AUTHCHA LEARN

Integrations

pfSense, OpenVPN, web policy, delivery channels for second factors, enforcement connectors and the API.

6 modules · about 180 minutes · Integrators and automation engineers

1. Web policies
2. The pfSense agent
3. OpenVPN users through RADIUS on pfSense
4. Delivery channels for any second factor
5. Enforcement connectors
6. The API and automation

Quizzes, the final assessment and live links are online at learn.authcha.cloud. Labs use the live demo at demo.authcha.cloud.

Module 1 of 6 · 30 minutes

Web policies

By the end you can

- State that Authcha decides web policy but does not block sites itself
- Write allow-list and deny-list policies with domains, wildcards and categories
- Assign a policy and predict which one wins
- Explain why an HTTPS block shows an error, and when decrypting is appropriate

Decide here, enforce elsewhere

A **web policy** says which websites a person may reach: "only `google.co.in`", or "everything except social networks". You attach it to a user, a directory group or a user type, and Authcha tells the enforcement point which policy each signed-in person has.

Authcha does not block websites itself. It decides at the RADIUS layer and **publishes** the decision. A separate enforcement point turns the decision into rules, and which one you choose decides how exactly the policy is held.

Enforcement point	Sees	Holds a user to one site?
Squid with the Authcha helper (peek-and-splice)	The TLS server name and the Host header	**Yes**, even when many sites share an address
A next-generation firewall with FQDN or URL objects	The name, if configured	Yes, with identity delivered by a connector
The pfSense agent, Squid path	The name	Yes
The pfSense alias path, or the alias connector	**Only the address**	**No, approximate**

Web policies are part of the NAC edition.

An address is not a website

One address serves many sites on a content delivery network, and a site's addresses change. A policy saying "only `google.co.in`" cannot be held by an address list. Use **Squid** whenever a person must be held to named sites, and use address-based paths only for what an address *can* express, such as rate limits or blocking ports.

Write a policy

Open **Access Control** → **Web policies**.

Field	Meaning
Name	1 to 48 characters: lower-case letters, digits, `-` and `_`. This is the name the device receives
Mode	**Allow-list**: the entries are the only destinations. **Deny-list**: the entries are blocked and everything else is allowed
Domains	Up to 500 entries
Categories	Up to 20 built-in groups

A policy needs at least one domain or category.

Entries

Write	Matches
`google.co.in`	That name only
`*.google.co.in`	Every subdomain, **not** `google.co.in` itself. Write both to cover both
`ip:203.0.113.7`	One address
`cidr:203.0.113.0/24`	A network
`ext:zip`	A file type, by the end of the URL path (deny-lists only)

Authcha checks every entry when you save. Names are lower-cased and converted to punycode, and a bare IP address as a domain is refused (say `ip:`). Only a leading

`*.` is a wildcard. URLs with paths, single labels such as `com` and over-long names are refused.

Categories

Built-in lists of base domains, expanded when the policy is read, so a change to the built-in list reaches existing policies at the next poll.

Category	Contains
`search`	Google, Bing, DuckDuckGo
`social`	Facebook, Instagram, X, TikTok, LinkedIn, Reddit, Snapchat, WhatsApp
`video-streaming`	YouTube and its media hosts, Netflix, Prime Video, Hotstar, Twitch
`ads-trackers`	Common advertising and tracking networks
`file-sharing`	Dropbox, Box, Mega, WeTransfer, OneDrive, Google Drive
`software-updates`	Windows Update, Ubuntu, Debian, Apple updates
`file-download`, `media-downloads`, `p2p-torrent`	**File types**, not domains

The map is deliberately small. It is **not** a content-filtering database.

> **A page is more than one host name**
>
> A policy "only Google India" must also allow `gstatic.com` and similar hosts the page loads, or it renders broken. List what your pages need.

Who gets which policy

Assignments attach a policy to a **user**, a **directory group** or a **user type**. A subject has at most one. When several apply, the most specific wins:

1. the **user**

2. a **directory group** the user belongs to (the lowest priority number wins if several); membership is the last synced one, so a directory that stops syncing leaves the restriction in place instead of lifting it
3. the **user type**

Someone with **no policy is unrestricted**. There is no "unrestricted" assignment: to exempt a person, take them out of the group or attach a policy to a narrower subject.

What a device learns

Every Access-Accept for a user with a policy carries the policy **name** in the `Authcha-Web-Policy` attribute. It is informational and never changes whether the user is accepted. MAB logins never get it. Enforcement points read the full feed over the API (`/api/web-policy/active-sessions` and `/api/web-policy/policies`), using a token of kind **Web policy agent** that can call exactly those requests and nothing else.

A policy change reaches a polling enforcement point at its next poll, with no re-authentication. You can also turn on **Re-authorize on policy change** so the person's session ends and they pick up the new policy at once. It is off by default.

The block page, and HTTPS

With the pfSense Squid agent, a blocked plain-HTTP site shows a styled "This page is blocked" page with the host name and a Go back button, instead of Squid's grey text.

A blocked HTTPS site cannot show any page. The browser asks for the site's certificate before sending anything, so Squid can only show a page if it decrypts that connection with a CA the device trusts. Without that, the browser shows a connection error, and that is how HTTPS works without decrypting it. The cleanest setup, for devices **you** own, decrypts **only the sites you block**, leaving banking and pinned apps alone. For guests and personal devices you cannot install a CA on, **do not decrypt**.

Lab HANDS-ON

On the demo as `demo-admin` (password `Authcha-Demo-2026`).

1. Open Web Policy (https://demo.authcha.cloud/nac/web-policy). Find `no-social-networks`, `no-video-streaming` and `no-downloads`. Open one and read its mode, categories and entries.

2. Look at the **Assignments**. For a user of type `network-user`, which policy applies, and why?

3. Make a test policy `google-only` as an allow-list with `google.co.in`, `*.google.co.in` and `gstatic.com`. Use the resolver to see what a user resolves to.

4. Assign it to a test user, then to their user type as well, and confirm the user's own assignment wins.

5. On your own install, read the feed with a Web policy agent token: `curl -s -H "Authorization: Bearer <token>" https://<host>/api/web-policy/policies`.

6. Add `ext:zip` to a deny-list and try it on an allow-list. Read the refusal.

Cheat sheet KEEP THIS

- **Authcha decides; Squid, a firewall or an agent enforces.**
- Modes: **allow-list** (only these) or **deny-list** (all but these). Entries: `name`, `*.name`, `ip:`, `cidr:`, `ext:` (deny only).
- Categories: `search`, `social`, `video-streaming`, `ads-trackers`, `file-sharing`, `software-updates`, and file types.
- Winner: **user** → **directory group** → **user type**. No policy = unrestricted.
- Address-based = approximate. **Squid = exact.**
- Blocked HTTPS = connection error unless Squid decrypts with a CA the device trusts. Decrypt only your own devices, only blocked sites.

- Read more: Web policies (https://demo.authcha.cloud/docs/web-policy).

Module 2 of 6 · 30 minutes

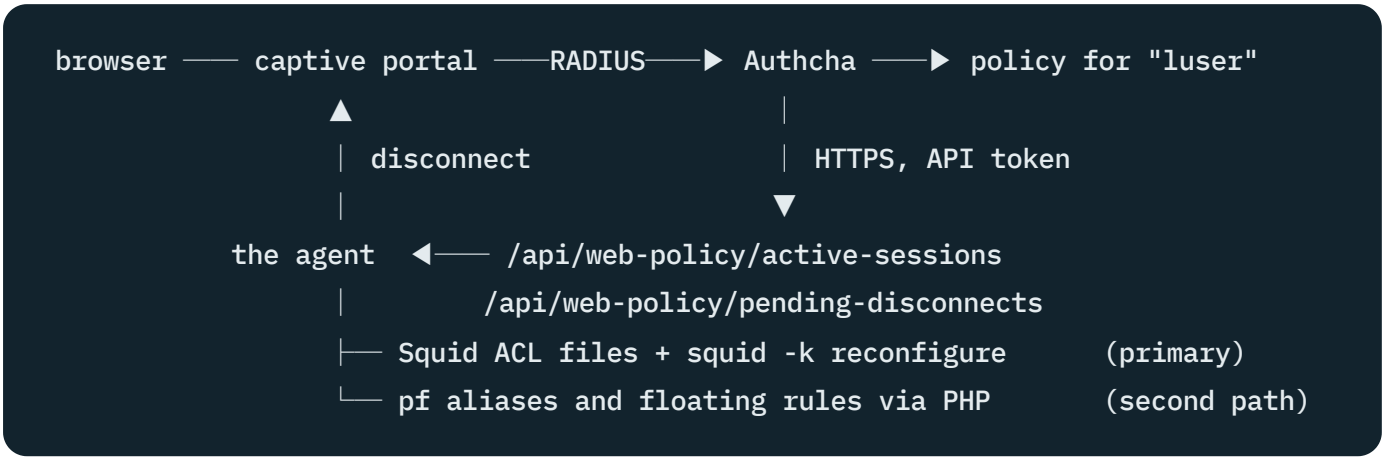
The pfSense agent

By the end you can

- Explain why pfSense CE needs an agent
- Install and run the agent in dry-run, then for real, with the right token and certificate
- Choose between the Squid path and the address-based path
- Run the agent as a service and verify it with --check

Two facts that shape everything

- **pfSense CE cannot enforce a per-user URL list by itself.** Authcha decides which policy a person has, and an **agent on the firewall** turns that into Squid rules or firewall rules.
- **pfSense CE has no RFC 5176 listener.** Authcha cannot send it a Disconnect-Request. The agent ends sessions through the portal's own disconnect instead.



The agent comes as a Python script (standard library only), a single static **FreeBSD binary** (Go, about 6 MB, no Python needed), or a static **Linux binary** (Rust) for a Squid host that is not pfSense. They take the same options and write the same files. It was written for pfSense 2.7 and verified in a lab on 2.9.0; the binaries have been tested on Linux against the agent suite, but the Go binary has **not** been run on a real pfSense yet.

> **Run it in dry-run first**
>
> The places where the agent depends on pfSense internals are marked in the guide. Always run `--dry-run` on your own firewall before letting it change anything.

Pick the path

	Squid ACLs (primary)	Aliases and floating rules (second)
Enforces	Source address and destination **name**	Source address and destination **IPs**
Accuracy	Good for names the browser asks for	Approximate: CDNs and shared hosting move
Needs	Squid in transparent mode, plus SSL filtering in **splice** mode for HTTPS	Nothing beyond pfSense
Choose it when	A person must be held to named sites	You cannot run Squid and "mostly right" is acceptable

Use `--mode squid` (the default), `pf` or `both`.

Set it up

1. **A token. Settings → API tokens → New**, kind **Web policy agent**. Copy it once. It can read the enforcement feed and nothing else. Save it on the firewall as `/usr/local/authcha/token` (mode 0600).
2. **The console certificate.** The agent always verifies it. Copy the CA that signed it to `/usr/local/authcha/ca.pem`, or use a certificate your firewall already trusts. Installing your own console certificate avoids certificate-change surprises.
3. **Install.** Copy the agent and its two PHP helpers to `/usr/local/authcha/`. (The binary needs no Python.)
4. **Squid (primary path).** Install the Squid package, enable it as a **transparent proxy** on the captive-portal interface and, for HTTPS, turn on **SSL filtering** with **Splice All**, so Squid reads the TLS server name without decrypting. Under **Custom Options (Before Auth)** add `include`

`/usr/local/etc/squid/authcha/webpolicy.conf`. It must come **before** the rule that allows your LAN.

5. **Captive-portal networks: add `--require-session`.** A transparent Squid fetches HTTPS sites for any client redirected to it, and a client that has not signed in would get through, because it has no restricted session and so no policy. With the option, the agent also refuses every address with no live portal session.
6. **Try it without changing anything:**

```
python3 /usr/local/authcha/authcha_webpolicy_agent.py \ --url https://authcha.example --token-file /usr/local/authcha/token \ --ca-file /usr/local/authcha/ca.pem --dry-run
```

1. **Run it continuously**, with `--interval 20`, a cron job, or the service script below.

Check, then run as a service

`--check` tests the setup and **changes nothing**: the console answers and accepts the token (TLS included), the Squid and PHP programs exist, the portal zone can be found. It prints `ok` or `FAIL` per line and exits 0 when all is fine. Run it first and after every change.

`integrations/pfsense/rc.d/authcha_webpolicy.sh` runs the agent under FreeBSD's `daemon(8)`, restarts it five seconds after it dies, and starts it at boot. It reads `/usr/local/etc/authcha-webpolicy.conf` for the URL, token file, CA file, `AUTHCHA_REQUIRE_SESSION`, the portal URL and `AGENT_ARGS` such as `--disconnect --captive-zone guest`. Commands: `status`, `stop`, `restart` and `remove` (which also runs `--remove-all`). The log is `/var/log/authcha_webpolicy.log`.

How fast is it

With `--interval`, the agent does not sleep: it asks the console to answer the moment a session starts or ends (or after 25 seconds with "unchanged"). A change is applied about **0.5 to 1 second** after a sign-in, and a disconnect you queue wakes the agent at once. For delivery within about a second **and** failure alerts, the **pfSense alias connector** (the Enforcement connectors module later in this track) uses the pfSense REST API package.

Exit codes

`0` done. `2` Authcha unreachable (nothing changed). `3` a change was refused by Squid or pfSense and rolled back. `4` configuration error.

Lab HANDS-ON

Needs a pfSense lab with the captive portal and Squid.

1. Create a **Web policy agent** token and save it with mode 0600.
2. Run `--check`. Fix each `FAIL` line.
3. Run with `--dry-run` and read what it would create.
4. Run it for real, sign a test user in at the portal and watch the Squid files appear in `/usr/local/etc/squid/authcha/`.
5. Sign in as a user with a deny-list policy and try a blocked and an allowed site.
6. Queue an **End session** in the console and time how long the agent takes to end it.
7. Run `authcha_webpolicy.sh remove` and confirm the agent's files and rules go.

Cheat sheet KEEP THIS

- pfSense CE: no per-user URL lists, **no CoA**. Agent decides nothing; it **applies** Authcha's policy and ends sessions.
- Token kind **Web policy agent**. `/usr/local/authcha/token` mode 0600. CA at `ca.pem`.
- Squid transparent + **Splice All** + include line **before** the allow-LAN rule.
- Captive portal: `--require-session`.
- Always `--dry-run`, then `--check`. Service: `rc.d/authcha_webpolicy.sh`.
- Exit: 0 ok, 2 unreachable, 3 refused change, 4 config.
- Read more: pfSense integration (https://demo.authcha.cloud/docs/pfsense-integration).

Module 3 of 6 · 30 minutes

OpenVPN users through RADIUS on pfSense

By the end you can

- Connect a pfSense OpenVPN server to Authcha as its authentication backend
- Give one user a fixed tunnel address and routes
- Explain why a route behind a client needs a kernel route, and how the agent adds it safely
- Name the limits: no accounting, no interactive second factor

One account for everything

Remote-access VPN users sign in to pfSense's OpenVPN server with the **same usernames and passwords as everyone else**. pfSense asks Authcha over RADIUS, so there is **no user list on the firewall**.

Status: not verified on a real pfSense OpenVPN server

The Authcha side is the same code the captive portal uses and is covered by its tests. The pfSense side follows pfSense's documentation but **has not been run against a real OpenVPN server by us**. The guide ends with a list of what to check first. Treat the first deployment as a pilot.

Set it up

1. **Register the firewall in Authcha. RADIUS Clients** → **Add** (the address its RADIUS packets come from, and a secret), then **Devices** → **Add** with the same address, device type **Captive portal / hotspot (PAP)** (the plain username-and-password profile; it has nothing to do with the captive portal feature itself) and **CoA mode: none**. If the firewall puts another address in `NAS-IP-Address`, add it under **Additional addresses**, or the request is refused as an unknown device.
2. **Add Authcha on pfSense**: **System** → **User Manager** → **Authentication Servers** → **Add**. Type **RADIUS**, protocol **PAP** (Authcha verifies a stored hash, so CHAP and MS-CHAP cannot work), Authcha's address, the secret, **Authentication only**, port 1812, and the **RADIUS NAS IP attribute** for the interface you registered.

3. **Use it in the OpenVPN server**: **VPN → OpenVPN → Servers**, mode **Remote Access (User Auth)** or **(SSL/TLS + User Auth)**, backend = the server from step 2. With **SSL/TLS + User Auth** each user needs a client certificate **and** a password, the recommended setting and the only second factor today. Turn **Enforce local group** off unless you create a group of the same name on the firewall.

Everything the central store decides applies: a disabled user, a locked user (three wrong passwords lock for 15 minutes; **Users → Unlock** clears it) or a disabled user type is refused, with the reason in **Live Auth**.

A fixed address and routes for one user

Under **Network Access → Network Policies → Fixed address and routes for one user**, enter the user, an address such as `10.99.0.50`, optionally a netmask, and routes. They apply from the next login. Two different fields exist and the difference matters:

In Authcha	In the pfSense override	What happens
Routes to push to the VPN client (`10.1.0.0/24`)	IPv4 Local Network/s	The client gets `push "route 10.1.0.0 255.255.255.0"` at its next connection
Framed-Route (`192.168.5.0/24 gw metric`)	IPv4 Remote Network/s	OpenVPN learns the network is **behind** this client. **No kernel route is added by the override**

pfSense gave the client the fixed address in the lab but did **not** act on `Framed-Route`: that RFC 2865 attribute names a network *behind* the user, and pfSense's OpenVPN only reads routes from a **client-specific override** (CSO) matched on the user name. So the pfSense agent, run with `--openvpn`, keeps one override per user that has routes. Turn **Username as Common Name** on, because the override is matched on the user name.

A network behind a client needs a kernel route

The override only tells OpenVPN which client owns the network. For traffic to reach it, pfSense also needs a route through the OpenVPN server's interface. Two ways:

- **Let the agent do it:** add `--openvpn-routes`. For each network behind a client it runs `route add -net NET -interface ovpnsN`, without restarting OpenVPN so **no connected user is dropped**. It **never replaces a route that already exists**,

refuses a network broader than /8, removes a route again when the user or route goes, and adds it back within ten minutes after OpenVPN or the firewall restarts.

- **By hand:** add the network to **IPv4 Remote Network/s** on the OpenVPN server. Saving restarts OpenVPN and **drops every connected user**, which is why the agent does not do it.

> **Use a network that exists only behind the client**
>
> A route toward a network **takes over** its traffic, including traffic that used to follow the default route. A Framed-Route for a network reachable some other way (a real office network, a public range) sends that traffic into the VPN and breaks the original path. "Never replaces a route that already exists" protects only a network with its own route entry. **The default route does not count**, because every network is "reachable" through it.

Check with `netstat -rn -f inet | grep 192.168.5` on the firewall: the network must show the `ovpns` interface.

Web limits for VPN users

pfSense's OpenVPN sends **no accounting**, so Authcha has no session to find a VPN user by. A user with a **fixed address** is therefore treated as signed in at that address, and the Squid agent applies their web policy there, provided the address is the one their traffic comes from after the tunnel and **Squid sees that traffic** (add the OpenVPN interface to the interfaces Squid serves). A VPN user **without** a fixed address gets no web limit.

Limits to remember

- **No accounting:** VPN sessions are not in **Network Sessions** and have no session quotas. Use pfSense's **Status** → **OpenVPN**.
- **No interactive one-time code:** a second factor today is the client certificate. SMS and WhatsApp codes are for the captive portal and the console, not the VPN.

Test the RADIUS side first

```
docker run --rm freeradius/freeradius-server:3.2.10 radtest -x alice 'her password'
```

`Received Access-Accept` means the device, user and password are right. `Access-Reject` plus the reason in **Live Auth** names the cause. Then connect with the OpenVPN client while watching **Live Auth**.

Lab HANDS-ON

On the demo as `demo-viewer` (password `Authcha-View-2026`) for steps 1 and 2.

1. Open Network Policies (https://demo.authcha.cloud/nac/policies) on the demo and find **Fixed address and routes for one user**. Which demo users have one?
2. Open Users (https://demo.authcha.cloud/users) and find a user with a fixed address and one with a route behind the client.
3. On a lab pfSense, follow steps 1 to 3 and connect a test user with a certificate and password.
4. Give the user a fixed address and a pushed route. Reconnect and read the tunnel address and routes on the client.
5. Add a Framed-Route for a network that exists only in your lab. Run the agent with `--openvpn --openvpn-routes --dry-run`, then for real, and check `netstat -rn`.
6. Try a Framed-Route for a network already reachable another way and read what the agent says.

Cheat sheet KEEP THIS

- pfSense: **System → User Manager → Authentication Servers → RADIUS, PAP**, then the OpenVPN server backend.
- Device type **Captive portal / hotspot (PAP)**, **CoA none**.
- Fixed address and routes: **Network Policies → Fixed address and routes for one user**. **Push routes** vs **Framed-Route**.

- Agent: `--openvpn` (overrides) and `--openvpn-routes` (kernel route, no restart, never over an existing route, nothing wider than /8).
- No accounting, no interactive second factor. Certificate + password is the second factor.
- **Not verified on a real OpenVPN server**: pilot first.
- Read more: pfSense OpenVPN (https://demo.authcha.cloud/docs/pfsense-openvpn).

Module 4 of 6 · 35 minutes

Delivery channels for any second factor

By the end you can

- Describe a delivery channel as one HTTP request you define, or an email through the server's mail settings
- Build a channel with placeholders, credentials and a success check
- Preview and test a channel without risk, and read its last result
- Set an ordered failover list per user type, and route device-login codes through channels

Why channels exist

Every deployment sends second-factor messages differently: one uses an SMS gateway with a URL, another WhatsApp Cloud API, another Telegram or Slack. Authcha does not keep a list of supported providers to wait for. A **delivery channel is one HTTP request that you describe**, so any service with an HTTP API works. A message can be a one-time **code**, a **passkey approval link** or an **enrollment link**.

Open **Integrations** → **Delivery Channels**. With **no channel defined, nothing changes**: codes still go through OTP Delivery, the WhatsApp and SMS service that ships with Authcha. A channel takes over only when you add one.

Build a channel

1. **Add channel**, then pick a starting point (Twilio SMS, Twilio WhatsApp, WhatsApp Cloud API, Telegram bot, Slack webhook, generic webhook, an SMS gateway that takes a URL, an email API) or **Custom API**. A preset only fills in the form. Every field stays editable.
2. Fill in the credentials. They are stored **encrypted** and are **never shown again**. They need `PKI_KEY_ENCRYPTION_KEY` in `.env`, and without it the console refuses to store a credential.
3. **Save**, then **Test**.

What you describe

Field	Meaning
Method and URL	`POST` for most providers, `GET` for gateways that take everything in the URL. Host and port are fixed. Placeholders may appear in the path and query
Headers	Any header you need. The caller sets `Host` and `Content-Length` itself
Query values	Added to the URL and escaped for you
Sign in with	Nothing, a **bearer token**, or **basic** (user and password)
Body	None, **JSON**, **form fields** or plain text
Who it is sent to	Which detail of the user it needs: mobile number, email, linked chat, user name, or none for a shared destination such as a Slack channel. A user without that detail is **skipped**, and the next channel is tried
OK from HTTP … to …	The status range that means "accepted". Default 200 to 299
Response field	A JSON Pointer such as `/ok` and the value it must have. For providers that answer 200 even on refusal (Telegram answers `{"ok": false}`)
Timeout and retries	1 to 30 seconds, and up to 3 more attempts after a network error, HTTP 429 or a 5xx. Other failures are not retried
Certificate authority	Optional PEM. When set, only that authority is trusted for this destination

Placeholders

Write `{{name}}` where a value goes. Single braces are plain text, so a JSON body needs no escaping.

Placeholder	Value
`{{message}}`	The finished text: "Your sign-in code is 482913. Valid for 5 minutes."
`{{otp}}`, `{{code}}`, `{{link}}`, `{{purpose}}`, `{{expiry}}`	The parts of the message
`{{mobile}}`, `{{email}}`, `{{chat_id}}`, `{{username}}`	The recipient
`{{kind}}`	`otp`, `link`, `enroll` or `test`
`{{secret.NAME}}`	A credential you stored. Add the `base64` filter after the name to get the same value base64 encoded (used for basic-auth style headers)

Every value is **escaped for the place it lands**: JSON string content in a JSON body, URL-encoded in a form body or query, and a control character is refused in a header. A user name or purpose line therefore cannot add a field, end a string or inject a header. Settings are checked when you save: an unknown placeholder, JSON that is not valid once filled in, or a missing credential is refused with the reason.

Examples

- **Telegram:** `POST https://api.telegram.org/bot{{secret.token}}/sendMessage`, body `{"chat_id":"{{chat_id}}","text":"{{message}}"}`, recipient *linked chat*, response field `/ok` equal to `true`.
- **An SMS gateway that wants everything in the URL:** method `GET`, URL `https://sms.example.com/api/send`, query values `apikey={{secret.key}}`, `to={{mobile}}`, `text={{message}}`, body none.
- **Slack:** `POST https://hooks.slack.com/services/{{secret.path}}`, body `{"text":"{{username}}: {{message}}"}`, recipient none.
- **Email through this server's mail settings:** choose that option under **How it is sent**. There is no request to describe: the message goes to the user's email address through the mail server set under **Integrations** → **Email**.

Safety rules that always apply

- **Plain HTTP is off by default.** An on-premises gateway that speaks only HTTP needs **Allow plain HTTP (not encrypted)** on that channel, and the console then shows an **Unencrypted HTTP** label on it at all times.
- The address of **this server** and link-local addresses (including cloud metadata) are always refused, and **redirects are never followed**.
- Error text never contains the URL or a credential.

> **Plain HTTP exposes the code**
>
> With plain HTTP the code, the link and any credential in the request cross the network in clear text. Use it only on a trusted private network, never across the internet.

Test before you rely on it

Test opens a dialog for a recipient. **Preview request** shows the exact request that would be sent, with credentials hidden, and sends nothing. **Send test** delivers a real message and reports what the provider answered. The channel's **last result** (sent or failed, and why) shows on its card.

Who gets which channel, and failover

Under **Who gets which channel**, each user type has an **ordered list**. A message tries the channels top to bottom and stops at the first that accepts it. A channel is **skipped**, not failed, when the user has no number, address or chat for it. A user type with no list of its own uses the **Default** list, and with no list at all the OTP Delivery service is used. A common order is **WhatsApp, then SMS, then email**.

Codes from your network devices

One-time codes for **device logins** are made and sent by the RADIUS server. To send them through channels, set `OTP_VIA_CHANNELS=true` in `.env` and recreate `freeradius`. FreeRADIUS then posts each code to the console, on this server only and with the same bearer token it already uses. With this on, a user with **no mobile number** can still get a code through a channel that uses their email or linked chat. Passkey approval links, enrollment links, the captive portal's codes and identity-provider codes use channels as soon as one exists, with no setting.

Limits

One HTTP request per message, so a provider that needs a token fetched on every send (an OAuth exchange) is not supported; use a long-lived key. Text only. The

response is judged by status and one JSON field. Providers that report delivery later by callback are treated as accepted when they accept the request.

Lab HANDS-ON

Needs your own install. The demo does not send messages.

1. Open **Integrations → Delivery Channels** and look at the presets.
2. Add a **Slack webhook** or **generic webhook** channel pointing at a test receiver you control. Store the secret, then **Preview request** and check the credentials are hidden.
3. Click **Send test** and confirm the receiver gets the message. Read the channel's last result.
4. Break it on purpose: use a wrong status range or a missing placeholder. Read how the console refuses it.
5. Add a second channel, then set an ordered list for one user type. Remove a user's number and confirm the first channel is skipped and the second is used.
6. Set `OTP_VIA_CHANNELS=true`, recreate `freeradius`, and sign in to a device as a user with an email but no number.

Cheat sheet KEEP THIS

- **Integrations → Delivery Channels.** One channel = one HTTP request (or email via the server's SMTP).
- Placeholders: `{{message}}`, `{{otp}}`, `{{mobile}}`, `{{email}}`, `{{chat_id}}`, `{{username}}`, `{{secret.NAME}}`. Escaped per place.
- Credentials: encrypted, never shown, need `PKI_KEY_ENCRYPTION_KEY`.
- Success: status range and optional **response field** pointer.
- **Preview** (no send) then **Send test**. Read **Last result**.

- Order per user type: **WhatsApp, SMS, email**. Missing detail = skipped.
- Devices: `OTP_VIA_CHANNELS=true` + recreate `freeradius`.
- Plain HTTP: opt-in, labelled **Unencrypted HTTP**. Own address and link-local always refused. No redirects.
- Read more: Delivery channels (https://demo.authcha.cloud/docs/delivery-channels).

Module 5 of 6 · 30 minutes

Enforcement connectors

By the end you can

- Explain what an enforcement connector delivers and why Authcha does not block anything itself
- Choose between Squid, the pfSense alias, a webhook, syslog, FortiGate and Palo Alto
- Describe the delivery guarantees: ordered, retried, reconciled
- Explain fail closed and what an outage does to restrictions

What they deliver

Authcha knows **who is on which address**, because every RADIUS accounting packet says "this user, at this address, on this NAS". It does not block anything itself. **Enforcement connectors** deliver that fact, together with the user's policy, to the devices that can act on it, within a second or two of the accounting Start and again of the Stop, and keep those devices correct across restarts and outages. Open **Network access → Enforcement**. It is part of the NAC edition.



Which one

	Decides on	Good for
Squid + the Authcha helper	The TLS server name and HTTP `Host` header	A user held to one site, or kept from some, **exactly**, even when sites share an address
pfSense alias	The address only	"These users are limited": rate limits, ports. **Approximate for websites**
Webhook	Whatever your receiver does	Any firewall, NAC or automation that takes an HTTP call
Syslog (CEF or JSON)	Whatever your SIEM or firewall does	Identity feeds into a SIEM, or a firewall that learns users from syslog
FortiGate (RADIUS accounting SSO)	FortiGate policies on the group Authcha supplies	FortiGate user-based policy
Palo Alto User-ID	PAN-OS policy keyed on the user	User-ID based security policy

An address-based enforcer cannot isolate one website on a CDN. Use Squid for that.

What every connector gets

- **Event driven.** A database trigger writes an outbox row and sends a notification. There is no polling. A one-second tick is only a safety net.
- **Ordered per address.** One pending delivery per connector and address. If a Start, Stop and second Start arrive together, the connector receives what is true **now**.
- **A "stop-start" is not a logout.** Some NAS (pfSense) end a session with the cause `NAS-Request` and start it again a second later. Releasing at once would leave a restricted user unrestricted for that gap, so such a Stop is held for `release_grace_seconds` (default 8) and a Start in that time cancels it.
- **At least once, idempotent.** A delivery is forgotten only after the device confirmed it, and every operation is a set operation. A webhook receiver can de-duplicate on `key`.
- **Retries and dead letters.** A failed delivery is retried after 1, 2, 4, up to 60 seconds. After ten attempts it is a **dead letter**, shown on the page with its error. **Retry** or **Discard** it.
- **Reconcile, not only deltas.** On console start, after an outage, when a connector is edited, on **Re-sync**, and every five minutes, the connector is pushed the **complete**

state. A restart, a lost event or a hand edit cannot leave it wrong for long.

- **Secrets sealed.** Keys and secrets are encrypted under `PKI_KEY_ENCRYPTION_KEY` and never returned by the API.
- **Guarded and verified.** A connector never reaches this host or a link-local address, redirects are refused, and TLS certificates are **always verified**. There is no "skip verification" switch.

Outages never lift a restriction

While a connector's device is unreachable, **nothing is released**. Aliases, User-ID entries and Squid's cache stay as they are, and the connector retries. What a **new** limited session gets while its connector cannot confirm it is the **fail closed** option:

	Off (default)	On
A limited session unconfirmed after the grace period (default 10 s)	Marked **unenforced**, and an alert is raised	The same, **and the session is disconnected** (quarantine)
The user	Keeps the unenforced access until the connector is back	Must sign in again, and is enforced once the connector returns

Authcha cannot configure a device that is down, so with fail closed it **removes the unenforced session** instead. Only users who resolve to a web policy are ever quarantined. Watch the metric `authcha_enforcement_unenforced_sessions`: alert on anything above zero.

The connectors in a little detail

- **pfSense alias.** Keeps an alias (default `authcha_limited`) equal to the addresses of limited users, optionally one per policy. It writes only aliases that differ, **waits until the live pf table matches**, then kills the connection states of changed addresses so existing connections meet the rule, and on any error **restores the previous contents**. It needs the pfSense REST API package and a certificate chain the console can verify: the server certificate must be a **leaf** certificate issued by the CA you paste, **not the CA certificate itself**. Its mock tests have **not** been checked against a real pfSense.
- **Webhook.** One signed JSON `POST` per event (`session.apply`, `session.release`, `session.revoke`, `snapshot`, `test`). The signature is

HMAC-SHA256 of `<timestamp>.<body>`, so a receiver can verify it. Answer any 2xx to confirm.

- **Syslog.** RFC 5424 over UDP, TCP or TLS, with a CEF or JSON body. UDP has no acknowledgement, so use TCP or TLS when delivery matters.
- **FortiGate.** Sends RADIUS **Accounting** packets (UDP 1813) with `Fortinet-Group-Name`. Not tested against a real FortiGate.
- **Palo Alto.** `type=user-id` login and logout messages, with a timeout so a dead console does not leave users mapped forever. Not verified against a real PAN-OS.
- **The Squid helper.** An external ACL helper that decides on the TLS server name from the policy Authcha resolved for the client's address. It supports **fail-closed or fail-open** and `--require-session`.

> **Be honest about what ran on real gear**
>
> Several connectors were exercised against mock devices written from vendor documentation. Test each on your own firewall before relying on it.

Lab HANDS-ON

1. Open Enforcement (https://demo.authcha.cloud/nac/enforcement) on the demo as `demo-admin` (password `Authcha-Demo-2026`). Look at the connectors, queue depth and health.
2. Add a **webhook** connector pointing at a test receiver you control and click **Test**.
3. Sign a test user in and watch the `session.apply` event arrive. Verify its HMAC signature with a few lines of Python.
4. Stop the receiver, sign another user in, and watch the delivery retry and eventually become a dead letter. Start the receiver and press **Retry**.
5. Turn on **Fail closed**, stop the receiver, sign a limited user in, and see the session quarantined after the grace period.

6 Read `authcha_enforcement_unenforced_sessions` in the metrics.

Cheat sheet KEEP THIS

- **Network access → Enforcement.** Connectors: **pfSense alias, webhook, syslog, FortiGate, Palo Alto, Squid helper**.
- Squid = **exact**. Address-based = **approximate**.
- Guarantees: event driven, **ordered per address**, at-least-once, retried 1 to 60 s, **dead letter after 10**, **full re-sync every 5 min**.
- `NAS-Request` stop is **held 8 s** by default.
- **Outages never lift a restriction. Fail closed** quarantines unconfirmed limited sessions.
- Webhook signature: **HMAC-SHA256 of** `<timestamp>.<body>`.
- Alert on `authcha_enforcement_unenforced_sessions > 0`.
- Read more: Enforcement connectors (https://demo.authcha.cloud/docs/enforcement-connectors).

Module 6 of 6 · 25 minutes

The API and automation

By the end you can

- Create an API token and call the API with it
- Read the live OpenAPI description and the built-in API page
- Handle the error statuses correctly in a script
- Create a user through the API and from the host command line

Everything goes through the API

Everything the console does goes through its REST API under `/api`, so anything you can do in the console you can script. Terraform, Ansible and CI jobs use the same endpoints as the console, with the same role permissions.

Tokens

- **Scripts:** create an API token under **Settings, API tokens** (or `POST /api/tokens`) and send it as `Authorization: Bearer ra_…`. A token has the role it was created with: an **admin** token may do everything an administrator may, a **viewer** token may only read. There are **no finer scopes**, so give automation the narrowest role that works.
- **Browsers:** the `ra_sid` session cookie from `POST /api/auth/login`.

The plaintext token is shown **once** at creation and Authcha stores only a hash, so a leaked token can be revoked but not recovered. A request with a Bearer header that is not valid, unexpired and unrevoked is refused with 401, **even if a cookie is also sent**. Changes made with a token appear in the audit log like any other. The API tokens page now warns when a token has **14 days or fewer** left, because automation using it stops when it expires.

Treat a token like a password

Never put a token in a script that others can read, and never send one in a ticket. Set an expiry, and revoke the old token when you issue a new one.

The OpenAPI description

Where	What
`GET /api/openapi.json`	The live description for your version. Needs a session or token
`/api/docs`	A built-in page listing every operation, who may call it and what it does. Works offline, with no external scripts or fonts. Sign in to the console in the same browser first
`docs/product/openapi.json`	A copy for tools that read a file: code generators, Postman, linters

Each operation lists the roles allowed through (`x-roles`) and the error model. A test fails whenever a route is added without being described, so the description cannot fall behind. `GET /api/guides` lists the product guides, and `GET /api/guides/<slug>` returns one as Markdown, for scripts and for anyone without a browser.

Errors

A failed request answers with a status and, mostly, `{ "error": "message" }`.

Status	Meaning
400	Invalid input or malformed JSON. The message names the problem
401	Not signed in, session expired, or the token is invalid, revoked or expired
403	The role, the administrator's provisioning scope, the license, the edition or the public demo refuses it. The message says which
404	No such object
409	Already exists, or in the wrong state
415	The body was not sent as `Content-Type: application/json`
422	The JSON parsed but a field is missing or the wrong type. The answer is plain text, not the JSON shape
429	Rate limited. Read the `Retry-After` header
502, 503	A service the request depends on refused or is unavailable
500	Internal error. Details only in the server log

A good script **retries 429 and 503 with a delay, and fails loudly on everything else.**

Create a user in one call

```bash
curl -sS -X POST https://authcha.example.internal/api/users \
  -H "Authorization: Bearer $AUTHCHA_TOKEN" \
  -H "Content-Type: application/json" \
  --data @- <<'JSON'
{
  "username": "alice",
  "password": "a-long-passphrase",
  "user_type": "human-read",
  "mobile_number": "",
  "email": "alice@example.com",
  "display_name": "Alice Example"
}
JSON
```

`username` is 1 to 64 characters from `A-Z a-z 0-9 @ . _ -`. The password is stored as a bcrypt hash and never returned. `user_type` must be an enabled entry of `GET /api/user-types` that your administrator account may provision. `mobile_number` is required, so send `""` if there is none. A name that already exists, even one in the trash, answers **409**. Change a password later with `PATCH /api/users/{username}`.

Create a user from the host

Without the console or a token, on the Authcha host:

```bash
sudo authcha user-create --username alice --user-type human-read \
     --mobile "+15550100" --password-file ./alice.pw
printf '%s\n' "$PW" | sudo authcha user-set-password --username alice --password-std
```

The password is **never an argument**, because it would show in the process list. Give it on standard input, in a file only its owner can read (mode 0600), in `AUTHCHA_USER_PASSWORD`, or at a prompt. `--update` changes a user if it exists and creates it if not. Exit codes: 0 done, 1 unexpected, 2 usage, 3 already exists (or does not exist), 4 refused (invalid name, unknown or disabled type, or the license). Each success writes an audit entry shown as `cli`.

Voucher redemption is the one public call

`POST /api/voucher/redeem` is public, for captive portals, with body `{"code", "mac_address"}`. Every failure has the same body and a randomised delay, and the rate limit answers 429.

Lab HANDS-ON

On your install (the demo's API is read-only).

1. Create a **viewer** token under **Settings, API tokens**. Save it once.
2. Call `curl -s -H "Authorization: Bearer $TOKEN" https://<host>/api/openapi.json | head` and then try a write call with it. Read the 403.
3. Open `/api/docs` in your signed-in browser and find the operation that creates a user and its `x-roles`.
4. Create a user with the API using an **admin** token. Run it twice and read the 409.
5. Send the call without the `Content-Type` header and read the 415.
6. Create a user from the host with `authcha user-create`, using a password file. Check the audit entry says `cli`.
7. Revoke the tokens you made.

Cheat sheet KEEP THIS

- Token: `Authorization: Bearer ra_…`. Roles **admin** or **viewer** only. Shown **once**. Warns at **14 days** left.
- Description: `/api/openapi.json`, `/api/docs`. Guides: `/api/guides`.
- Errors: **400** input, **401** auth, **403** role/license/edition/demo, **409** exists, **415** wrong content type, **422** wrong field, **429** rate limit (`Retry-After`).

- Create a user: `POST /api/users`, or `sudo authcha user-create` (**no password as an argument**).
- Public: `POST /api/voucher/redeem` only.
- Read more: API reference (https://demo.authcha.cloud/docs/api-reference).