point entry is created either by authentication (the AP is registered from the Access-Request, see [RADIUS Shared Secrets](#)) or, for accounting-only access points, by the first Accounting-Request. An access point never appears if its packets are dropped because the source IP matched no credential subnet.

Troubleshooting

The totals do not change

Symptoms: The octet and packet totals for an access point stay at zero, even though clients use the network.

Possible causes:

- The access point does not send Stop packets. OmniTWAG adds octets and packets only on a Stop.
- The access point does not send accounting at all.

Resolution:

1. Confirm that the access point sends Accounting-Request packets to OmniTWAG.

2. Confirm that the access point sends a Stop at the end of a session.
3. Confirm that the accounting shared secret on the access point matches the credential on file. See [RADIUS Shared Secrets](#).

The active session count looks too high

Symptoms: The `acct_active_sessions` value is higher than the real number of active clients.

Possible causes:

- The access point sent Start packets but did not send the matching Stop packets, for example after an access-point reboot.

Resolution:

1. Confirm that the access point sends a Stop for each session.
2. Set an interim-update interval on the access point, so OmniTWAG sees active sessions.
3. The active count corrects itself as new Stop packets arrive. The count does not go below zero.

This affects only `acct_active_sessions`. The **Active Clients** figure on the Access Points page (and the `client_count` API field) is not affected, because it is a tally of live session processes rather than the accounting counter. See [Active client count](#).

An access point does not appear

Symptoms: An access point that sends accounting does not appear in the list.

Possible causes:

- OmniTWAG dropped the packets because the source IP matched no credential subnet.

Resolution:

1. Check the `radius_auth_unauthorized_source_count` metric. See [Metrics Reference](#) and [RADIUS Rejections](#).
2. Add a credential for the subnet of the access point. See [RADIUS Shared Secrets](#).

Related Pages

- [Overview](#)
- [Sessions](#)
- [RADIUS Shared Secrets](#)
- [RADIUS Rejections](#)
- [Metrics Reference](#)
- [Architecture](#)

[Overview](#)

OmniTWAG

Performance

Benchmarks

Control-plane authentication and signalling throughput of the Trusted WLAN Access Gateway.

Test environment

CPU	Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz (2 cores)
Runtime	Elixir 1.17.0 / Erlang OTP 27 (JIT) (erts 15.0.1)
Measurement	1 s warmup + 3 s measure + 1 s memory per op
Commit	<code>main</code> @ <code>01ec4e8</code>

Throughput is reported as **ips** (iterations per second: higher is faster). **Avg** and **Median** are per-call latency in microseconds; the median is the better estimate of typical steady-state latency. **Mem/op** is the heap memory allocated per call.

Full authentication (end-to-end)

The topline figure: each scenario chains the complete per-attach sequence (EAP Identity decode, authentication-vector handling, the EAP-AKA' key hierarchy, challenge build, MAC, response decode, and MAC verification), so its throughput (**ips**) is **complete attaches, i.e. sessions, per second per core**. *Local mode* computes the authentication vector on-box (Milenage); *STa mode* receives it from the HSS over SWx/STa.

Scenario	Throughput (ips)	Avg (μs)	Median (μs)	Mem/op
Full EAP-AKA' attach, STa mode (vector from HSS)	17,311	57.77	52.90	7,048 B
Full EAP-AKA' attach, local mode (vector computed on-box)	12,493	80.05	73.31	10,560 B

Cryptography

Per-authentication cryptographic operations: SIM authentication-vector generation (Milenage), the EAP-AKA' key hierarchy (CK'/IK', session keys, MAC), ERP fast re-authentication keys, and the generic 3GPP key-derivation function.

Operation	Throughput (ips)	Avg (μs)	Median (μs)	Mem/op
WLCP key derivation (EMSK -> WLCP key)	262,975	3.80	3.54	72 B
EAP-AKA' build Challenge packet (encode)	257,991	3.88	0.87	456 B
EAP-AKA' derive CK'/IK'	225,191	4.44	3.93	328 B
EAP-AKA' compute MAC (HMAC-SHA-256/16)	223,503	4.47	4.11	72 B
3GPP generic KDF (HMAC-SHA-256)	198,596	5.04	4.32	248 B
ERP derive rMSK (per re-auth)	113,269	8.83	8.01	376 B
ERP full fast re-auth (rRK -> rIK -> rMSK)	61,496	16.26	14.97	520 B
Milenage compute_all (RES, CK, IK, AK, MAC- A)	52,451	19.07	16.11	3,560 B
EAP-AKA' derive_keys (MK -> all session keys)	31,696	31.55	28.93	1,552 B
EAP-AKA' full auth crypto (CK'/IK' + all keys)	28,380	35.24	32.59	1,808 B

Operation	What it does
EAP-AKA' build Challenge packet	Encodes the EAP-Request/AKA'-Challenge (AT_RANDOM, AT_AUTN, AT_KDF, AT_MAC).
WLCP key derivation	Derives the WLCP protection key from the EMSK (multi-connection mode).
3GPP generic KDF	HMAC-SHA-256 key-derivation function used across the key hierarchy.
EAP-AKA' compute MAC	HMAC-SHA-256 AT_MAC over an EAP packet, truncated to 16 bytes.
EAP-AKA' derive CK'/IK'	Transforms CK/IK into CK'/IK' bound to the access-network name.
ERP derive rMSK	Per-re-auth session key for EAP Re-authentication (RFC 6696).
ERP full fast re-auth	Full rRK → rIK → rMSK derivation chain for a fast re-auth.
Milenage compute_all	One AKA authentication vector: RES, CK, IK, AK, MAC-A.
EAP-AKA' derive_keys	Expands the master key into K_encr, K_aut, K_re, MSK, EMSK.
EAP-AKA' full auth crypto	CK'/IK' derivation plus the full session-key expansion.

Packet codecs

Signalling-plane encode/decode operations: EAP packet parsing, WLCP (TS 24.244) message handling for multi-connection mode, and the MPPE key

attribute built on every RADIUS Access-Accept.

Operation	Throughput (ips)	Avg (µs)	Median (µs)	Mem/op
WLCP decode (PDN Connection Request)	4,229,558	0.24	0.20	448 B
WLCP encode (PDN Connection Request)	2,492,441	0.40	0.24	160 B
WLCP encode + decode round-trip	1,989,407	0.50	0.40	608 B
EAP decode Identity packet	336,816	2.97	2.66	1,136 B
MPPE build VSA (Send-Key, RC4 + MD5)	135,026	7.41	5.32	1,296 B

Operation	What it does
WLCP decode	Parses a WLCP PDN Connection Request from the wire.
WLCP encode	Serialises a WLCP PDN Connection Request to the wire.
WLCP encode + decode round-trip	Encode then decode of one WLCP message.
EAP decode Identity packet	Parses an EAP Identity response (NAI extraction).
MPPE build VSA	Builds the MS-MPPE-Send-Key RADIUS VSA (RC4 + MD5) carrying the MSK.

RADIUS Shared Secrets (per source-IP group)

Overview

OmniTWAG authenticates each RADIUS client (an access point, or NAS) with a shared secret. This feature lets an operator manage **many** shared secrets, each tied to one or more source-IP subnets. A group of access points that share a subnet also shares one secret. The operator adds, changes, and removes these credentials on demand through the REST API or the control-panel web interface. The store is persistent, so credentials survive a restart.

Table of Contents

- [Concepts](#)
- [How authentication uses a credential](#)
- [Credential model](#)
- [Per-AP breakout and charging](#)
- [REST API](#)
- [Web interface](#)
- [Configuration](#)
- [Migration from a single static secret](#)
- [Metrics](#)
- [Troubleshooting](#)

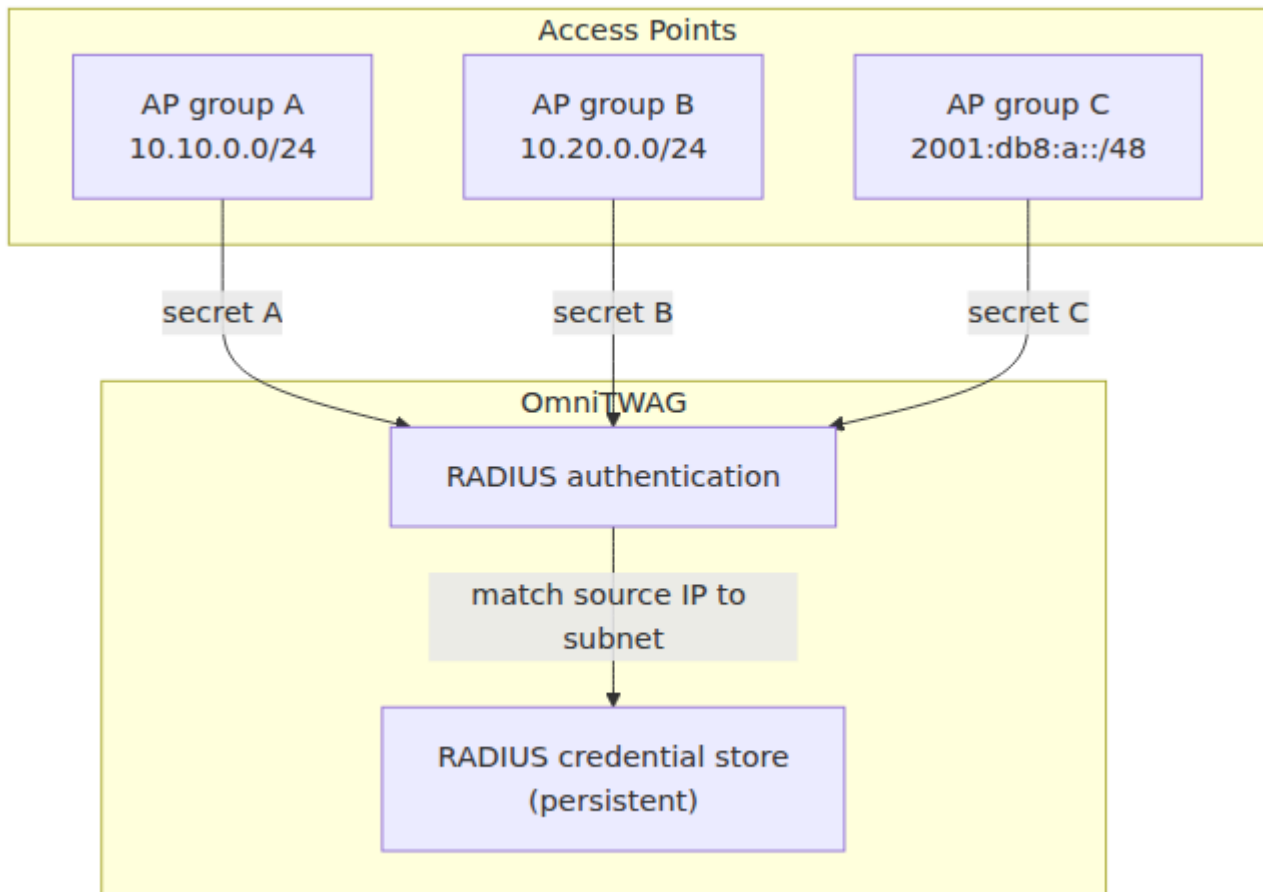
Concepts

A **credential** ties one shared secret to one or more **source subnets**. A source subnet is a CIDR block. The subnet can be IPv4, IPv6, or a mix of both.

The **source IP** is the IP address of the access point that sends the RADIUS packet. OmniTWAG reads the source IP from the received UDP packet.

OmniTWAG does not use the `NAS-IP-Address` attribute for secret selection, because an attacker can set that attribute to any value.

A credential subnet also **authorizes** the access point. If the source IP matches a credential subnet, the access point is authorized. If the source IP matches no credential subnet, OmniTWAG drops the packet. This means that when an operator adds a credential for a new access point, the access point is authorized at the same time.



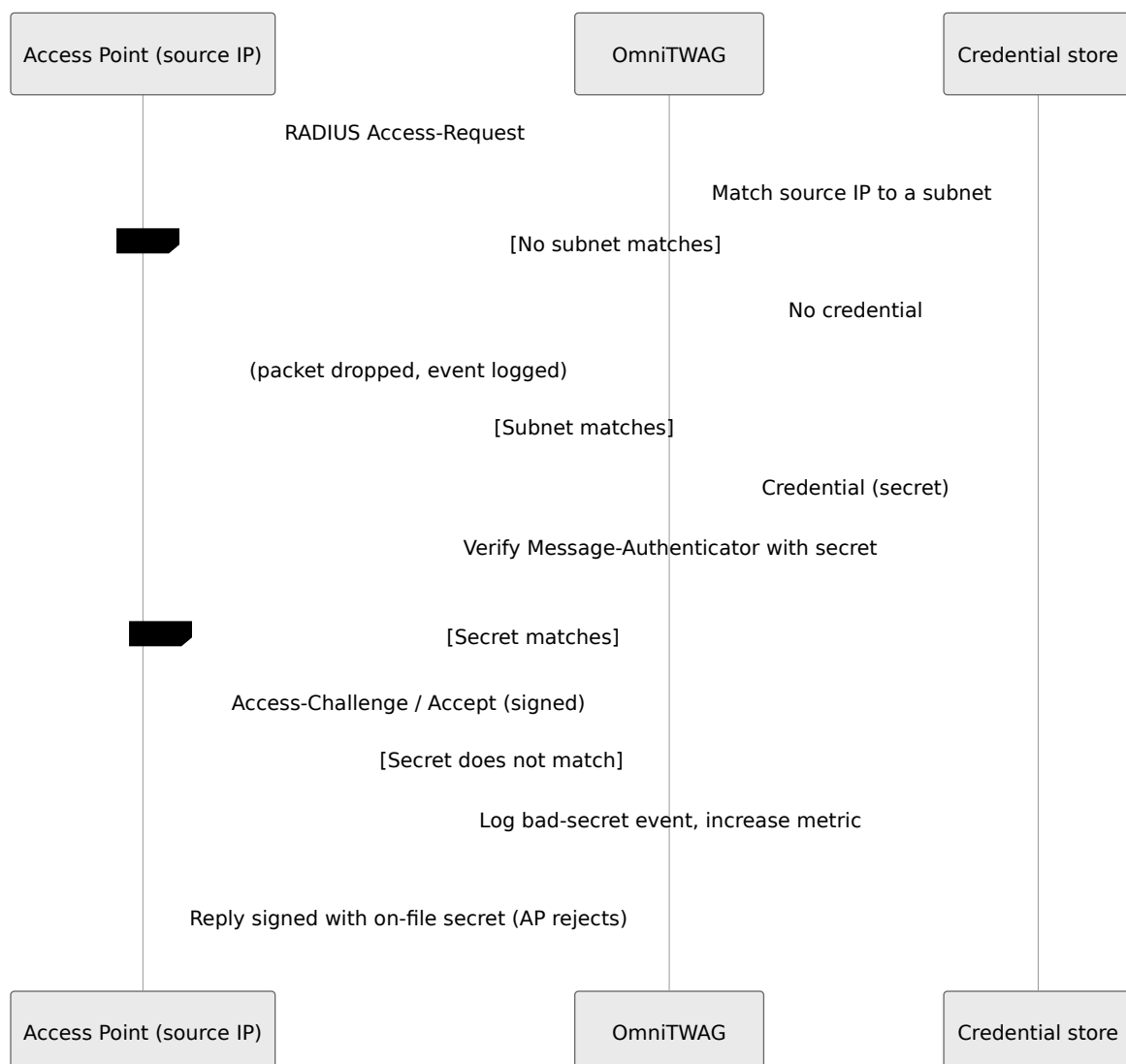
How authentication uses a credential

OmniTWAG resolves the shared secret for every incoming RADIUS packet. The steps are:

1. OmniTWAG reads the source IP of the packet.

2. OmniTWAG finds the credential whose subnet matches the source IP. If more than one subnet matches, OmniTWAG uses the subnet with the **longest prefix** (the most specific match).
3. If no subnet matches, OmniTWAG drops the packet. The source is not authorized.
4. If a subnet matches, OmniTWAG uses that credential's secret to decode the packet, to check the **Message-Authenticator**, to encrypt the MPPE keys, and to sign the reply.

If a subnet matches but the **Message-Authenticator** does not verify, the access point holds a different secret than the one on file. OmniTWAG logs this event and increases a metric. OmniTWAG still processes the packet. The reply is signed with the on-file secret, so the access point rejects the reply and authentication fails. This behaviour makes a misconfigured secret easy to find. It does not drop the packet, so a valid edge case is never dropped by mistake.



Credential model

Each credential has these fields.

Field	Type	Description
<code>id</code>	String	Unique identifier (UUID). The server sets this value.
<code>name</code>	String	Human-readable label, for example <code>Melbourne CBD APs</code> .
<code>secret</code>	String	RADIUS shared secret.
<code>subnets</code>	List of String	One or more CIDR blocks (IPv4, IPv6, or a mix), for example <code>["10.10.0.0/24", "2001:db8:a::/48"]</code> .
<code>breakout_type</code>	String	Data-plane handling for sessions from this access point: <code>tunneled</code> , <code>nsw</code> , or <code>local</code> . Default <code>tunneled</code> . See Per-AP breakout and charging .
<code>address_source</code>	String	For <code>nsw</code> breakout, how the UE IP is assigned: <code>pool</code> or <code>dhcp</code> . Default <code>pool</code> . See Per-AP breakout and charging .
<code>charging_mode</code>	String	Gy online-charging behaviour: <code>charged</code> , <code>uncharged</code> , or <code>auth_only</code> . Default <code>charged</code> . See Per-AP breakout and charging .
<code>ip_pool_id</code>	String	Optional <code>id</code> of an IP pool that addresses <code>nsw</code> sessions from this access point. <code>null</code> means no pool.
<code>inserted_at</code>	Integer	Time the credential was created, in milliseconds since the Unix epoch. The server sets this value.
<code>updated_at</code>	Integer	Time the credential was last changed, in milliseconds since the Unix epoch. The server

Field	Type	Description
		sets this value.

The operator sets `name`, `secret`, `subnets`, `breakout_type`, `address_source`, `charging_mode`, and `ip_pool_id`. The server sets `id`, `inserted_at`, and `updated_at`.

Per-AP breakout and charging

Each credential also controls the data plane and charging for the access points in the group. The defaults preserve the standard behaviour: a GTP tunnel to the PGW with online charging.

breakout_type

`breakout_type` sets how subscriber traffic leaves the TWAG.

Value	Description
<code>tunneled</code>	OmniTWAG establishes a GTP-C/GTP-U session to the PGW and forwards the subscriber traffic over the tunnel. This is the default. See S2a GTP-C Interface .
<code>nswo</code>	Non-Seamless WLAN Offload. OmniTWAG anchors the session locally and does not tunnel to the PGW. OmniTWAG assigns the UE IP from an IP pool . The subscriber traffic transits the TWAG and breaks out to the local network.
<code>local</code>	The access point owns the data plane. OmniTWAG authenticates the subscriber and assigns no IP. The UE gets its address from the local network.

address_source

`address_source` applies only when `breakout_type` is `nswo`. It sets how the UE IP is assigned. See [IP Pools](#).

Value	Description
<code>pool</code>	OmniTWAG pre-allocates a fixed address from the associated IP pool for each session. This is the default.
<code>dhcp</code>	The OmniTWAG DHCP server dishes out an address from the pool range when the UE sends a DHCP DISCOVER.

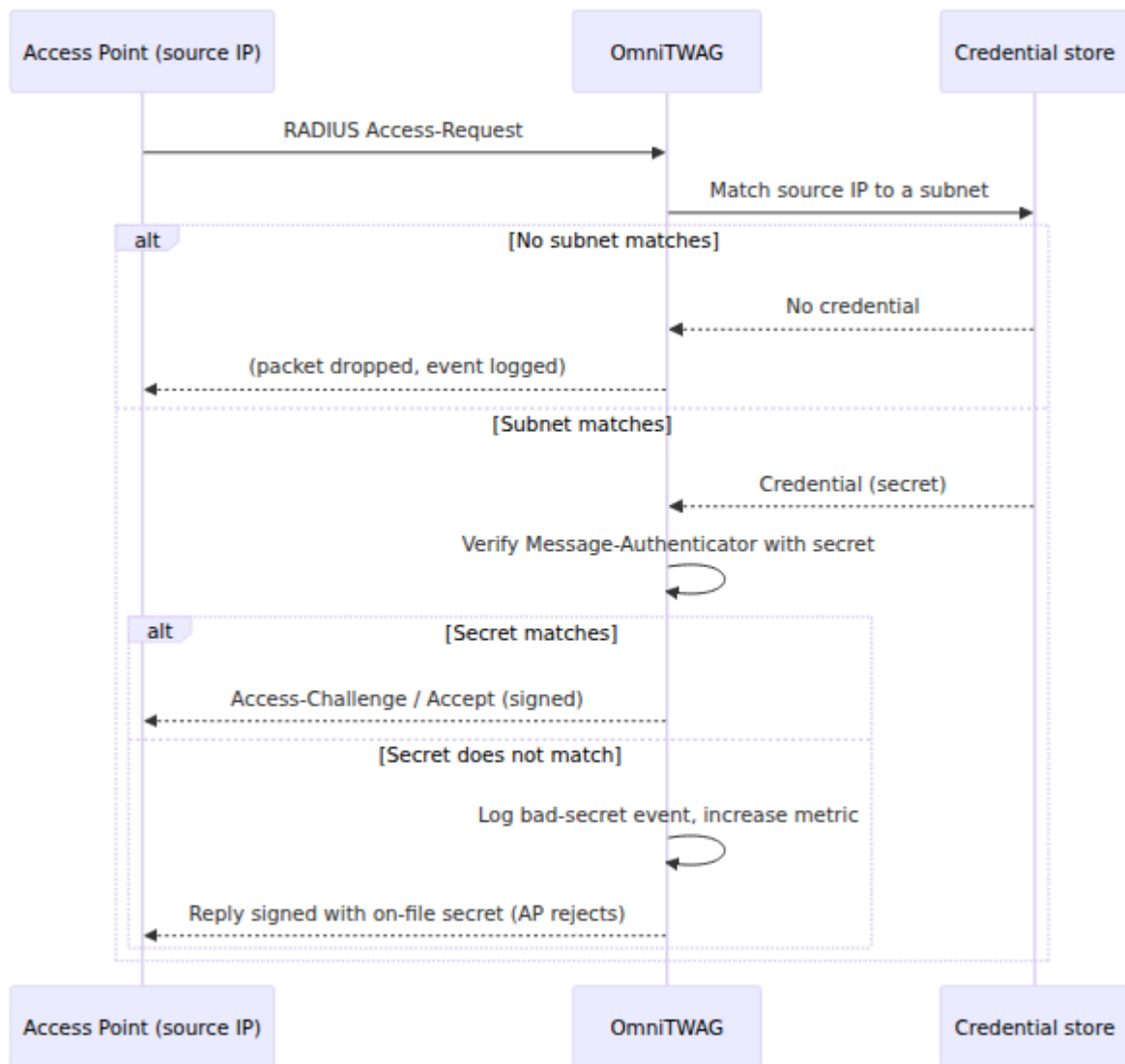
charging_mode

`charging_mode` sets the Gy online-charging behaviour. See [Online Charging](#).

Value	Description
<code>charged</code>	OmniTWAG opens a Gy online-charging session and reports usage to the OCS. This is the default.
<code>uncharged</code>	OmniTWAG establishes the data session but does not open a Gy session. Use this value for flat-rate or non-charged access.
<code>auth_only</code>	OmniTWAG authenticates the subscriber but does not establish a data session. Use this value when another system provides connectivity.

How the fields interact

`charging_mode: auth_only` takes priority. When a credential uses `auth_only`, OmniTWAG does not establish a data session, so `breakout_type` and `address_source` have no effect.



REST API

The OmniTWAG REST API serves these endpoints over HTTPS on the configured `api_ex` port (`8444` by default). The base path is `/radius_credentials`.

Method	Path	Action	Description
GET	/radius_credentials	index	List all credentials.
GET	/radius_credentials/{id}	show	Get one credential by <code>id</code> .
POST	/radius_credentials	create	Create a credential.
PUT	/radius_credentials/{id}	update	Change a credential.
DELETE	/radius_credentials/{id}	delete	Remove a credential.

Request body

`POST` and `PUT` accept a JSON body with these fields.

Field	Type	Required	Description
<code>name</code>	String	No	Human-readable label.
<code>secret</code>	String	Yes	RADIUS shared secret. Must not be empty.
<code>subnets</code>	List of String	Yes	One or more valid CIDR blocks. Must not be empty.
<code>breakout_type</code>	String	No	<code>tunneled</code> , <code>nsw</code> , or <code>local</code> . Default <code>tunneled</code> . See Per-AP breakout and charging .
<code>address_source</code>	String	No	<code>pool</code> or <code>dhcp</code> (for <code>nsw</code>). Default <code>pool</code> .
<code>charging_mode</code>	String	No	<code>charged</code> , <code>uncharged</code> , or <code>auth_only</code> . Default <code>charged</code> .
<code>ip_pool_id</code>	String	No	<code>id</code> of an IP pool for <code>nsw</code> addressing. An empty string means no pool.

`PUT` changes only the fields in the body. Fields that are not in the body keep their current value.

Responses

Status	Meaning
200 OK	The request was successful (index, show, update, delete).
201 Created	The credential was created.
404 Not Found	No credential has the given <code>id</code> .
422 Unprocessable Entity	The <code>secret</code> is empty, a subnet is not a valid CIDR block, or <code>breakout_type</code> / <code>address_source</code> / <code>charging_mode</code> is not a valid value.

Example: list credentials

Request:

```
GET /radius_credentials
```

Response:

```
{
  "count": 1,
  "radius_credentials": [
    {
      "id": "0d1e2f3a-4b5c-6d7e-8f90-1a2b3c4d5e6f",
      "name": "Melbourne CBD APs",
      "secret": "s3cr3t",
      "subnets": ["10.10.0.0/24", "2001:db8:a::/48"],
      "inserted_at": 1734512400000,
      "updated_at": 1734512400000
    }
  ]
}
```

Example: create a credential

Request:

```
POST /radius_credentials
```

```
{
  "name": "Melbourne CBD APs",
  "secret": "s3cr3t",
  "subnets": ["10.10.0.0/24", "2001:db8:a::/48"]
}
```

A successful request returns `201 Created` with the new credential, including its `id`.

Example: an invalid subnet

Request:

```
{
  "secret": "s3cr3t",
  "subnets": ["10.10.0.0/24", "not-a-subnet"]
}
```

Response (`422 Unprocessable Entity`):

```
{
  "error": "invalid CIDR subnet(s): not-a-subnet"
}
```

Web interface

The control panel has a **RADIUS Secrets** page at `/radius_credentials`. The page lists every credential with its name, subnets, a masked secret, and the last-changed time. The page has a form to add a credential and controls to edit

or delete a credential. Enter one or more subnets in the subnets field. Separate the subnets with a comma or a new line.

Configuration

The RADIUS credential store keeps its data in a DETS file. Set the file path in the `radius_config` block.

```
config :omnitwag,  
  radius_config: %{  
    # Path to the persistent RADIUS credential store.  
    credentials_dets_path: "priv/radius_credentials.dets",  
    # Legacy single-secret fields (see "Migration" below).  
    allowed_source_subnets: [],  
    secret: "123456"  
  }
```

Parameter	Type	Required	Default
credentials_dets_path	String	No	priv/radius_credentials.
secret	String	No	123456
allowed_source_subnets	List of String	No	[]

Migration from a single static secret

Earlier versions used one static `secret` and an optional `allowed_source_subnets` allow-list. On first start, if the credential store is empty, OmniTWAG seeds one credential from these legacy fields. The seed runs only when both of these are true:

- The store has no credentials.
- The `secret` is set and `allowed_source_subnets` is not empty.

The seeded credential uses the legacy `secret` and the legacy subnets. This keeps an existing deployment working after an upgrade. OmniTWAG logs a message when it seeds a credential.

If `allowed_source_subnets` is empty, OmniTWAG does not seed a credential. In that case OmniTWAG rejects all RADIUS requests until the operator adds a credential. Add credentials through the [REST API](#) or the [web interface](#).

Per-packet authentication does not read `secret` or `allowed_source_subnets`. These fields serve only as the first-start seed. Manage live credentials through the API or the web interface.

Metrics

OmniTWAG exposes these counters on the Prometheus endpoint. See the [Metrics Reference](#) for the endpoint and query examples.

`radius_auth_unauthorized_source_count`

Type: Counter **Description:** Number of RADIUS packets dropped because the source IP matched no credential subnet.

Label	Description
<code>source</code>	Source IP of the dropped packet.

radius_auth_bad_secret_count

Type: Counter **Description:** Number of RADIUS packets whose source IP matched a credential, but whose Message-Authenticator did not verify. The access point holds a different secret than the one on file.

Label	Description
source	Source IP of the access point.
credential_id	id of the matched credential.

Troubleshooting

An access point gets no response

Symptoms: The access point sends RADIUS requests but receives no reply. The radius_auth_unauthorized_source_count metric increases.

Possible causes:

- No credential subnet contains the source IP of the access point.
- The access point sends from a different IP than expected, for example a NAT address.

Resolution:

1. Find the source IP of the access point in the OmniTWAG logs.
2. Confirm that a credential subnet contains that IP.
3. If no credential covers the IP, add a credential for the correct subnet.

An access point is authorized but authentication fails

Symptoms: The radius_auth_bad_secret_count metric increases for the access point. Clients cannot authenticate.

Possible causes:

- The access point holds a different shared secret than the credential on file.

Resolution:

1. Read the `credential_id` label on the metric, or find the bad-secret message in the logs.
2. Compare the secret on the access point with the secret in the credential.
3. Correct the secret on the access point, or update the credential to match.

Credentials are lost after a restart

Symptoms: The credential list is empty after a restart. OmniTWAG rejects all requests.

Possible causes:

- The `credentials_dets_path` points to a directory that is not writable, or to a path that is cleared on restart.

Resolution:

1. Set `credentials_dets_path` to a writable, persistent data directory.
2. Confirm that the OmniTWAG process can write to that path.
3. Add the credentials again through the API or the web interface.

RADIUS Rejections

Overview

OmniTWAG tracks RADIUS sources whose packets were rejected. This lets an operator spot an unknown or misconfigured access point at a glance. The feature is the mirror image of [RADIUS Shared Secrets](#): the credentials store records the sources that OmniTWAG **accepts**, and the rejection tracker records the sources that OmniTWAG **rejects**.

The tracker holds its data in memory. The data is not persistent. The tracker resets on restart. This is by design. See [Troubleshooting](#).

Table of Contents

- [Concepts](#)
- [How a packet reaches the tracker](#)
- [Rejection reasons](#)
- [Entry model](#)
- [Store behaviour](#)
- [REST API](#)
- [Metrics](#)
- [Troubleshooting](#)

Concepts

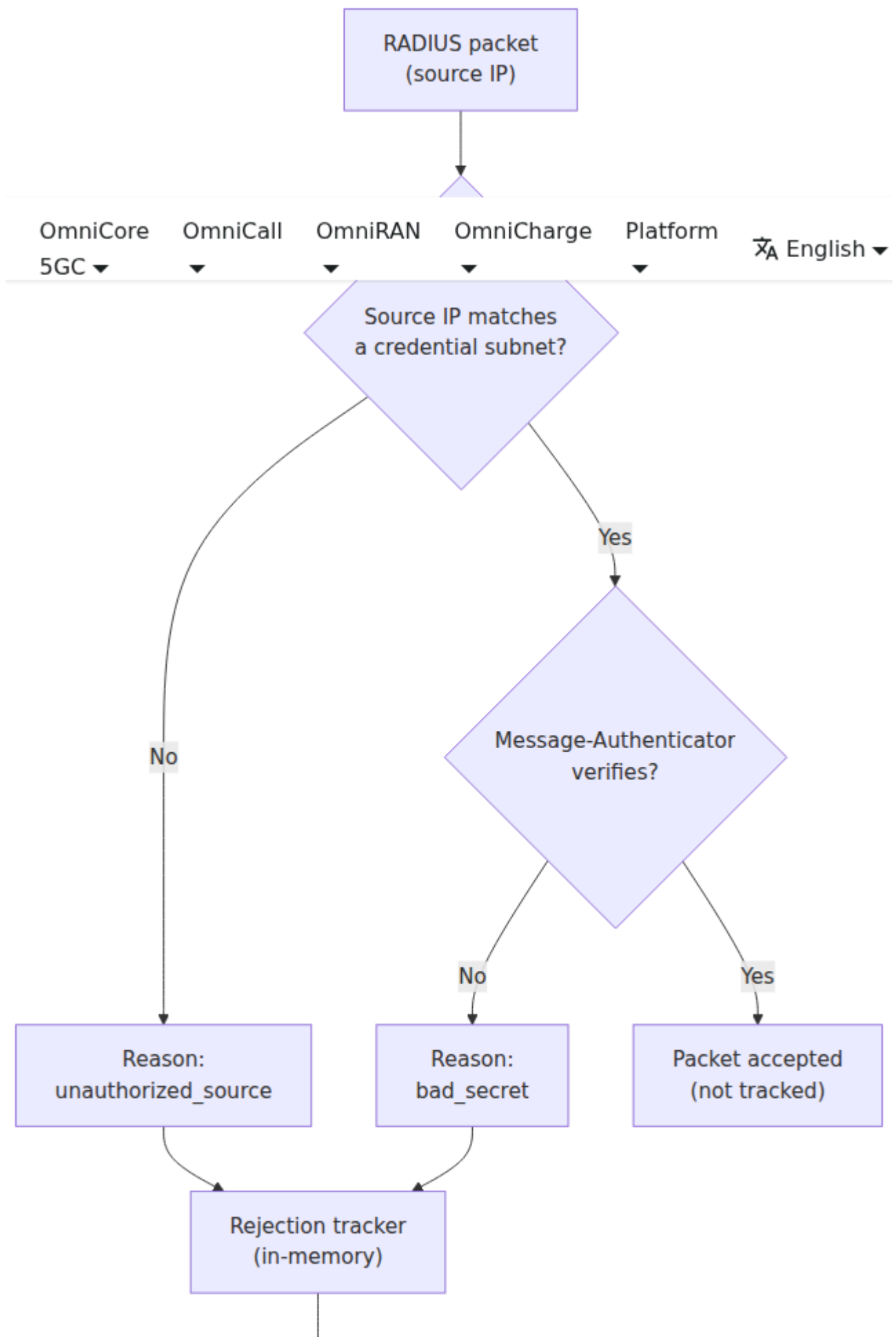
A **rejected source** is an IP address that sent a RADIUS packet that OmniTWAG did not accept. The tracker records one entry per unique pair of **source** and **reason**. If the same source is rejected many times for the same reason, OmniTWAG increases a counter on the existing entry. OmniTWAG does not add a new row.

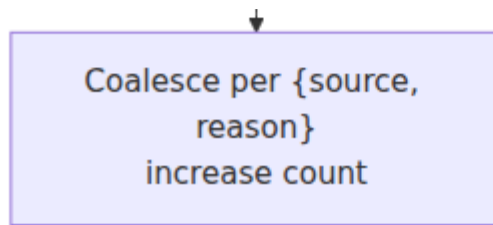
The **source IP** is the IP address that sent the RADIUS packet. OmniTWAG reads the source IP from the received UDP packet. OmniTWAG does not use the `NAS-`

`IP-Address` attribute, because an attacker can set that attribute to any value.

How a packet reaches the tracker

OmniTWAG matches every incoming RADIUS packet against the credential subnets. A packet is rejected when it fails one of two checks. The tracker records the source and the reason for each rejected packet.





Rejection reasons

A source is rejected for one of two reasons.

Reason	Meaning
<code>unauthorized_source</code>	The source IP matched no credential subnet. OmniTWAG dropped the packet. This is the unknown-access-point case.
<code>bad_secret</code>	A credential subnet matched, but the packet <code>Message-Authenticator</code> did not verify. The access point holds a different shared secret than the one on file.

For a `bad_secret` entry, the tracker also records the credential that matched. This tells the operator which credential to check.

Entry model

Each entry has these fields.

Field	Type	Description
<code>source</code>	String	Source IP of the rejected packet. This value is the entry identifier.
<code>reason</code>	String	Reason for the rejection: <code>unauthorized_source</code> or <code>bad_secret</code> .
<code>count</code>	Integer	Number of times OmniTWAG rejected this source for this reason.
<code>first_seen</code>	Integer	Time the source was first rejected, in milliseconds since the Unix epoch.
<code>last_seen</code>	Integer	Time the source was last rejected, in milliseconds since the Unix epoch.
<code>credential_id</code>	String	<code>id</code> of the matched credential. Set only when <code>reason</code> is <code>bad_secret</code> .
<code>credential_name</code>	String	<code>name</code> of the matched credential. Set only when <code>reason</code> is <code>bad_secret</code> .

The `credential_id` and `credential_name` fields are empty for an `unauthorized_source` entry, because no credential matched.

Store behaviour

The tracker is an in-memory store. It has these properties.

- **Non-persistent.** The store resets on restart. It does not survive a process restart.
- **Coalesced.** OmniTWAG records one entry per pair of `source` and `reason`. A repeat rejection increases `count` and updates `last_seen`. It does not add a row.

- **Bounded.** The store holds a maximum number of sources. The default maximum is 256.
- **Least-recently-seen eviction.** When the store is full, OmniTWAG removes the entry with the oldest `last_seen` value to make room for a new source. The store is bounded because a source IP can be spoofed. A flood of spoofed source IPs cannot grow the store without limit.

REST API

The OmniTWAG REST API serves these endpoints on the configured API port (8444 by default). The base path is `/radius_rejections`.

Method	Path	Action	Description
GET	<code>/radius_rejections</code>	index	List rejected sources. The list is paginated. Search by <code>source</code> or <code>reason</code> .
DELETE	<code>/radius_rejections/{id}</code>	delete	Forget a source. <code>{id}</code> is the source IP.

Use `DELETE` to forget a source after you resolve it, for example after you add a credential that covers the source IP.

Responses

Status	Meaning
<code>200 OK</code>	The request was successful.
<code>404 Not Found</code>	No entry has the given source IP.

Example: list rejected sources

Request:

```
GET /radius_rejections
```

Response:

```
{
  "count": 2,
  "radius_rejections": [
    {
      "source": "203.0.113.7",
      "reason": "unauthorized_source",
      "count": 42,
      "first_seen": 1734512400000,
      "last_seen": 1734512999000,
      "credential_id": null,
      "credential_name": null
    },
    {
      "source": "10.10.0.15",
      "reason": "bad_secret",
      "count": 3,
      "first_seen": 1734512500000,
      "last_seen": 1734512890000,
      "credential_id": "0d1e2f3a-4b5c-6d7e-8f90-1a2b3c4d5e6f",
      "credential_name": "Melbourne CBD APs"
    }
  ]
}
```

Example: forget a source

Request:

```
DELETE /radius_rejections/203.0.113.7
```

Response:

```
{
  "deleted": "203.0.113.7"
}
```

Metrics

The rejection tracker complements two Prometheus counters that the credentials feature documents. Each counter increases at the same time that OmniTWAG records a rejection.

Counter	Rejection reason
<code>radius_auth_unauthorized_source_count</code>	<code>unauthorized_source</code>
<code>radius_auth_bad_secret_count</code>	<code>bad_secret</code>

The counters give totals over time. The tracker gives a live, per-source view. See [RADIUS Shared Secrets](#) for the counter definitions, and the [Metrics Reference](#) for the Prometheus endpoint and query examples.

Troubleshooting

An unknown access point appears in the list

Symptoms: A source appears with `reason` set to `unauthorized_source`. The `radius_auth_unauthorized_source_count` metric increases for that source.

Possible causes:

- No credential subnet contains the source IP.
- The access point is behind NAT. The RADIUS source IP that OmniTWAG sees differs from the address of the access point itself.

Resolution:

1. Read the `source` field of the entry.
2. Confirm that this is a known access point.
3. Add a credential subnet that covers the source IP that OmniTWAG actually sees. For a NAT case, use the translated address, not the address of the access point. See [RADIUS Shared Secrets](#).
4. Delete the entry with `DELETE /radius_rejections/{id}` to forget the source.

A known access point appears as bad_secret

Symptoms: A known source appears with `reason` set to `bad_secret`. The `radius_auth_bad_secret_count` metric increases for that source.

Possible causes:

- The access point holds a different shared secret than the credential on file.

Resolution:

1. Read the `credential_id` and `credential_name` fields of the entry. These name the matched credential.
2. Compare the secret on the access point with the secret in that credential.
3. Correct the secret on the access point, or update the credential to match. See [RADIUS Shared Secrets](#).
4. Delete the entry with `DELETE /radius_rejections/{id}` to forget the source.

The list is empty after a restart

Symptoms: The rejection list is empty after a restart, even though sources were listed before the restart.

Cause:

- This is expected behaviour. The tracker holds its data in memory. The data is not persistent. The store resets on restart.

Resolution:

- No action is needed. New rejections repopulate the list. Use the `radius_auth_unauthorized_source_count` and `radius_auth_bad_secret_count` metrics for a history that survives a restart.

Overview

S2a / GTP-C Interface to the PGW

Overview

The S2a interface connects the OmniTWAG to the Packet Data Network Gateway (PGW). It carries GTP-based PDN connectivity for a Trusted WLAN Access Network (TWAN). OmniTWAG acts as the TWAN GTP-C endpoint. This role is like an ePDG on the S2b interface, but it uses S2a interface types and a TWAN Identifier.

The S2a interface has two planes:

- **GTP-C (control plane)**: creates, modifies, and deletes PDN sessions. It also runs path management. This document covers GTP-C.
- **GTP-U (user plane)**: carries UE data in GTP tunnels. See the [User Plane](#) guide.

GTP-C on S2a follows [3GPP TS 29.274](#) (GTPv2-C). The TWAG uses GTPv2-C message types and information elements (IEs) with S2a-specific interface types.

Table of Contents

- [Where S2a Fits: The Three Breakout Modes](#)
- [Role in the Session Lifecycle](#)
- [S2a Interface Types](#)
- [Session Establishment](#)
 - [Trigger](#)
 - [Create Session Request](#)
 - [Default Bearer](#)
 - [Create Session Response](#)
 - [Emergency Sessions](#)

- [Bearer Management](#)
- [Session Teardown](#)
- [Path Management](#)
- [Configuration](#)
- [Reference Tables](#)
- [Observability](#)
- [Troubleshooting](#)
- [3GPP References](#)

Where S2a Fits: The Three Breakout Modes

OmniTWAG resolves a data-plane mode per access point (AP), from the RADIUS credential that matches the AP. The `breakout_type` value on the credential selects one of three modes. See [RADIUS Credentials](#) for the `breakout_type`, `address_source`, and `ip_pool_id` settings.

Mode (<code>breakout_type</code>)	Who assigns the UE IP	Data plane	S2a / GTP used
<code>tunneled</code>	PGW (in the PDN Address Allocation IE)	GRE from the AP to the TWAG, then GTP-U on S2a to the PGW, then out to the internet by the PGW	Yes. This document.
<code>nswo</code>	TWAG, from an IP pool	GRE from the AP to the TWAG, then local breakout to the internet (NAT) at the TWAG	No. No PGW session.
<code>local</code>	The local network at the AP	The AP owns the data plane. The TWAG carries no traffic.	No. No PGW session, no IP.

S2a GTP-C applies only to `tunneled` mode. In `tunneled` mode the PGW allocates the UE IP address and applies policy and charging. In `nswo` and `local` modes the TWAG does not create a PDN session, so no S2a message is sent.

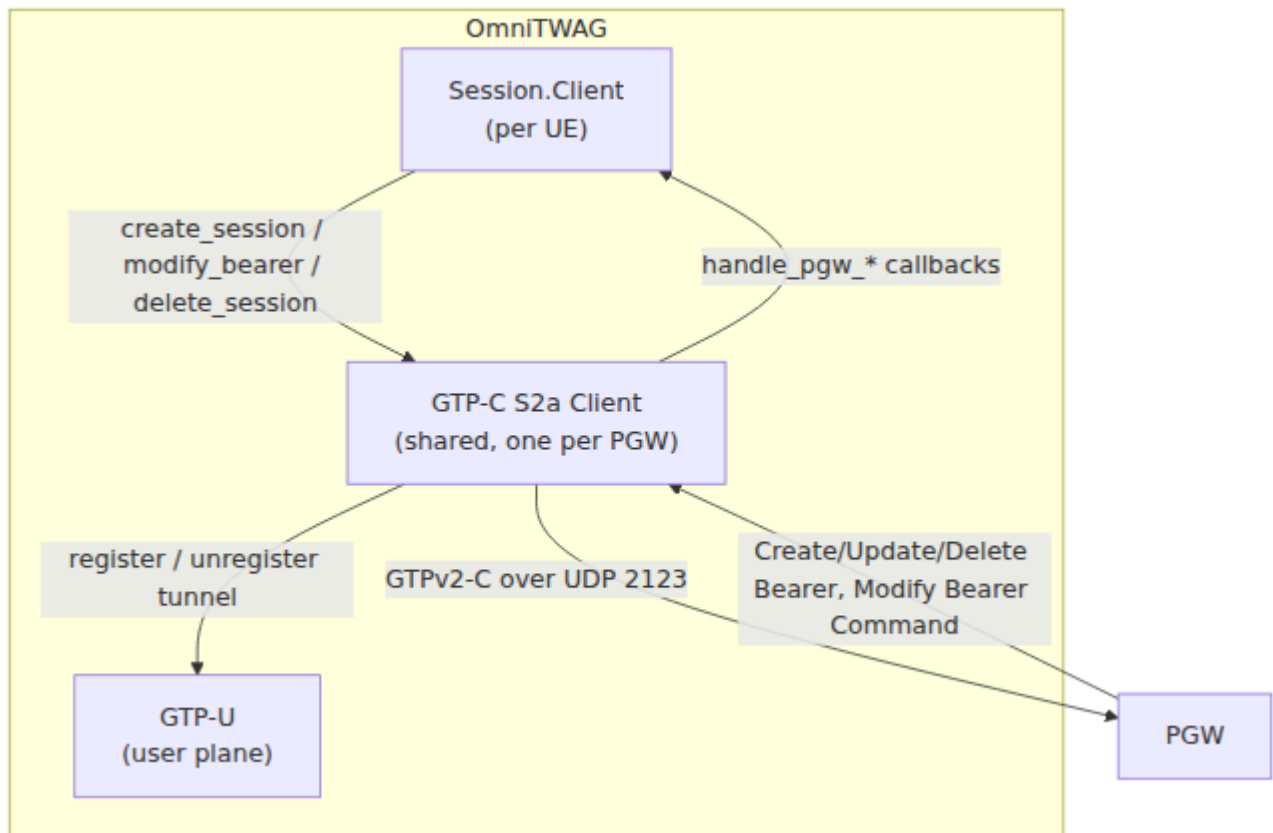
Charging in `tunneled` mode is done by the PGW, not the TWAG. The TWAG-side Gy online charging in **Online Charging** applies to the locally-broken-out modes.

Note: A separate global Non-Seamless WLAN Offload (NSWO) handler (`Twag.NSWO.Handler`, enabled with `nswo.enabled: true`) provides local breakout for every session when GTP is not used. See the **Local Breakout** section of the User Plane guide. The per-AP `nswo` breakout type described above is the credential-driven model and is preferred.

Role in the Session Lifecycle

The GTP-C client is a single, shared process (`Twag.Gtp.GtpcS2a`). It handles all sessions against one configured PGW. Each authenticated `tunneled` UE session

drives one Create Session exchange. The per-session `Session.Client` process calls the GTP-C client. It does not send GTPv2-C messages itself.



S2a Interface Types

The TWAG uses S2a F-TEID interface types. These are different from S2b (ePDG), which uses types 30 and 31. The values follow [TS 29.274 Table 8.22-1](#).

Interface Type	Value	Direction	Use
s2a_twan_gtp_c	35	TWAG control	TWAG F-TEID in Create Session Request
s2a_twan_gtp_u	34	TWAG user	TWAG bearer F-TEID (GTP-U)
s2a_pgw_gtp_c	36	PGW control	PGW F-TEID in Create Session Response
s2a_pgw_gtp_u	37	PGW user	PGW bearer F-TEID (GTP-U)

The RAT Type is always WLAN (value 3).

Session Establishment

Trigger

A successful EAP authentication starts the session. The AP sends an Accounting-Start, which triggers post-accept processing. The `Session.Client` resolves the AP breakout mode. For `tunneled` mode it calls the GTP-C client to create the PDN session. If the `gtp.enabled` parameter is `false`, the TWAG skips S2a. If the session has no IMSI, the TWAG does not create the session. For `nswo` and `local` modes the TWAG does not send any S2a message. See [Sessions](#) for the full session lifecycle.

Create Session Request

The TWAG sends a Create Session Request (message type per TS 29.274 §7.2.1) to the PGW. The TWAG generates a new, non-predictable control-plane TEID and a user-plane TEID for the request. It builds the request with these IEs:

Information Element	Source / Value	Notes
IMSI	UE subscription	Omitted if not present.
Serving Network (MCC/MNC)	<code>gtp.mcc</code> / <code>gtp.mnc</code> , or first digits of the IMSI	Omitted if either value is missing.
RAT Type	WLAN (3)	Fixed for S2a.
F-TEID (control)	Interface type 35, TWAG control TEID, TWAG local IP	Identifies the TWAG GTP-C endpoint.
Access Point Name (APN)	<code>gtp.apn</code> , default <code>internet</code> . Emergency APN if requested.	See Emergency Sessions .
Selection Mode	2 (network-provided, subscription not verified)	The APN is always network-provided in TWAN.
PDN Address Allocation (PAA)	Requested type with a zero address	The TWAG requests an address. The PGW assigns it.
Aggregate Maximum Bit Rate (APN-AMBR)	Default 256000 uplink / 256000 downlink	Overridden if a rate is supplied.
Bearer Context to Create	Default EBI 5, bearer F-TEID (type 34), QoS	See Default Bearer .
APN Restriction	2 (normal), or 0 for emergency	
ME Identity (IMEISV)	UE equipment identity	Omitted if not present.

Information Element	Source / Value	Notes
Recovery	TWAG recovery counter	Used for restart detection.
Charging Characteristics	0x0800	Fixed value.
TWAN Identifier	SSID from <code>twan_ssid</code> , optional BSSID from <code>twan_bssid</code>	Identifies the WLAN.
UE Time Zone	Derived from the host local time offset	
Indication	Handover flag	The handover flag is set for a handover-in.

The PDN type defaults to IPv4. It can be set to `:ipv4`, `:ipv6`, or `:ipv4v6` through the `gtp.pdn_type` parameter. The requested PAA address is always zero, so the PGW assigns the real address.

Note: The Additional Protocol Configuration Options (APCO) IE is not included. The TWAG does not request DNS or P-CSCF addresses through APCO at this time. It reads DNS from the PCO or Additional PCO IE in the response if the PGW supplies one.

Default Bearer

The TWAG creates one default bearer per session. The default EBI is 5. The bearer has no TFT filters, so it carries all traffic. The bearer-level QoS in the request uses these fixed values:

QoS Parameter	Value
QCI	9
Priority Level	10
Pre-emption Capability	disabled
Pre-emption Vulnerability	enabled
GBR / MBR (uplink and downlink)	0

The TWAG does not support dedicated bearers. See [Bearer Management](#).

Create Session Response

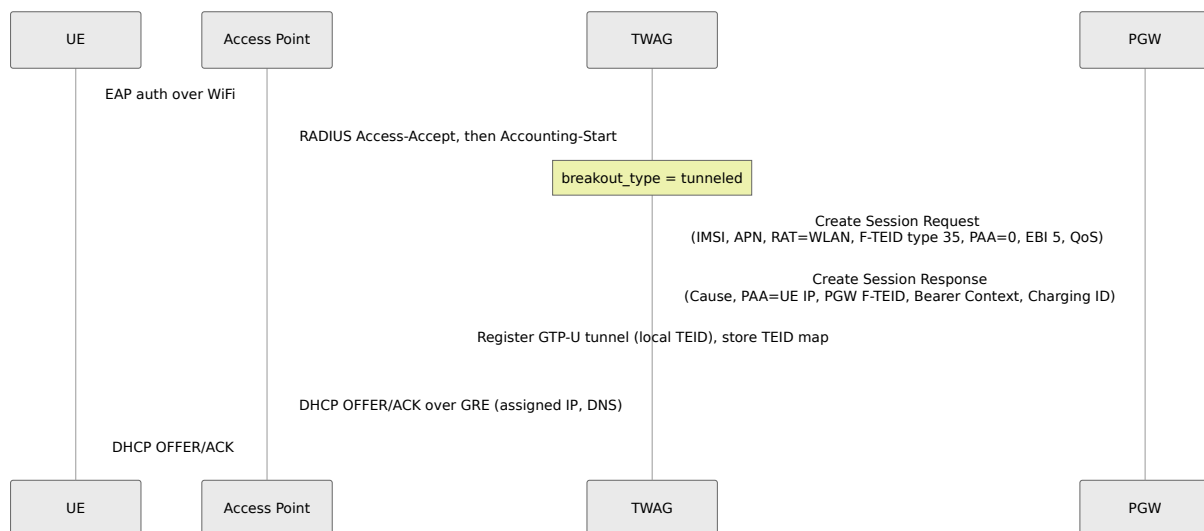
The TWAG matches the response to the pending request by sequence number. It checks the Cause IE. The request is a success if the cause is `request_accepted`, `new_pdp_type_due_to_network_preference`, or `new_pdp_type_due_to_single_address_bearer_only`. Any other cause returns a `gtpc_error` with that cause value.

On success, the TWAG reads these values from the response:

Response Element	Extracted Value	Use
PDN Address Allocation (PAA)	Assigned UE IP address	Delivered to the UE by DHCP over GRE. See IP Address Management .
F-TEID (PGW control)	PGW control TEID and PGW IP	Used as the destination for later control messages.
Bearer Context	Bearer F-TEID (PGW GTP-U TEID and IP), EBI, Charging ID	Used to register the GTP-U tunnel.
APN-AMBR	Session rate limits	Stored on the session.
PCO / Additional PCO	DNS server addresses	Delivered to the UE by DHCP.

The TWAG stores a mapping from its local control TEID to the PGW control TEID and the UE identity. This mapping routes later PGW-initiated messages to the correct session. If no bearer F-TEID is present, the TWAG falls back to the PGW F-TEID for the user-plane tunnel.

After a success, the `Session.Client` registers the GTP-U tunnel (which generates the TWAG-side local user-plane TEID), pre-allocates the DHCP lease with the assigned IP and DNS, and registers the UE for GRE downlink routing. See [User Plane](#) for the tunnel and downlink path.



Emergency Sessions

If the session requests emergency service, the TWAG uses the emergency APN. The `emergency_apn` parameter sets this APN. The default value is `sos`. For an emergency session, the APN Restriction is set to 0 (no restriction).

Bearer Management

Default Bearer Only

OmniTWAG supports the default bearer only. It does not support dedicated bearers. This follows the TWAN Transparent Single-Connection Mode (TSCM) profile. The bearer manager can track a bearer table keyed by EBI, but S2a establishes only the default bearer.

Modify Bearer (TWAG-Initiated)

The TWAG can send a Modify Bearer Request (TS 29.274 §7.2.7). This request carries a bearer context with the default EBI, the TWAG user-plane F-TEID (type 34), and the TWAG local IP. The destination TEID is the PGW control TEID. The TWAG matches the Modify Bearer Response by sequence number and checks the Cause IE for success. A handover-in reuses the Create Session path with the handover flag set, not a separate Modify Bearer.

PGW-Initiated Procedures

The PGW can start bearer procedures. The TWAG receives these on its GTP-C socket. The destination TEID in the message is the TWAG local control TEID. The TWAG uses the stored TEID map to find the PGW control TEID and the UE identity. It then notifies the matching `Session.Client` process.

PGW Message	TWAG Response	Behaviour
Create Bearer Request	Create Bearer Response, cause <code>request_rejected</code>	The TWAG rejects all dedicated bearers (TSCM). No bearer is created.
Modify Bearer Command	Modify Bearer Request back to the PGW	The TWAG acknowledges the new parameters (APN-AMBR, F-TEID). If the send fails, it sends a Modify Bearer Failure Indication with cause <code>system_failure</code> .
Update Bearer Request	Update Bearer Response, cause <code>request_accepted</code> per bearer	The TWAG accepts the QoS update and notifies the session.
Delete Bearer Request	Delete Bearer Response, cause <code>request_accepted</code>	See Session Teardown .

For a Modify Bearer Command, the TWAG reads the APN-AMBR and the bearer context (EBI, new F-TEID, QoS). It notifies the session process, then sends a Modify Bearer Request to the PGW to acknowledge. If the bearer context carries a new GTP-U F-TEID, the session re-registers the GTP-U tunnel with the new remote values. For an Update Bearer Request, the TWAG reads the APN-AMBR and each bearer context (EBI, QoS), notifies the session, then accepts each bearer in the response.

Session Teardown

Delete Session (TWAG-Initiated)

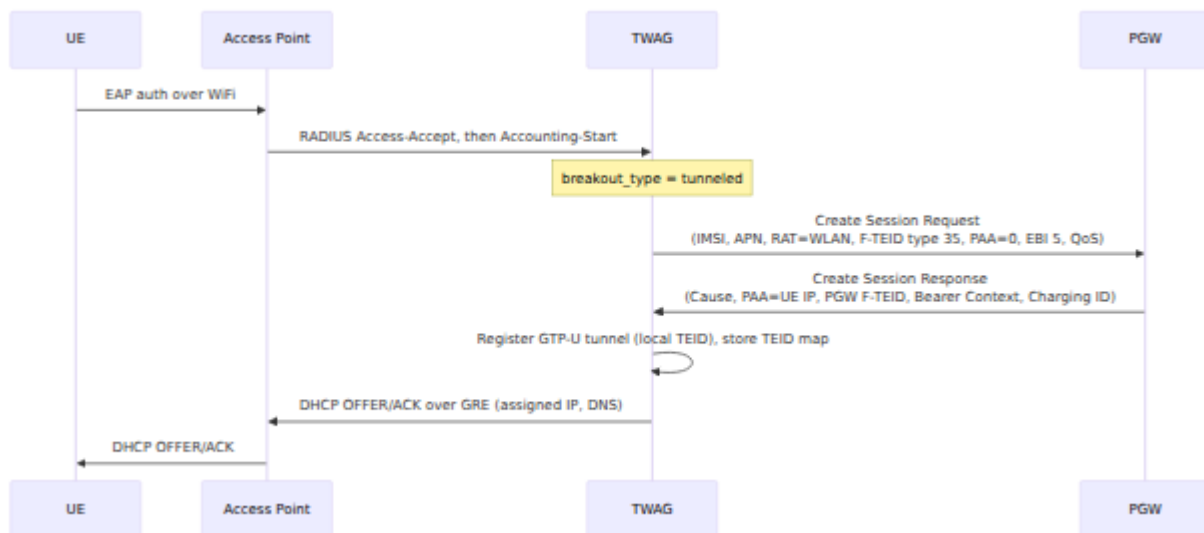
The TWAG deletes the PDN session when the UE disconnects. The `Session.Client` teardown calls the GTP-C client. The TWAG sends a Delete Session Request (TS 29.274 §7.2.9) to the PGW control TEID. The request carries:

- EPS Bearer ID (the default EBI, default 5)
- Cause (default `local_detach`)

The TWAG matches the Delete Session Response by sequence number and checks the Cause IE. It then removes the TEID map entries for that PGW control TEID. If the session has no PGW control TEID (for example an `nsw` or `local` session), the TWAG skips the GTP-C delete. After the delete, the `Session.Client` unregisters the GTP-U tunnel, releases the DHCP lease and any IP-pool address, unregisters the UE from GRE, and cleans up the session.

Delete Bearer (PGW-Initiated)

The PGW can start teardown with a Delete Bearer Request. The TWAG reads the linked EBI, the EBI list, and the cause. It notifies the matching `Session.Client` process, which does a local-only cleanup. The TWAG does not send a Delete Session Request in this case, because the PGW already deleted the bearer. The TWAG then sends a Delete Bearer Response with cause `request_accepted`. If the linked EBI is the default bearer, the whole PDN connection is torn down.



Path Management

Echo

The TWAG runs GTPv2-C path management with Echo Request and Echo Response (TS 29.274 §6). The `echo_enabled` parameter turns this on. It is on by default. The TWAG sends an Echo Request to the PGW every 60 seconds. Each Echo Request carries the TWAG sequence number. Each Echo Response carries a Recovery counter.

When the PGW sends an Echo Request, the TWAG replies with an Echo Response that carries the TWAG Recovery counter.

Path State

The TWAG tracks the path state to the PGW. A successful Echo Response marks the path UP and resets the failure count. Consecutive Echo failures increment a counter. After 3 consecutive failures, the TWAG marks the path DOWN.

PGW Restart Detection

The TWAG stores the PGW Recovery counter from each Echo Response. If the counter changes, the TWAG detects a PGW restart. It then cleans up all

affected sessions. It notifies each session process with a `pgw_restart` cause and clears the TEID map.

Request Timeout and Retransmission

The TWAG uses a T3-RESPONSE timer and N3-REQUESTS retries for control requests.

Timer / Counter	Value	Meaning
T3-RESPONSE	3000 ms	Time to wait for a response before a retransmission.
N3-REQUESTS	3	Maximum retransmissions before the request fails.

If a request times out, the TWAG retransmits the same message with the same sequence number. It restarts the timer. After 3 retries with no response, the TWAG fails the request. A Create Session failure returns `timeout` to the caller. Echo retransmissions follow the same timer and retry limits. An Echo failure counts toward the path-down threshold.

The TWAG opens the GTP-C UDP socket at startup. If the socket cannot open, the TWAG retries every 5 seconds.

Configuration

S2a is configured under the `gtp` map. This map is part of the `omnitwag` application configuration in `runtime.exs`.

```
config :omnitwag,  
  gtp: %{\n    # Enable the S2a interface (needed for tunneled breakout)\n    enabled: true,\n\n    # PGW GTP-C endpoint\n    pgw_ip: "10.5.198.1",\n    pgw_port: 2123,\n\n    # TWAG local IP for GTP-C and GTP-U F-TEIDs\n    local_ip: "10.5.198.200",\n\n    # GTP-U user-plane port\n    gtpu_port: 2152,\n\n    # Path management (Echo)\n    echo_enabled: true,\n\n    # Create Session Request defaults\n    apn: "internet",\n    pdn_type: :ipv4\n  }\n}
```

GTP Parameters

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>false</code>	Enable the S2a interface. Set to <code>true</code> for tunneled breakout. When <code>false</code> , the TWAG does not start the GTP-C client and skips S2a for every session.
<code>pgw_ip</code>	String	Yes (if enabled)	<code>127.0.0.1</code>	PGW GTP-C IP address. The TWAG sends all GTPv2-C messages to this address.
<code>pgw_port</code>	Integer	No	<code>2123</code>	PGW GTP-C UDP port. The standard GTP-C port is 2123.
<code>local_ip</code>	String	No	Auto-detected	TWAG local IP. Used in F-TEIDs and as the source for GTP-C and GTP-U. If not set, the TWAG auto-detects the outbound IP.
<code>gtpu_port</code>	Integer	No	<code>2152</code>	GTP-U UDP port for the user plane. See User Plane .

Parameter	Type	Required	Default	Description
<code>echo_enabled</code>	Boolean	No	<code>true</code>	Enable GTPv2-C Echo path management to the PGW.
<code>apn</code>	String	No	<code>internet</code>	Access Point Name for the PDN session.
<code>pdn_type</code>	Atom	No	<code>:ipv4</code>	PDN type: <code>:ipv4</code> , <code>:ipv6</code> , or <code>:ipv4v6</code> .
<code>mcc</code>	String	No	First 3 IMSI digits	Mobile Country Code for the Serving Network IE.
<code>mnc</code>	String	No	IMSI digits 4-5	Mobile Network Code for the Serving Network IE.

Emergency and TWAN Parameters

The following parameters are read from the `omnitwag` application environment, not from the `gtp` map.

Parameter	Type	Default	Description
<code>emergency_apn</code>	String	<code>sos</code>	APN used for an emergency session. APN Restriction is set to 0 for these sessions.
<code>twan_ssid</code>	String	<code>TWAG</code>	SSID placed in the TWAN Identifier IE.
<code>twan_bssid</code>	Binary	(unset)	Optional 6-byte BSSID for the TWAN Identifier IE. Omitted if not a valid 6-byte value.

Note: The GTP-C client connects to a single PGW. There is no per-APN or multi-PGW selection at this time. To change the PGW, update `pgw_ip`.

Reference Tables

GTPv2-C Message Types (S2a)

Message	Direction	Purpose
Echo Request / Response	Both	Path management.
Create Session Request / Response	TWAG to PGW	Establish the PDN session.
Modify Bearer Request / Response	TWAG to PGW	Update the bearer F-TEID or QoS.
Delete Session Request / Response	TWAG to PGW	Tear down the PDN session.
Create Bearer Request / Response	PGW to TWAG	Dedicated bearer. Rejected by the TWAG.
Update Bearer Request / Response	PGW to TWAG	Update QoS. Accepted by the TWAG.
Modify Bearer Command	PGW to TWAG	PGW asks the TWAG to update bearer parameters.
Modify Bearer Failure Indication	TWAG to PGW	Sent when a Modify Bearer Command cannot be honoured.
Delete Bearer Request / Response	PGW to TWAG	PGW-initiated teardown.

GTPv2-C Cause Codes

The TWAG uses these cause values. The codes follow [TS 29.274 §8.4](#).

Code	Name	Meaning
2	Local Detach	Default cause for a TWAG-initiated Delete Session.
13	Network Failure	Peer-reported network failure.
16	Request Accepted	Success.
64	Context Not Found	The peer has no matching context.
72	System Failure	Used in a Modify Bearer Failure Indication.
92	User Authentication Failed	Authentication rejected.
93	APN Access Denied	The APN is not allowed.
94	Request Rejected	Used when the TWAG rejects a Create Bearer Request.
96	IMSI/IMEI Not Known	The subscriber is not known.

Observability

The TWAG emits telemetry events for bearer and teardown activity. These events drive the metrics in the [Metrics Reference](#).

Event	When
<code>[:twag, :bearer, :modify_command]</code>	The TWAG receives a PGW Modify Bearer Command.
<code>[:twag, :bearer, :update]</code>	The TWAG receives a PGW Update Bearer Request.
<code>[:twag, :detach, :pgw_delete]</code>	The TWAG receives a PGW Delete Bearer Request.
<code>[:twag, :session, :teardown]</code>	The session is torn down.
<code>[:twag, :detach, :cleanup_start]</code>	Session teardown starts.
<code>[:twag, :detach, :cleanup_complete]</code>	Session teardown completes.

Client Status

The GTP-C client reports a status snapshot. This status shows the health of the S2a path.

Field	Meaning
<code>connected</code>	The GTP-C UDP socket is open.
<code>pgw_ip</code> / <code>pgw_port</code>	The configured PGW endpoint.
<code>pending_requests</code>	Count of in-flight GTP-C requests.
<code>path_up</code>	The Echo path state to the PGW.
<code>echo_failures</code>	Count of consecutive Echo failures.
<code>recovery_counter</code>	The TWAG Recovery counter.

Logging

The GTP-C client logs each procedure. Key log lines include:

- Create Session, Delete Session, and Modify Bearer sends and responses.
- Path state changes: `Path to PGW ... is UP` and `Path to PGW ... is DOWN`.
- PGW restart detection with the old and new Recovery counters.
- PGW-initiated Create, Update, Modify, and Delete Bearer messages.

Troubleshooting

Create Session Fails

Symptoms: Sessions do not establish. The log shows `GTP Create Session failed`.

Possible causes:

- The PGW rejects the request with a non-success cause.
- The path to the PGW is down.

- The APN is not provisioned on the PGW.
- The IMSI is not known to the core.

Resolution:

1. Check the returned cause code against the **Cause Codes** table.
2. Check the path state with the client status (`path_up`).
3. Confirm the APN is provisioned on the PGW.
4. Confirm the subscriber is provisioned in the HSS.

Path to PGW is Down

Symptoms: The log shows `Path to PGW ... is DOWN`. New sessions fail.

Possible causes:

- The PGW is unreachable on UDP 2123.
- A firewall blocks GTP-C.
- The PGW is not running.

Resolution:

1. Confirm the `pgw_ip` and `pgw_port` values are correct.
2. Confirm the network path allows UDP 2123 in both directions.
3. Confirm the PGW GTP-C service is running.
4. Wait for the next Echo. A successful Echo Response marks the path UP.

Sessions Drop After a PGW Restart

Symptoms: All sessions tear down together. The log shows `PGW restart detected`.

Possible causes:

- The PGW restarted and changed its Recovery counter.

Resolution:

1. This is expected behaviour. The TWAG cleans up stale sessions after a PGW restart.
2. Confirm the PGW is stable. UEs re-authenticate and re-establish sessions.

Sessions Establish but No Data Flows

Symptoms: The UE gets an IP address, but traffic does not pass.

Possible causes:

- The GTP-U tunnel or the GRE path is not up.

Resolution:

1. Confirm `gtp.enabled` and `gre.enabled` are both `true` for tunneled breakout.
2. Check the GTP-U and GRE statistics in the [User Plane](#) guide.

3GPP References

Specification	Title
TS 29.274	Evolved GPRS Tunnelling Protocol for Control plane (GTPv2-C).
TS 23.402	Architecture enhancements for non-3GPP accesses (TWAN, S2a).
TS 24.302	Access to the EPC via non-3GPP access networks.

[Overview](#)

Sessions

Overview

A **session** represents one authenticated subscriber on OmniTWAG. OmniTWAG keys a session by subscriber identity (the IMSI or username). OmniTWAG holds sessions in memory. The Sessions API serves the live session state.

The Sessions view also carries the subscriber's traffic. OmniTWAG merges the RADIUS accounting record into the session by `Acct-Session-Id`. A session is therefore the single source for both a subscriber's state and the subscriber's traffic. There is no separate accounting endpoint. Accounting is part of the session.

Table of Contents

- [Concepts](#)
- [Authentication status lifecycle](#)
- [Session model](#)
- [Authentication status reference](#)
- [REST API](#)
- [Per-session actions](#)
- [JSON examples](#)
- [Web interface](#)
- [Troubleshooting](#)

Concepts

Each authenticated subscriber has one session. OmniTWAG creates the session when a RADIUS Access-Request starts an EAP authentication. OmniTWAG updates the session as authentication progresses and as the data path is established.

A session tracks the subscriber identity, the access point, the client MAC, the allocated UE IP address, the EAP method and phase, and the GTP and DNS state. When RADIUS accounting arrives for the subscriber, OmniTWAG merges the traffic counters into the session.

The session shape is open-ended. The keys that are present depend on the auth method and on how far the session has progressed. A session that has only started EAP has fewer keys than a session that has reached RADIUS Accept and established a GTP and DHCP data path.

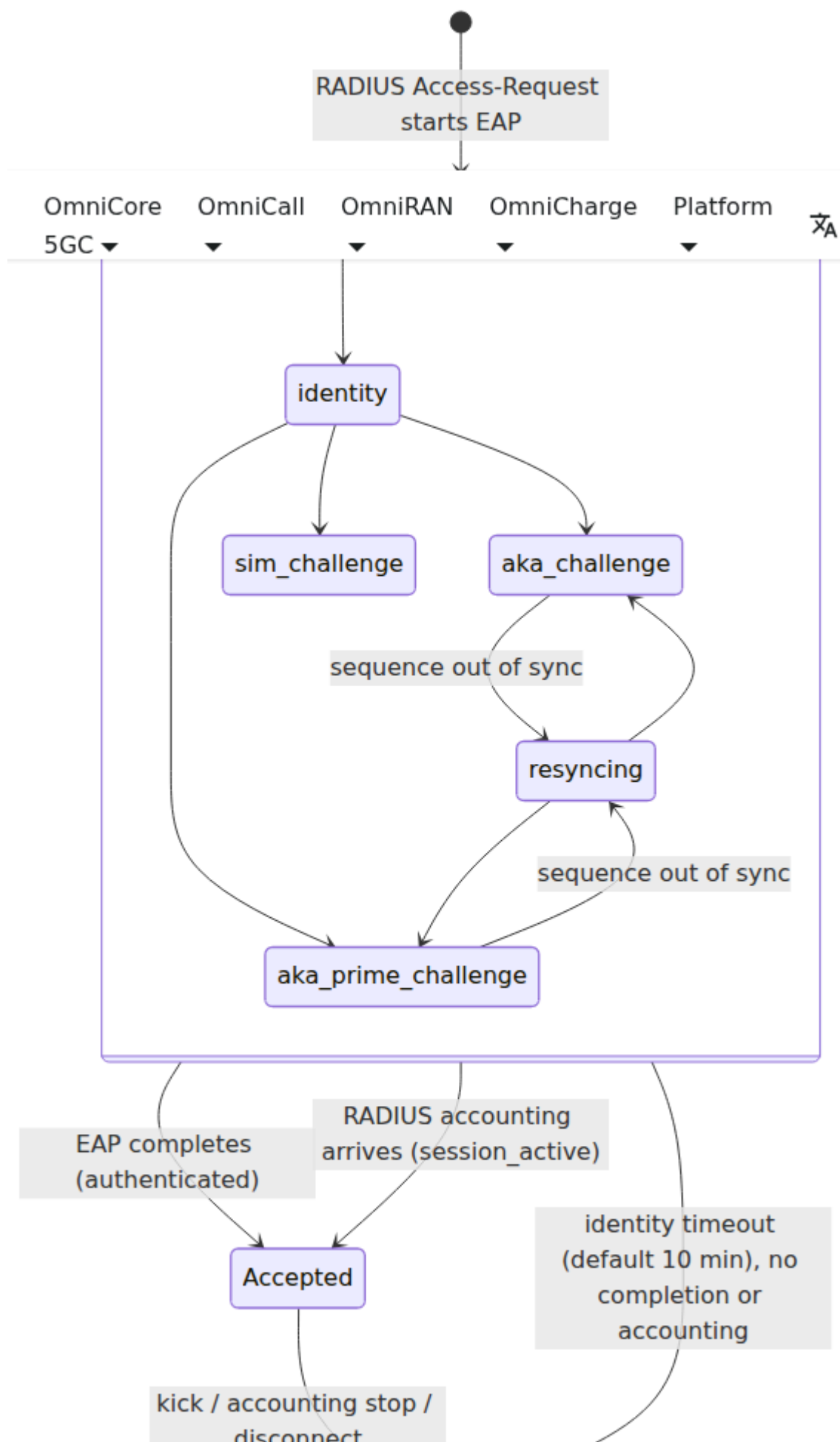
Authentication status lifecycle

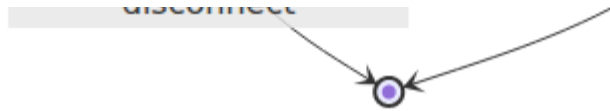
A session has an authentication status. The status is derived from the session's EAP phase (`eap_phase`).

- A session is **Authenticating** while `eap_phase` is an in-progress auth phase. The in-progress phases are `identity`, `aka_challenge`, `aka_prime_challenge`, `sim_challenge`, and `resyncing`.
- A session is **Accepted** once it is authenticated, or once it is active (`session_active`).

A session that receives RADIUS accounting is promoted to active. Accounting flows only for a subscriber that is already authenticated. The arrival of accounting is therefore proof that the subscriber is up, so OmniTWAG marks the session Accepted.

A session that stays in an in-progress auth phase is reaped. If a session started authentication, never completed, and never received accounting, OmniTWAG tears the session down after a timeout. This stops a half-finished auth attempt from lingering as Authenticating forever. The default timeout is 10 minutes.





Session model

A session has the fields below. The shape is open-ended, so a session may also carry other GTP or charging keys. Every field below is rendered as a printable string, an integer, or a list. Binary values are rendered as printable strings or as hex. IP tuples are rendered as dotted-quad strings.

Field	Type	Description
<code>identity</code>	String	Session identity. This is the IMSI or the username. OmniTWAG keys the session by this value.
<code>imsi</code>	String	Subscriber IMSI.
<code>mac_address</code>	String	UE MAC address.
<code>ue_ip</code>	String	Allocated UE IP address, as a dotted-quad string.
<code>nas_ip</code>	String	AP source IP (RADIUS <code>NAS-IP-Address</code>).
<code>called_station_id</code>	String	Access point identifier (RADIUS <code>Called-Station-Id</code>).
<code>calling_station_id</code>	String	Client MAC (RADIUS <code>Calling-Station-Id</code>).
<code>acct_session_id</code>	String	RADIUS accounting session id. OmniTWAG merges the accounting record by this key.
<code>eap_phase</code>	String	Current EAP phase. This value drives the authentication status. See Authentication status reference .
<code>eap_type</code>	String	EAP method in use, for example <code>aka</code> or <code>aka_prime</code> .
<code>pgw_ip</code>	String	PGW IP address, as a dotted-quad string.
<code>bearer_id</code>	Integer	EPS bearer id.

Field	Type	Description
<code>dns_servers</code>	List of String	DNS servers assigned to the UE.
<code>created_at</code>	Integer	Session creation time, in milliseconds since the Unix epoch.
<code>last_activity</code>	Integer	Last activity time, in milliseconds since the Unix epoch.
<code>established_at</code>	Integer	Session establishment time, in milliseconds since the Unix epoch.
<code>acct_input_octets</code>	Integer	Bytes received from the UE. Merged from RADIUS accounting.
<code>acct_output_octets</code>	Integer	Bytes sent to the UE. Merged from RADIUS accounting.
<code>acct_input_packets</code>	Integer	Packets received from the UE. Merged from RADIUS accounting.
<code>acct_output_packets</code>	Integer	Packets sent to the UE. Merged from RADIUS accounting.
<code>acct_session_time</code>	Integer	Accounting session duration, in seconds. Merged from RADIUS accounting.
<code>acct_status_type</code>	Integer	Latest accounting status. 1 is Start, 2 is Stop, 3 is Interim.
<code>acct_last_update</code>	Integer	Last accounting update time, in milliseconds since the Unix epoch.

The `acct_*` fields are present only after the subscriber's accounting record has been merged into the session.

Authentication status reference

OmniTWAG maps the `eap_phase` value to a displayed status. The table below lists the mapping.

<code>eap_phase</code>	Displayed status	Auth state
<code>identity</code>	Identity	Authenticating
<code>aka_challenge</code>	AKA Challenge	Authenticating
<code>aka_prime_challenge</code>	AKA' Challenge	Authenticating
<code>sim_challenge</code>	SIM Challenge	Authenticating
<code>resyncing</code>	Resyncing	Authenticating
<code>authenticated</code>	Accepted	Accepted
<code>session_active</code>	Accepted	Accepted
<code>accepted</code>	Accepted	Accepted
<code>rejected</code>	Rejected	Rejected

Any other `eap_phase` value counts as Authenticating.

REST API

The OmniTWAG REST API serves these endpoints over HTTPS on the configured API port (`8444` by default). The base path is `/sessions`.

Method	Path	Action	Description
GET	/sessions	index	List authenticated sessions. The response is paginated and searchable.
GET	/sessions/{id}	show	Get one session by <code>identity</code> .
DELETE	/sessions/{id}	delete	Remove the session and disconnect the subscriber.
PATCH	/sessions/{id}	action	Perform an action on the session. The action is in the JSON body.
PUT	/sessions/{id}	action	Perform an action on the session. The action is in the JSON body.

The `{id}` path parameter is the session `identity` (the IMSI or username).

Search and pagination

The index endpoint accepts pagination and search parameters.

Parameter	Type	Description
<code>offset</code>	Integer	Number of sessions to skip before the page.
<code>limit</code>	Integer	Maximum number of sessions in the page.
<code>q</code>	String	Search term. OmniTWAG matches the term against the search fields below.

The `q` term is matched against these session fields.

Search field	Matches
<code>identity</code>	Session identity (IMSI or username).
<code>imsi</code>	Subscriber IMSI.
<code>nas_ip</code>	AP source IP (<code>NAS-IP-Address</code>).
<code>called_station_id</code>	Access point identifier.
<code>calling_station_id</code>	Client MAC.
<code>mac_address</code>	UE MAC address.
<code>ue_ip</code>	Allocated UE IP address.

Action request body

`PATCH` and `PUT` accept a JSON body with one field.

Field	Type	Required	Description
<code>action</code>	String	Yes	The action to perform. Must be <code>kick</code> , <code>start_accounting</code> , or <code>stop_accounting</code> .

See [Per-session actions](#) for what each action does.

Responses

Status	Meaning
200 OK	The request was successful (index, show, delete, action).
404 Not Found	No session has the given identity.
422 Unprocessable Entity	The action is not one of kick, start_accounting, or stop_accounting.

A successful action returns a body with a `status` field and the `identity`. A delete returns `{"status": "deleted", "identity": ...}`.

Per-session actions

An action runs an administrative operation against a live session. Send the action in the JSON body of a `PATCH` or `PUT` request to `/sessions/{id}`.

Action	Effect	Result status
kick	Send a RADIUS CoA-Disconnect to the access point and tear the session down.	kicked
start_accounting	Mark the session active and begin accounting. This promotes the session to Accepted.	accounting_started
stop_accounting	End accounting, as if the access point sent Accounting-Stop. This tears the session down.	accounting_stopped

Both kick and stop_accounting tear the session down. Use kick to actively disconnect the subscriber at the access point. Use stop_accounting to end the session as the network would end it.

JSON examples

A session with traffic

Request:

```
GET /sessions/0010100000000001
```

Response:

```
{
  "identity": "0010100000000001",
  "imsi": "0010100000000001",
  "mac_address": "AA:BB:CC:DD:EE:FF",
  "ue_ip": "10.45.0.2",
  "nas_ip": "192.0.2.10",
  "called_station_id": "AA-BB-CC-DD-EE-FF:WiFi-Calling",
  "calling_station_id": "11-22-33-44-55-66",
  "acct_session_id": "5A2E1F0C",
  "eap_phase": "session_active",
  "eap_type": "aka_prime",
  "pgw_ip": "192.0.2.20",
  "bearer_id": 5,
  "dns_servers": ["8.8.8.8", "8.8.4.4"],
  "created_at": 1712000000000,
  "last_activity": 1712000600000,
  "established_at": 1712000004000,
  "acct_input_octets": 1048576,
  "acct_output_octets": 5242880,
  "acct_input_packets": 1200,
  "acct_output_packets": 3400,
  "acct_session_time": 600,
  "acct_status_type": 3,
  "acct_last_update": 1712000600000
}
```

A kick action

Request:

```
PATCH /sessions/0010100000000001
```

```
{
  "action": "kick"
}
```

Response (200 OK):

```
{
  "status": "kicked",
  "identity": "0010100000000001"
}
```

An unknown action

Request:

```
{
  "action": "reboot"
}
```

Response (422 Unprocessable Entity):

```
{
  "error": "Unknown action \"reboot\" (expected
kick|start_accounting|stop_accounting)",
  "identity": "0010100000000001"
}
```

Web interface

The control panel has a **RADIUS Clients** page. The page lists every session with its authentication status. The page shows the count of Accepted, Authenticating, and Rejected sessions. Use the page to review live sessions at a glance.

Troubleshooting

A session is stuck "Authenticating"

Symptoms: A session shows the Authenticating status. The status does not change to Accepted. The session stays in an EAP phase such as `identity`,

`aka_challenge`, `aka_prime_challenge`, `sim_challenge`, or `resyncing`.

Possible causes:

- The subscriber started EAP authentication but never completed it.
- The access point stopped sending EAP responses.
- The subscriber failed authentication and did not retry.

How it clears:

1. OmniTWAG arms a timeout when the session enters an in-progress auth phase.
2. If the session never completes authentication and never receives accounting, OmniTWAG tears the session down when the timeout fires. The default timeout is 10 minutes.
3. A session that completes authentication moves to Accepted. A session that receives RADIUS accounting is promoted to active and also moves to Accepted.

To remove a stuck session at once, send a `kick` action or a `DELETE` request to `/sessions/{id}`.

A session does not show traffic

Symptoms: A session is Accepted, but the `acct_*` fields are absent or zero.

Possible causes:

- No RADIUS accounting record has been received for the subscriber yet.
- The accounting record has a different `Acct-Session-Id` than the session.

Resolution:

1. Confirm that the access point sends RADIUS accounting to OmniTWAG.
2. Confirm that the accounting `Acct-Session-Id` matches the session `acct_session_id`. OmniTWAG merges traffic by this key.
3. Wait for the next accounting interim update, then read the session again.

Overview

User Plane and Data Path

Overview

OmniTWAG separates the control plane from the user plane. The control plane runs the authentication and the session signalling. The user plane carries the real UE traffic. This page describes the user plane. It explains how OmniTWAG moves UE packets, how it builds and tears down the GTP-U tunnel, and how it forwards downlink packets back to the UE.

For the GTP-C session signalling that sets up the tunneled data path, see [S2a / GTP-C](#). For the full system view see [Architecture](#).

Table of Contents

- [Control Plane and User Plane](#)
- [Data Path Modes](#)
- [GRE Data Path](#)
- [GTP-U Tunnelling](#)
- [Downlink Forwarding](#)
- [Bearers and Traffic Flow Templates](#)
- [Error Handling](#)
- [MTU and Fragmentation](#)
- [Configuration](#)
- [Observability](#)

Control Plane and User Plane

The control plane and the user plane run as separate processes.

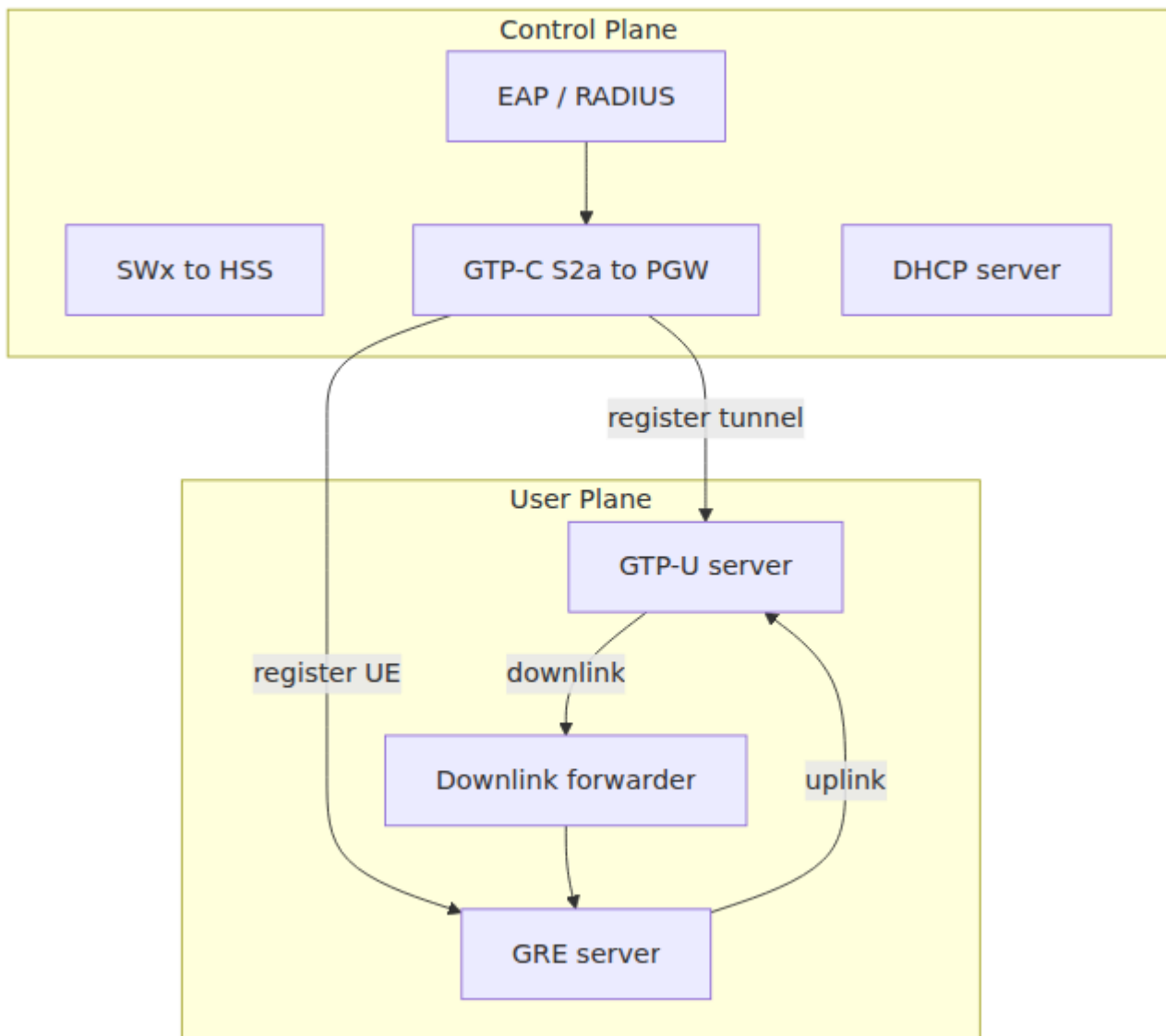
The **control plane** does the following work:

- EAP authentication over RADIUS with the access point.
- Auth vector fetch from the HSS over Diameter SWx.
- GTP-C session signalling to the PGW on the S2a interface. This creates and deletes the PDN connection and the bearers. See [S2a / GTP-C](#).
- DHCP address delivery to the UE. See [IP Address Management](#).

The **user plane** does the following work:

- GRE tunnelling to the access point.
- GTP-U tunnelling to the PGW on the S2a-U interface, per [3GPP TS 29.281](#) (tunneled mode only).
- Downlink packet forwarding toward the UE.
- Uplink packet classification into bearers.

The control plane sets up the user-plane state. After authentication, the session process creates the GTP-C session (in tunneled mode), registers the GTP-U tunnel, and registers the UE with the GRE server. Then the user plane carries the traffic.



Data Path Modes

OmniTWAG resolves a data-plane mode per access point (AP) from the RADIUS credential that matches the AP. The `breakout_type` value on the credential selects one of three modes. See [RADIUS Credentials](#) for the `breakout_type`, `address_source`, and `ip_pool_id` settings.

Mode (breakout_type)	UE IP assigned by	TWAG carries traffic	Path
tunneled	PGW	Yes	GRE from AP to TWAG, then GTP-U on S2a to the PGW.
nswo	TWAG, from an IP pool	Yes	GRE from AP to TWAG, then local breakout to the internet (NAT) at the TWAG.
local	Local network at the AP	No	The AP owns the data plane. The TWAG carries no packets and assigns no IP.

The GRE data path applies to both `tunneled` and `nswo` breakout. In both modes the AP sends UE frames to the TWAG in GRE, and the TWAG sends downlink packets back to the AP in GRE. The difference is the uplink after GRE:

- **tunneled**: the TWAG encapsulates the uplink packet in GTP-U and sends it to the PGW. The PGW assigns the UE IP and applies policy and charging.
- **nswo**: the TWAG breaks the uplink out locally to the internet with a NAT masquerade rule. The TWAG assigns the UE IP from an IP pool. There is no PGW or GTP-C session. The TWAG-side Gy **Online Charging** applies.

In `local` mode the TWAG neither assigns an IP nor carries traffic, so neither the GRE nor the GTP-U path is used for that AP.

Tunneled Mode (GTP-U to PGW)

In tunneled mode, OmniTWAG tunnels the UE traffic to the PGW. The PGW allocates the UE IP address, applies the policy and the charging, and provides internet access. This is the S2a interface per **3GPP TS 23.402** Section 16.

The uplink path is:

```
UE --WiFi--> AP --GRE--> TWAG --GTP-U S2a--> PGW --> Internet
```

The downlink path is the reverse:

```
Internet --> PGW --GTP-U S2a--> TWAG --GRE--> AP --WiFi--> UE
```

To use this mode, set `gtp.enabled` to `true` and `gre.enabled` to `true`, and set the AP credential `breakout_type` to `tunneled`. See [Configuration](#).

Local Breakout (NSWO)

Non-Seamless WLAN Offload (NSWO) is local breakout. The UE traffic exits directly from the TWAG to the internet. The traffic does not go to the PGW. The TWAG assigns the UE IP and applies NAT masquerade. This is per [3GPP TS 23.402](#) Section 16.2.

OmniTWAG has two ways to reach NSWO:

1. **Per-AP breakout (preferred)**: set the AP credential `breakout_type` to `nsw`. The UE address comes from the [IP pool](#) named by `ip_pool_id`. The credential's `address_source` selects how:
 - `pool` pre-allocates a fixed address from the pool and registers the UE for GRE downlink at once.
 - `dhcp` binds the UE MAC to the pool and lets the TWAG DHCP server dish out an address on the DHCP DISCOVER. The GRE downlink is registered when the address is known.
2. **Global NSWO handler**: the `Twag.NSW0.Handler` provides local breakout for every session when GTP is not used. It is selected with `nsw.enabled: true` (and applies when `gtp.enabled: false`). The handler manages a single local IP pool and configures an `iptables` NAT masquerade rule. On session establishment the TWAG allocates a local-pool address, delivers it to the UE over DHCP, and registers the UE for GRE downlink, with no PGW / GTP-C session. The allocation is released on session teardown.

For both paths the uplink is NAT-masqueraded to the internet and the downlink returns through the GRE path. See [IP Address Management](#) and [IP Pools](#) for the addressing.

The global NSWOW handler reads these parameters from the `nswow` map:

Parameter	Type	Default	Description
<code>enabled</code>	Boolean	<code>false</code>	Start the global NSWOW handler and route sessions via local breakout.
<code>local_pool</code>	String (CIDR)	<code>10.100.0.0/16</code>	Pool the UE address is allocated from.
<code>nat_interface</code>	String	<code>eth0</code>	Egress interface for the NAT masquerade rule.
<code>dns_servers</code>	List	<code>["8.8.8.8", "8.8.4.4"]</code>	DNS servers advertised to the UE. Local breakout has no PGW PCO to carry them.

GRE Data Path

The GRE server (`Twag.Gre.Server`) is the user-plane endpoint toward the access point. The access point sends the UE frames to the TWAG in GRE, per [RFC 2784](#) and [RFC 2890](#). The GRE server also sends the downlink packets and the DHCP responses back to the access point. GRE is the AP-facing path for both `tunneled` and `nswow` breakout.

The GRE server uses a raw socket for IP protocol 47 (GRE), opened with `:socket.open/3`. The host must grant the `CAP_NET_RAW` capability. If the socket does not open, the server logs an error and retries every 10 seconds. GRE is off by default. To turn it on, set `gre.enabled` to `true`.

Inbound GRE Classification

The GRE server reads the outer IP header, strips it, and decodes the GRE header. The server supports two GRE payload types:

GRE protocol	Meaning	Handling
0x6558	Transparent Ethernet Bridging	Extract the inner Ethernet frame, then classify by ethertype.
0x0800	IPv4	Treat as an uplink IPv4 packet.

For a Transparent Ethernet Bridging frame, the server reads the inner ethertype:

Ethertype	Meaning	Handling
0x0800	IPv4	If the packet is DHCP, pass it to the DHCP server. If not, forward it to the uplink path.
0x0806	ARP	Handle locally. Do not forward.
0x86DD	IPv6	Forward to the uplink path.

The server keeps a map of the UE MAC address to the access-point IP. The server learns this map from the source MAC of each inbound frame. This map lets the server route the downlink frames back to the correct access point.

Uplink Handling

For an uplink IP packet, the GRE server reads the source IP. This is the UE IP. The server looks up the UE IP in the registered-UE table to find the session identity. It finds the session process in the registry and reads the session local TEID. It then sends the packet asynchronously to the GTP-U server with that local TEID (in tunneled mode). The asynchronous send avoids a deadlock between the GRE and GTP-U processes.

If there is no session for the UE IP, or the session has no GTP-U tunnel yet, the server drops the packet. A runt IP packet (less than 20 bytes) is also dropped.

Note: For `nswo` breakout the local IP pool and the NAT masquerade carry the uplink to the internet. The GRE server registers the UE for downlink routing in the same way as tunneled mode.

UE Registration for Downlink

When the session becomes active, the session process registers the UE with the GRE server. The registration holds the UE IP, the UE MAC, the access-point IP, and the session identity. The GRE server uses this data to route the downlink packets. On session teardown, the session process unregisters the UE.

GTP-U Tunnelling

The GTP-U server (`Twag.Gtp.GtpU`) is the user-plane endpoint toward the PGW in tunneled mode. It runs a single UDP socket on port `2152` and holds a table of tunnels. The server implements GTPv1-U per [3GPP TS 29.281](#).

TEID Handling

Each tunnel has a local TEID and a remote TEID. The remote TEID is the PGW TEID. The TWAG uses the remote TEID in the header of the packets that it sends to the PGW. The PGW uses the local TEID in the header of the packets that it sends to the TWAG. The TWAG looks up the local TEID in the tunnel table to find the tunnel.

When the session process gets the Create Session Response from the PGW, it registers the tunnel. The GTP-U server generates the local TEID. The TWAG uses a cryptographically random 32-bit value for the local TEID, and it never uses zero. This follows the non-predictable TEID requirement in [3GPP TS 29.274](#).

A registered tunnel holds these fields:

Field	Description
<code>local_teid</code>	The TWAG-side TEID. The PGW sends downlink packets with this value.
<code>remote_teid</code>	The PGW-side TEID. The TWAG sends uplink packets with this value.
<code>remote_ip</code>	The PGW GTP-U IP address.
<code>remote_port</code>	The PGW GTP-U UDP port. Default <code>2152</code> .
<code>session_id</code>	The session identity for the tunnel.

Uplink Encapsulation

For each uplink UE packet, the GTP-U server builds a GTPv1-U G-PDU header and puts the header in front of the inner IP packet. The header carries the GTP version (1), the protocol type (GTP), the message type (G-PDU, 255), the payload length, and the remote TEID. The server sends the packet to the PGW IP and port over the UDP socket. The GRE server calls the GTP-U server on the asynchronous send path, which avoids a deadlock between the two processes.

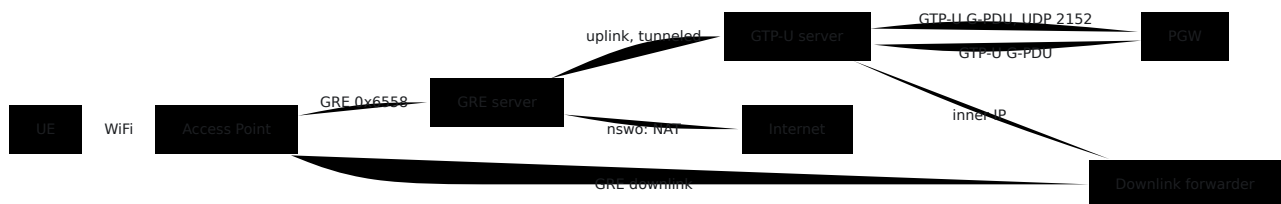
Downlink Decapsulation

For each packet that arrives on the GTP-U socket, the server reads the GTP-U header. The server acts on the message type:

Message	Type	Action
G-PDU	255	Look up the tunnel by the local TEID, then forward the inner IP packet to the downlink forwarder.
Echo Request	1	Reply with an Echo Response.
Echo Response	2	Accept and ignore.
Error Indication	26	Accept and ignore.
End Marker	254	Accept and ignore.

For a G-PDU, the server strips the GTP-U header and any optional header fields (sequence number, N-PDU number, and next-extension-header type). Then the server passes the inner IP packet to `Twag.Downlink`. See [Downlink Forwarding](#).

If a G-PDU arrives for a TEID that has no tunnel, the server drops the packet and sends a GTP-U Error Indication back to the source. See [Error Handling](#).



Downlink Forwarding

The downlink forwarder (`Twag.Downlink`) receives the inner IP packets from the GTP-U decapsulation. The forwarder sends each packet toward the UE. The forwarder has two paths and prefers GRE.

GRE Path (Preferred)

If the GRE server runs, the forwarder sends the packet through GRE. The forwarder reads the destination IP of the inner packet. This is the UE IP. The forwarder calls the GRE server. The GRE server looks up the UE IP in the registered-UE table, wraps the packet in a Transparent Ethernet Bridging GRE frame, and sends it to the access-point IP. The access point delivers the frame to the UE over WiFi.

For the downlink Ethernet frame, the GRE server uses the UE MAC as the destination MAC. The server uses a fixed locally-administered MAC (`02:00:00:00:00:01`) as the source MAC.

Raw Socket Path (Fallback)

If the GRE server does not run, or the GRE send fails, the forwarder uses a raw socket. The forwarder injects the raw IP packet. The kernel then routes the packet to the UE from the local routing table.

The forwarder opens two raw sockets: one for IPv4 and one for IPv6. The forwarder sets the `IP_HDRINCL` option (and `IPV6_HDRINCL` for IPv6). This option means the forwarder supplies the full IP header. On some kernels the socket still works without this option, so a failure to set the option is not fatal.

The host must grant the `CAP_NET_RAW` capability for the raw sockets. If a socket does not open, the forwarder logs a warning and disables that address family. If neither GRE nor a raw socket is available, the forwarder counts and drops the packets, and logs a warning one time.

Path Selection

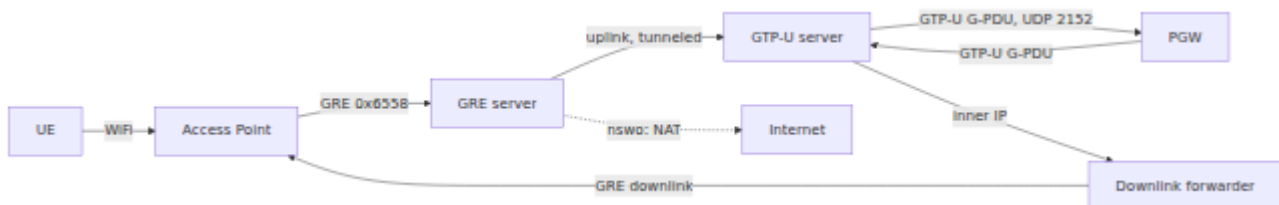
The `prefer_gre` parameter sets the path selection order.

- `prefer_gre: true` (default): the forwarder tries the GRE tunnel first and falls back to the raw socket if GRE is unavailable or the GRE send fails.
- `prefer_gre: false`: the forwarder tries the raw socket for the packet address family first and uses GRE only when that raw socket is unavailable.

Packet Checks

The forwarder drops a packet in these cases:

- The forwarder is disabled by configuration.
- No socket is available, and GRE is not available.
- The packet is a runt (less than 20 bytes).
- The forwarder cannot read the destination IP.



Bearers and Traffic Flow Templates

The bearer manager (`Twag.Bearer.Manager`) holds the bearers for a session. Each bearer maps to an EPS Bearer Identity (EBI). A bearer holds the GTP-U TEIDs, the Traffic Flow Template (TFT) packet filters, and the QoS class (QCI). One bearer is the default bearer. This follows [3GPP TS 23.402](#) Section 16.1.6.1.

Default and Dedicated Bearers

Bearer type	TFT filters	Purpose
Default	None	Carries all traffic that no dedicated bearer matches.
Dedicated	One or more	Carries the traffic that matches its TFT packet filters.

If the default bearer is removed, the bearer manager signals a full session teardown. The removal of a dedicated bearer removes only that bearer.

TFT Packet Filters

A TFT is a Traffic Flow Template per [3GPP TS 24.008](#) Section 10.5.6.12. A TFT holds an operation and a list of packet filters. Each packet filter holds a direction, a precedence, and a list of components. The TFT parser supports these component types:

Component	Match
IPv4 remote address and mask	The packet destination address.
IPv4 local address and mask	The packet source address.
IPv6 remote/local address and prefix	The IPv6 address.
Protocol identifier	The IP protocol number.
Single local port / local port range	The UE source port.
Single remote port / remote port range	The destination port.
Type of service / traffic class	The TOS byte, with a mask.
Security parameter index	Not matched in the TWAN uplink.
Flow label	Not matched (IPv6 only).

The parser supports the TFT operations: create, delete, add filters, replace filters, delete filters, and no operation. The bearer manager applies these operations when the PGW sends an Update Bearer Request.

Uplink Classification

The uplink classifier binds each uplink packet to a bearer. The rules are:

1. Read the source IP, the destination IP, the protocol, the ports, and the TOS from the packet.

2. Test the packet against the TFT filters of the dedicated bearers.
3. A filter matches only if all its components match. This is AND logic.
4. If more than one filter matches, select the filter with the lowest precedence number. A lower precedence number is a higher priority.
5. If no dedicated bearer matches, select the default bearer.

An empty filter (a filter with no components) matches all packets. This is a wildcard.

Note: The bearer manager and the TFT classifier are present for dedicated bearer support. The uplink GRE path forwards the traffic on the session local TEID. A full multi-bearer uplink split depends on the PGW to create dedicated bearers for the session, which the TWAG rejects under TSCM (see [S2a / GTP-C](#)).

Error Handling

GTP-U Echo

The PGW and the TWAG use the GTP-U Echo Request and Echo Response to test the tunnel path. When the GTP-U server receives an Echo Request, it replies with an Echo Response. The GTP-C S2a process runs its own echo when `gtp.echo_enabled` is `true`. See [S2a / GTP-C](#).

GTP-U Error Indication

If a G-PDU arrives for a TEID that has no tunnel, the GTP-U server sends a GTP-U Error Indication back to the source. The Error Indication carries the unknown TEID and the TWAG GTP-U peer address. This tells the PGW to stop sending traffic on the dead tunnel. The server also drops the packet and increases the error count.

GTP-U End Marker

The GTP-U server can send an End Marker on a tunnel before it tears the tunnel down. This is for a handover. The End Marker tells the peer that no more

packets will follow on the old path.

Decode Errors

If the GTP-U server cannot decode a packet, it logs a warning and increases the error count. If the GRE server cannot decode a packet, or gets an unknown protocol or ethertype, it counts the event and drops the packet.

MTU and Fragmentation

The GRE server builds the outer IPv4 header for each downlink GRE packet. The GRE protocol number is 47.

The GTP-U and GRE encapsulation each add header bytes to the UE packet. To avoid fragmentation, the access point should reduce the TCP MSS and the tunnel MTU. The reference Cisco access-point configuration sets `mss 1410` and `mtu 1500` on the GRE tunnel. See [Architecture](#) for the access-point configuration.

Configuration

The user-plane processes read the `:omnitwag` application configuration. The relevant blocks are `gtp`, `gre`, and `downlink`. The AP breakout mode comes from the RADIUS credential, not from these blocks. See [RADIUS Credentials](#).

```
config :omnitwag,  
  # GTP-U and GTP-C toward the PGW (S2a) - tunneled mode  
  gtp: %{  
    enabled: false,  
    pgw_ip: "10.5.198.1",  
    pgw_port: 2123,  
    local_ip: "10.5.198.200",  
    gtpu_port: 2152,  
    echo_enabled: true  
  },  
  
  # GRE toward the access point  
  gre: %{  
    enabled: false,  
    local_ip: "10.5.198.200"  
  },  
  
  # Downlink forwarding  
  downlink: %{  
    enabled: true,  
    ip_hdr_incl: true,  
    prefer_gre: true  
  }
```

GTP Parameters

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>false</code>	Start the GTP-C and GTP-U servers. Set <code>true</code> for tunneled mode.
<code>pgw_ip</code>	String	No	<code>"10.5.198.1"</code>	The PGW IP address for GTP-C and GTP-U.
<code>pgw_port</code>	Integer	No	<code>2123</code>	The PGW GTP-C UDP port.
<code>local_ip</code>	String	No	Auto-detected	The TWAG local IP for GTP. Used as the source for GTP-U packets and for the Error Indication peer address.
<code>gtpu_port</code>	Integer	No	<code>2152</code>	The local UDP port for the GTP-U server. This is the standard GTP-U port.
<code>echo_enabled</code>	Boolean	No	<code>true</code>	Run the GTP-C echo toward the PGW. See S2a / GTP-C .

GRE Parameters

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>false</code>	Start the GRE server. Set <code>true</code> to receive UE frames from the access point over GRE. Needed for both <code>tunneled</code> and <code>nsw</code> breakout.
<code>local_ip</code>	String	No	<code>"0.0.0.0"</code>	The local IP to bind the GRE raw socket. This is the destination IP that the access point sends the GRE traffic to.

Downlink Parameters

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	No	<code>true</code>	Start the downlink forwarder. When <code>false</code> , the forwarder drops all downlink packets.
<code>ip_hdr_incl</code>	Boolean	No	<code>true</code>	Set the <code>IP_HDRINCL</code> and <code>IPV6_HDRINCL</code> options on the raw sockets so the forwarder supplies the full IP header. When <code>false</code> , the option is not set. Leave <code>true</code> unless the kernel rejects <code>IP_HDRINCL</code> .
<code>prefer_gre</code>	Boolean	No	<code>true</code>	Path selection order. When <code>true</code> , the forwarder tries GRE first and falls back to the raw socket. When <code>false</code> , it tries the raw socket first and uses GRE only when the raw socket is unavailable.

Host Capability

The GRE server and the downlink raw-socket path both use raw sockets. The host must grant the `CAP_NET_RAW` capability. Without this capability, the raw sockets do not open. The GRE server retries the socket every 10 seconds. The downlink forwarder falls back or drops packets, as described in [Downlink Forwarding](#).

Observability

The user-plane processes keep internal statistics. Each process reports its own status, covering the GTP-U tunnel endpoint, the GRE server, and the downlink forwarder. For the Prometheus metrics that OmniTWAG exports, see the [Metrics Reference](#).

GTP-U Status

The GTP-U status reports `socket_open`, `local_ip`, `tunnel_count`, and a nested `stats` map:

Statistic	Description
<code>packets_in</code>	The count of packets received on the GTP-U socket.
<code>packets_out</code>	The count of packets sent on the GTP-U socket.
<code>bytes_in</code>	The total bytes received.
<code>bytes_out</code>	The total bytes sent.
<code>errors</code>	The count of decode errors and unknown-tunnel events.
<code>tunnel_count</code>	The number of registered tunnels (top-level field).

GRE Status

The GRE status reports `running`, `socket_open`, `local_ip`, `ap_tunnels`, `registered_ues`, and a nested `stats` map:

Statistic	Description
packets_in	The count of GRE packets received.
packets_out	The count of GRE packets sent.
bytes_in / bytes_out	The total bytes in and out.
dhcp_in	The count of DHCP packets received from the access point.
uplink_forwarded	The count of uplink packets forwarded to GTP-U.
downlink_forwarded	The count of downlink packets sent to the access point.
unknown_protocol	The count of packets with an unknown protocol or ethertype.
errors	The count of decode and send errors.
registered_ues	The number of registered UEs (top-level field).

Downlink Statistics

Statistic	Description
<code>forwarded</code>	The count of IPv4 packets forwarded (GRE and IPv4 raw socket).
<code>forwarded_v6</code>	The count of IPv6 packets forwarded.
<code>dropped</code>	The count of dropped packets.
<code>errors</code>	The count of send errors.
<code>bytes</code>	The total forwarded bytes.
<code>enabled</code>	The forwarder enabled state.
<code>socket_open</code>	The raw IPv4 socket state.

Overview

