AUTHCHA LEARN

Identity and security

Certificates, RadSec, passkeys, Authcha as an identity provider, posture and device compliance.

5 modules · about 155 minutes · Security engineers and administrators

1. Certificates and EAP-TLS
2. RadSec: RADIUS over TLS
3. Passkeys and push sign-in for device logins
4. Authcha as an identity provider
5. Posture checks and MDM compliance

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 5 · 35 minutes

Certificates and EAP-TLS

By the end you can

- Explain how Authcha's certificate authority works and when a certificate is accepted
- Issue a user certificate and a device certificate
- Configure a client for EAP-TLS and set the expected server name
- Revoke a certificate and make the device drop off at once
- Turn on CRL and OCSP for other systems that trust your certificates

Why certificates

With EAP-TTLS a person types a password. A password can be phished, reused or shared. With **EAP-TLS** the **certificate is the credential**: nobody types anything at the switch or access point, and there is nothing to phish. EAP-TLS runs alongside EAP-TTLS, MAB and the device-login flows, so you can move people and devices over one at a time.

How it works

- **The certificate authority (CA).** On first start the RADIUS service creates a private CA for this installation and uses it to issue its own server certificate. The console then takes the CA over: its private key is stored **encrypted in the database** under `PKI_KEY_ENCRYPTION_KEY`, and is **removed from the RADIUS service**. From then on the console issues every certificate and renews the server certificate automatically.
- **Admission.** A client certificate is accepted only if **all** of these are true: it chains to the Authcha CA, the console issued it and has not revoked it, it has not expired, and, for a user certificate, the user exists, is not locked and has an enabled user type. Authcha checks this at **every connection**, so revocation takes effect immediately with no CRL to publish.
- **Identity.** The session belongs to the certificate's owner: the user, or the device name. The name the device sends in the clear is ignored, so a certificate cannot claim to be someone else.
- **Authorization.** A certificate can name a **device class**, and the session then gets that class's VLAN, QoS and timeout, exactly like MAB. Posture still applies on top.

Issue a certificate

Network Access → Certificates → Issue certificate.

1. Choose **A user** or **A device**. A **user** certificate signs in as an existing RADIUS user, so locking or deleting the user stops the certificate too. A **device** certificate signs in as the device name you give, and needs a device class.
2. Optionally set a device class, a validity period (users default to 1 year, devices to 2, **maximum 5 years**), a MAC address for your records and DNS names.
3. Choose where the key comes from:
4. **Generate here.** Authcha makes the key and gives you a `.p12` file and its password **once**. The private key is not stored, so neither can be shown again. For older macOS, iOS or Android that refuse the file, issue again with **Legacy .p12 encryption**.
5. **Paste a CSR.** The device keeps its own key and you paste its signing request. Only the public key is used.

Accepted keys: RSA 2048 bits or more, ECDSA P-256 and P-384.

Configure the client

Every client needs the **CA certificate** (download it under **Certificate authority → Download CA certificate**) and its own **client certificate**. Also set the expected **server name**, shown on the same page. Without it, a client trusts any certificate from the CA.

- **Windows:** import the `.p12` into the user or computer store and the CA into Trusted Root Certification Authorities. Deploy a wired or Wi-Fi profile that uses **Smart card or other certificate** (EAP-TLS), with **Verify the server's identity** on and the server name filled in. At scale use Group Policy or Intune.
- **macOS and iOS:** a configuration profile with a Certificate payload for the CA, a PKCS#12 payload for the client certificate, and a Wi-Fi or Ethernet payload with TLS and the trusted server name.
- **Android:** install the CA as a Wi-Fi certificate and the `.p12` as a user certificate, then choose EAP method TLS, the CA, the server's domain and the user certificate.
- **Linux:** NetworkManager with `802-1x.eap tls` and the CA, client certificate and `802-1x.domain-suffix-match`.

Revoke

Choose **Revoke** next to the certificate and pick a reason. The certificate is refused at its next connection. **Also disconnect the current session** is on by default: Authcha

sends a Disconnect-Request so the device drops off immediately, which needs CoA configured on the switch or controller. **Revocation cannot be undone.** Issue a new certificate if access is needed again.

CRL and OCSP for other systems

Authcha asks its own database about every certificate, so it never needs a CRL. **Other** systems that accept your certificates, such as a VPN gateway, a web server or Windows, cannot ask the database. For them Authcha can publish a **CRL** for each CA, rebuilt the moment something is revoked, and an **OCSP responder** that answers good, revoked or unknown from the live database.

It is off until you turn it on under **Certificates → Certificate authority → Revocation information**. Enter the **base URL** those systems will use (a scheme, a name or address, and an optional port), turn on **Publish**, and save. From then on certificates you issue carry the CRL and OCSP addresses. **Certificates issued earlier are not changed.**

These feeds are published over plain HTTP on **port 80**, because a client checking the revocation of an HTTPS server's certificate cannot first fetch the list over that same HTTPS connection. Keep port 80 reachable from the systems that validate your certificates. They contain serial numbers of revoked certificates and nothing else.

The server certificate

Certificate authority shows the server certificate supplicants see. When the Authcha CA issued it, Authcha renews it 45 days before expiry with the same name and CA, so clients keep trusting it. A server certificate from your **own** CA is yours to renew, and Authcha then makes a separate CA for client certificates.

> **The key that matters**
>
> `PKI_KEY_ENCRYPTION_KEY` protects the CA's private key. **Back it up with the database.** A restored database cannot issue or renew certificates without it. See the Operations track.

Lab HANDS-ON

Steps 1 and 2 on the demo as `demo-admin` (password `Authcha-Demo-2026`). The rest on your install. The public demo does not publish revocation information.

1. Open Certificates (https://demo.authcha.cloud/nac/certificates) on the demo and look at issued certificates, their identity, class and expiry.
2. Open **Certificate authority** and find the server name and the CA download.
3. On your install, issue a **user** certificate for a test user, with **Generate here**. Save the `.p12` and its password.
4. Issue a **device** certificate from a CSR you made with `openssl`.
5. Configure a Linux client with NetworkManager (or `eapol_test` with `eap=TLS`) and sign in with the certificate.
6. Revoke the certificate with **Also disconnect the current session** on. Confirm the session ends and the next login is refused.
7. Turn on **Publish** for revocation information, issue a new certificate, and inspect it for the CRL and OCSP addresses.

Cheat sheet KEEP THIS

- **EAP-TLS**: the certificate is the credential. No password, nothing to phish.
- Accepted only if: chains to the Authcha CA, issued **and not revoked** by the console, not expired, and (for users) the user is enabled. Checked at **every** connection.
- Issue: **Network Access** → **Certificates**. Keys: RSA 2048+, P-256, P-384. Max **5 years**. `.p12` shown **once**.
- Clients need the **CA** and the **expected server name**.
- **Revoke** also disconnects the session by default. It is permanent.
- CRL and OCSP: off by default, plain HTTP **port 80**, new certificates only.
- Back up `PKI_KEY_ENCRYPTION_KEY`.

- Read more: Certificates (https://demo.authcha.cloud/docs/certificates), Enrollment (https://demo.authcha.cloud/docs/enrollment).

Module 2 of 5 · 25 minutes

RadSec: RADIUS over TLS

By the end you can

- Say what RadSec adds over UDP RADIUS and what it keeps the same
- Issue a RadSec certificate bound to the right addresses
- Explain what happens when a RadSec certificate is revoked
- Decide when CoA over RadSec is safe to turn on

What RadSec adds

Classic RADIUS sends UDP packets protected by a **shared secret**. Anyone who learns the secret can forge requests, and packets crossing networks you do not control can be watched. **RadSec** sends the same RADIUS over a **TLS connection**. The device proves who it is with a **certificate** Authcha issued, and Authcha proves who it is with its server certificate. Nobody on the path can read or change the traffic, and there is **no shared secret to leak**.

Use it for devices that support it, and especially for traffic that crosses networks you do not control. **UDP 1812 and 1813 keep working alongside it**, so you can move devices one at a time. RadSec is part of the NAC edition.

How it works

- **Connection.** The device opens a TCP connection to port **2083** and both sides present certificates (mutual TLS). TLS 1.2 or newer is required, TLS 1.3 clients negotiate a post-quantum hybrid key exchange where they support it, and a device without a certificate is refused during the handshake.
- **A separate CA.** RadSec certificates come from their **own** CA, created the first time you issue one. A user or device certificate can never authenticate a network device over RadSec, and a RadSec certificate can never be used for EAP-TLS.
- **Admission.** A connection is accepted only if the certificate chains to the RadSec CA, Authcha issued it, it is not revoked or expired, and the connection comes from an address the certificate is **bound** to.
- **Address binding.** Every RadSec certificate is bound to the addresses, networks or host names the device connects from. A stolen certificate and key are useless from anywhere else. Networks wider than /16 are refused, and only IPv4 is supported.

- **Live revocation.** Authcha checks the database for **every request** on a RadSec connection, not only when it opens.
- **The shared secret** is the fixed word `radsec`, as RFC 6614 requires. It adds no security; TLS does the work.
- **Policy is unchanged.** 802.1X, MAB, access rules and posture work exactly as over UDP.

Issue a certificate

Network Access → Certificates → RadSec (network devices) → Issue RadSec certificate.

1. Enter the **device name**.
2. Enter the **IP addresses or networks** the device connects from, and optionally host names. At least one is required. **With NAT in between, enter the translated address, the one Authcha sees.**
3. Choose validity (default 2 years, maximum 5) and where the key comes from: **Generate here** (a `.p12` and the key as PEM, once) or **Paste a CSR**.
4. Download the certificate, the key and the **server CA certificate**.

The first certificate you issue also starts the RadSec listener, which restarts the RADIUS service once, for a few seconds.

Configure the device

The **RadSec** page shows everything and includes snippets for Cisco IOS-XE, Aruba AOS-CX, Juniper Junos, Fortinet FortiOS and radsecproxy.

Setting	Value
Server	The server name on the page, port **2083**
Validate the server against	The **server CA certificate** you downloaded
Present	The certificate and key you issued
TLS version	1.2 or newer
Shared secret	`radsec`, where one is still asked for

The device must connect to the **server name** (or an address the server certificate covers) or it refuses Authcha's certificate. Treat the snippets as a starting point: keywords differ by platform and release, and not every model supports RadSec. For a device that does not, run **radsecproxy** next to it. The device speaks UDP to the proxy, and the proxy speaks RadSec to Authcha.

Revocation and renewal

Revoke stops the device. When it takes effect:

Situation	Effect
The device opens a new connection	Its first request is refused
It already has a connection open	Its **next request** is refused
The connection itself	Stays open but useless, until closed: at most 1 hour after it opened, or after 5 minutes of silence

A device that **also** has a UDP shared secret is not affected by revoking its RadSec certificate. Remove that client too if you want it gone. **Edit sources** changes allowed addresses without a new certificate. **Renew** by issuing a new certificate, installing it, then revoking the old one. Authcha renews its own server certificate automatically, and devices that trust the server CA keep working.

CoA and Disconnect over RadSec

By default a Disconnect goes over UDP 3799 with a per-device secret. For a device with a RadSec certificate you can send it over TLS instead. Because RFC 6614 keeps the roles of the two ends, Authcha opens a **second, outbound TLS connection to the device's RadSec port** to send it.

Only turn this on if the device supports it

The device must accept RADIUS/TLS connections on its RadSec port (a RadSec **server**, not just a client), handle CoA-Request and Disconnect-Request on it, accept Authcha's certificate as a client certificate, and let Authcha's address in. Many devices only implement RadSec as a client, or accept CoA only over UDP. Leave it off for those, and Authcha keeps using UDP. It is tested against a software peer, **not** vendor hardware, so verify it on yours.

On the RadSec page, use the device's lightning-bolt action, tick *Send CoA and Disconnect over RadSec*, set the port and save. **Send test** sends a CoA-Request for a session that does not exist. A CoA-NAK ("session not found") proves the channel works and changes nothing.

Firewall and turning it off

Open **2083/tcp** from your devices to Authcha, and keep UDP 1812/1813 for devices still on UDP. If you enable CoA over RadSec, also allow TCP from Authcha to the device's RadSec port. To turn RadSec off, set `RADSEC_ENABLED=false` in `.env` and restart `freeradius`. UDP is unaffected either way.

Lab HANDS-ON

Needs your own install and a device or radsecproxy.

1. Open the **RadSec** page and read the status. It says **Not started** until you issue a certificate.
2. Issue a RadSec certificate for a lab device, bound to its address.
3. Configure radsecproxy (or the device) with the CA, certificate and server name. Send a login through it and find it in **Live Auth**.
4. Edit the certificate's sources to a wrong address and confirm the connection is refused. Put it back.
5. Revoke the certificate and watch the next request get refused.

6 Open Certificates (https://demo.authcha.cloud/nac/certificates) on the demo to see the list view, noting RadSec is disabled there.

Cheat sheet KEEP THIS

- RadSec = RADIUS over **TLS**, TCP **2083**, mutual certificates, secret `radsec`.
- Own CA, separate from EAP-TLS. Certificates are **address-bound** (IPv4, nothing wider than /16).
- Revocation checked on **every request**. Open connections end within 1 h or 5 min idle.
- Use **radsecproxy** for devices without RadSec.
- CoA over RadSec: only for devices that act as a RadSec **server**. Leave off otherwise.
- Off: `RADSEC_ENABLED=false`. UDP keeps working.
- Read more: RadSec (https://demo.authcha.cloud/docs/radsec).

Module 3 of 5 · 30 minutes

Passkeys and push sign-in for device logins

By the end you can

- Describe how passkey push works from the device, the phone and Authcha's side
- Meet the requirements: a named HTTPS console and a trusted certificate
- Enroll a user and set the push keyword and approval time
- Explain why number matching defeats push fatigue

What it replaces

A password can be phished. A code can be read over someone's shoulder or forwarded. A **passkey** is a cryptographic key kept on the person's phone, unlocked by fingerprint, face or PIN, and tied to **your console's web address**, so a look-alike site gets nothing.

Authcha uses passkeys in two ways:

- **Administrators** sign in to the console with a passkey.
- **Network device users**, the people who log in to routers, switches and firewalls, sign in with **passkey push**: they type a keyword instead of a password, get a link on their phone, tap the number the device shows, and confirm with their passkey.

Password and one-time code keep working alongside passkeys, per user, so you can roll out gradually.

How passkey push works

1. On the device, the user types their username and, as the password, your organisation's **push keyword** (default `push`).
2. Authcha sends a link to their phone over the same WhatsApp or SMS channel it uses for codes.
3. The device shows a two-digit number, for example **47**.
4. On the phone the link opens a page showing which device is being signed in to, from which address and when, and **three numbers**. The user taps **47** and confirms with fingerprint, face or PIN.
5. Back on the device, they finish signing in with the same privileges as with a code.

If they pick the wrong number, or tap **This wasn't me**, the sign-in is blocked and recorded.

Why number matching matters

An attacker who knows a username can make Authcha send a push, but approving it means picking the number shown on **the attacker's** screen. Someone who reflexively taps "approve" on a request they did not start picks a number at random, and blocks the attempt **two times out of three**, and the denial is logged. The default messages deliberately **leave the number out**, so it can only be read from the device being signed in to.

> **Never put the number in the message**
>
> If you customise the message templates, do not include `{code}`. A message that shows the number lets a user approve without looking at the device, which defeats the protection.

Requirements

- **A DNS name and HTTPS.** `APP_URL` in `.env` must be the `https://` address people use, with a DNS name such as `https://authcha.example.com`. An IP address or plain `http://` does not work.
- **A trusted certificate.** Phones must trust the console's certificate: a public CA, or your enterprise CA installed on every managed phone. Browsers disable passkeys on pages with certificate warnings.
- **Reachability.** Phones must reach that address on Wi-Fi, VPN or the internet.
- **Phones.** Any current iPhone (iOS 16 or later) or Android phone (Android 9 or later with Google Play services) with a screen lock.
- **Working message delivery** for codes, and a mobile number on each user.
- **Edition.** Sending enrollment links is part of **Passwordless Sign-in** (Professional and Enterprise). Console passkeys for administrators, and push for users who already have a passkey, work in every edition.

When `APP_URL` does not qualify, Authcha says why under **Profile** → **Passkeys**, does not offer passkeys at sign-in, and refuses to send enrollment links.

Enroll a user

1. **Users**, then the fingerprint icon on the user's row.

2. **Send enrollment link**. Authcha sends a one-time link to the phone and also shows it once, with a QR code.
3. The user opens it, names the device, chooses **Create passkey** and confirms with fingerprint, face or PIN.

A link works **once** and expires after **24 hours**. A new link cancels any earlier unused one. If the user has no mobile number, the link is still created, so share the QR code in person. A user can have several passkeys, and revoking one stops it approving sign-ins at once.

Settings

Under **Settings** → **Authentication Policy** → **Passwordless push**:

Setting	Default	Notes
Push keyword	`push`	One keyword organisation-wide, 4 to 32 characters, case-sensitive, and different from the code trigger phrase
Time to approve	90 seconds	How long the phone link stays answerable and how long an approval can be used. 30 to 300 seconds

Security properties worth knowing

- **User verification is always required.** A stolen unlocked phone is not enough, and neither is the link alone.
- **Phishing resistance.** A passkey works only on your console's own address.
- **One request at a time.** A new push cancels an earlier unanswered one. An approval is used once.
- **Fail closed.** Expiry, a wrong number, a delivery failure or an outage is a refusal, never an accept.
- **Tokens are never stored readable.** Links carry 256-bit random tokens, and Authcha stores only digests.
- **Licensing never blocks sign-in** for someone who already has a passkey.

Watch it

Audit Trail → **OTP Funnel** shows `push_sent`, `push_approved`, `push_denied` and `push_expired`. A `push_denied` means someone entered that user's name

and the keyword on a device and was refused, which is worth a look. **Admin Actions** records every enrollment link and every passkey added or removed.

Lab HANDS-ON

Steps 1 and 2 on the demo (viewer login `demo-viewer`, password `Authcha-View-2026`). The rest need your own install with a real `APP_URL`.

1	Open	Profile (https://demo.authcha.cloud/profile)	on the demo and read the	**Passkeys**	section.	
2	Open	Settings (https://demo.authcha.cloud/settings)	and find	**Authentication Policy**	.	
3	Check	`APP_URL`	on your install. Is it an	`https://`	DNS name? If not, what would you change?	
4	Enroll yourself:	**Users**	, the fingerprint icon,	**Send enrollment link**	, then create the passkey on your phone.	
5	On a test device, sign in with your username and the push keyword. Pick the right number on the phone and finish on the device.					
6	Repeat and pick a	**wrong**	number. Find the	`push_denied`	event in the	**OTP Funnel** .

Cheat sheet KEEP THIS

- Push: username + **push keyword** (default `push`) → link on phone → device shows **47** → pick it → passkey confirm → finish on device.
- Needs: `APP_URL` = `https://<DNS name>`, a certificate phones trust, reachable address, phones iOS 16+ or Android 9+, mobile numbers, message delivery.
- Enroll: **Users**, fingerprint icon, **Send enrollment link** (one use, **24 h**).
- Defaults: keyword `push`, approve within **90 s** (30 to 300).
- **Never** put `{code}` in a message template.

- Watch `push_denied` in the **OTP Funnel**.
- Read more: Passwordless sign-in (https://demo.authcha.cloud/docs/passwordless).

Module 4 of 5 · 30 minutes

Authcha as an identity provider

By the end you can

- Explain what an identity provider does and what Authcha offers
- Register a server-side and a browser or phone application
- Choose scopes and read the claims an application receives
- Manage an application: switch off, rotate the secret, restrict who may sign in

The idea

People already have an Authcha account for network devices. Authcha can also sign them in to your **applications**, such as webmail, a wiki, a ticketing system or software you wrote, by acting as a standard **OpenID Connect (OIDC) provider**. Any application that offers "Sign in with OpenID Connect", "OIDC" or "generic OAuth 2" can use it.

- **Passkey first.** People use the passkey they already enrolled: fingerprint, face or PIN, nothing to phish.
- **One account.** The same directory, user types, numbers and passkeys. Lock or delete a user once and every application stops accepting them.
- **Single sign-on.** After one sign-in, people move between applications without signing in again, for up to 8 hours.
- **Standard.** Authorization code flow with PKCE, signed tokens any OIDC library can verify, discovery, refresh, revocation and logout.

This needs the **Identity Provider** module (Professional and Enterprise).

Words used here

An **issuer** is the identity provider's own address, which every application checks. A **client** is an application registered with it. A **scope** is a permission to see certain facts, and a **claim** is one such fact about the person.

Requirements

`APP_URL` in `.env` must be the `https://` address people use, **with a DNS name**, such as `https://id.example.com`. It becomes the issuer. Users need enrolled passkeys (or you allow the password-and-code fallback per application). Applications call Authcha's endpoints server to server, so their hosts must trust Authcha's certificate.

Everything an application needs is in one **discovery document**: `https://id.example.com/.well-known/openid-configuration`. The console shows it with a copy button under **Integrations → Applications**. Only the **authorization code** flow is supported. Implicit and hybrid flows are not.

Register an application

Integrations → Applications → Add application.

1. Enter a **name**. People see "Sign in to *name*".
2. Choose the **type**:
3. **Runs on a server** (confidential): the application can keep a client secret. Webmail, wikis and most self-hosted software.
4. **Runs in the browser or on a phone** (public): single-page and mobile apps. No secret, so the application must use **PKCE**.
5. Enter the **redirect URIs**, one per line, also called callback URLs. They are compared **character for character**, with no wildcards, and Authcha never redirects anywhere else.
6. Optionally open **Advanced**, then **Add application**.
7. Copy the **client ID** and, for a server-side app, the **client secret**. The secret is shown **once**, and Authcha stores only a hash.

Advanced settings

Setting	Default	Meaning
Scopes	`openid profile email`	Which scopes the application may request
Access-token audience	none	Values in the token's `aud` claim. A server that validates tokens itself expects its own name
Access-token lifetime	10 minutes	How long an access token is valid
Stay signed in for	30 days	Refresh-token lifetime, extended each use
Your own application	off	Skips the consent screen. For applications your organisation runs
Allow password and one-time code	off	A fallback for people without a passkey
Who may sign in	every user type	Restrict to some user types
Post-logout redirect URIs	none	Where the app may send people after sign-out

Scopes and claims

An application asks for **scopes**, and each releases certain **claims**. `openid` is always required.

Scope	Claims
`openid`	`sub` : a stable, opaque identifier that is **never the username** and never changes
`profile`	`preferred_username` , `name`
`email`	`email` , `email_verified`
`groups`	`groups` : the person's user type
`offline_access`	Allows a refresh token

`email_verified` is `true` only when an administrator set the address or it came from your directory. **Use `sub` as the key for the person's account** in your application, and `preferred_username` for display and matching existing accounts. The ID token also carries `auth_time`, `amr` (how they signed in: a passkey is `hwk` with `user`, password and code is `pwd` with `otp`) and `sid`.

Manage applications

- **Switch off** stops all new sign-ins at once, keeping the settings.
- **Rotate secret** issues a new secret, shown once. The old one stops **immediately**, so update the application straight away.
- **Delete** removes the application and its sign-ins permanently.
- The type cannot be changed. Add a new application instead.

Every change is recorded in the audit trail.

Lab HANDS-ON

On the demo as `demo-admin` (password `Authcha-Demo-2026`) you can look at the list, and on your install you can do all of it.

1	Open	Integrations (https://demo.authcha.cloud/integrations)	on the demo and find	**Applications**	. Look at the discovery URL.
2	On your install, fetch the discovery document with	`curl -s https://<host>/.well-known/openid-configuration | head -30`			and find the authorization, token and JWKS endpoints.
3	Register a test	**confidential**	application with one redirect URI. Save the client ID and secret.		
4	Open the OIDC playground at	`/oidc/playground`	on your install (if present) or use any OIDC test client, and sign in with a passkey.		
5	Decode the ID token and check	`sub`	, `preferred_username`	and	`amr` .

6	**Rotate the secret**	and confirm the old one stops working. Then	**switch the application off**	and confirm a new sign-in is refused.

Cheat sheet KEEP THIS

- Authcha = **OIDC provider**. Needs the **Identity Provider** module and `APP_URL` = `https://<DNS name>`.
- Discovery: `/.well-known/openid-configuration`. Flow: **authorization code + PKCE** only.
- Register under **Integrations** → **Applications**. **Confidential** = has a secret (shown once). **Public** = PKCE.
- Redirect URIs: exact match, no wildcards.
- Scopes: `openid` (required), `profile`, `email`, `groups`, `offline_access`. Key accounts on `sub`.
- Lifetimes: access token **10 min**, refresh **30 days**, SSO up to **8 h**.
- Read more: Identity provider (https://demo.authcha.cloud/docs/identity-provider).

Module 5 of 5 · 35 minutes

Posture checks and MDM compliance

By the end you can

- Explain how posture and MDM compliance each feed an access decision
- Roll out a posture policy safely: monitor first, then enforce
- Enroll endpoints and read the compliance reports
- Describe how MDM matching works and why a stale answer is never "compliant"

Two sources of "is this device healthy?"

Authcha can ask two different questions before it lets a device have full access.

	Posture	MDM compliance
Who knows	A small **agent** on the device	Your **mobile device management** system (Intune or Jamf Pro)
Edition	Enterprise	Enterprise
Applies to	802.1X logins only	Anything an access rule can match, matched by MAC or certificate
Result	The device lands on a **remediation network**	An access rule admits or refuses it

Both use the same idea: a device that is not healthy still connects, and goes to a restricted place or is refused by a rule you write.

Endpoint posture

An agent on Windows and Linux reports to Authcha. Authcha compares the report with the **policy** for the device's class and marks it *compliant* or *non-compliant*. At 802.1X login it returns normal access, or the remediation network's VLAN, filter and session limit. If the answer changes during a session, Authcha **disconnects the session** so the switch re-authenticates it at once, which needs a CoA secret on the device. Posture never affects MAB or device-administration logins.

State	Network access
Compliant	Normal
Non-compliant (fails or could not run a required check)	Remediation network
Not reporting (missed its check-ins)	Remediation, or normal if the policy allows
Awaiting first report	Normal during the grace period, then remediation

"Couldn't verify" counts as failing.

What is checked

Built-in: antivirus running and current, firewall enabled, disk encryption (BitLocker on Windows, LUKS on Linux) and OS updates current. Add your own **registry rules** (Windows) and **process rules** (Windows and Linux). The agent sends **only pass or fail**, never file names or process lists.

Roll it out safely

The safe order is **monitor first, then enforce.**

1. **Create a remediation network.** Give the `quarantine` class (or one you make) a restricted VLAN under **Device classes**.
2. **Create a policy in monitor mode. Posture** → **Policies**, choose the device class, select the required checks. A new policy starts in **Monitor**, where nobody is restricted and no disconnects are sent.
3. **Enroll endpoints** under **Posture** → **Endpoints** → **Enroll devices**.
4. **Watch the Overview**, which shows who *would* be restricted. Fix the common failures or adjust the policy until it looks right.
5. **Switch to Enforce.**

> **A policy with no required checks treats every device as compliant**
>
> Nothing happens until you choose checks. And a `quarantine` class with no VLAN restricts nobody.

Enroll devices

Every device runs the agent and is enrolled under the 802.1X username it signs in with. **Enroll devices** gives ready-to-copy commands for:

- **one device**: a one-time secret for a username, valid 1 to 90 days
- **several users**: a list of usernames gives one secret each, as CSV
- **a fleet token**: for Intune, Group Policy, SCCM or Ansible, valid up to 90 days and optionally limited to a number of devices. A fleet-enrolled device's posture counts only for sessions from **its own network adapters**, so a token cannot vouch for someone else

The console serves the agent itself, with a one-line install for Debian or Ubuntu and for Windows (PowerShell as Administrator), checking the file's SHA-256. The endpoint must trust the console's certificate, so use a public one, or give private-CA clients the CA certificate. When you upgrade Authcha, enrolled agents update themselves.

Policies, reports and alerts

Each class has one policy, and a class with none uses `default`'s. Key settings: **Mode** (Monitor or Enforce), the **remediation network**, **grace for newly enrolled devices**, and **when a device stops reporting** (restrict it, which is recommended, or keep allowing it). **Overview** shows the compliance rate, a 30-day trend, the checks failing most and per-class results. **Alerts** can warn when compliance falls below a threshold (default 90%) or when many devices stop reporting at once, by email or an HTTPS webhook. Revoke a device's enrollment when you retire or reimage it. Revoking is always allowed.

MDM compliance

Authcha can ask Microsoft Intune (through Microsoft Graph) or Jamf Pro whether a device is compliant, and access rules decide on the answer.

- **Safe when the MDM is not.** Authcha keeps **its own copy** of each device's state and decides from the copy. If the MDM is slow or down, sign-ins are not slowed. Once the copy is older than the maximum age it counts as **unknown**, never as compliant.
- **Secrets stay secret.** The client secret is stored encrypted and never shown again.

How a device is matched

Network requests carry the **MAC address**, so that is what Authcha matches on. The MDM must know the device's Wi-Fi or Ethernet MAC. A device using a **private Wi-Fi address** will not match, so turn that off in the managed Wi-Fi profile.

For **EAP-TLS device certificates**, Authcha can match from the certificate instead, which is more reliable: the certificate carries the device's **serial number** or **Azure AD device ID** (in the common name or a DNS name), and it is compared with the MDM field. A certificate naming a device the MDM does not know counts as **Not managed**.

State	Meaning
Compliant	The MDM says so
Non-compliant	The MDM says so
Unknown	No verdict, data too old, connector off, or no usable MAC
Not managed	No connected MDM lists it

If a MAC is in more than one MDM, the **worst answer counts**. Intune's grace period is shown as **unknown**, because a network rule saying "compliant" should not admit a device Intune is already warning about.

React when compliance changes

A connected device keeps its network until it re-authenticates, which can be hours. Switch on **Re-authorize sessions when a device's compliance changes** on a connector (off by default) and Authcha sends a Disconnect-Request when a sync finds a change, so the device comes back through your rules. Limits protect you: the **first sync never disconnects anyone**, at most 25 sessions per sync (default) and 200 per hour, and a device at most once every 30 minutes.

> **Not yet proven on real tenants**
>
> The guide states plainly that the Intune and Jamf sync follows the vendors' API documentation but **has not been tested against live tenants**. Try it with one device first.

Lab HANDS-ON

Steps 1 and 2 on the demo as `demo-admin` (password `Authcha-Demo-2026`). The rest need an Enterprise license and your own devices.

1. Open Posture (https://demo.authcha.cloud/nac/posture) on the demo and look at the Overview, policies and endpoints.
2. Open MDM Compliance (https://demo.authcha.cloud/nac/mdm) and see how connectors are listed.
3. Give the `quarantine` class a VLAN, create a Monitor policy for one class, and enroll one test laptop.
4. Make the laptop fail a check on purpose (turn its firewall off). Watch the Overview show it would be restricted.
5. Switch the policy to **Enforce** and watch the device get disconnected and re-authenticated into the remediation VLAN. Fix the laptop and watch it return.
6. Write an access rule with MDM compliance *Non-compliant, Not in any MDM or Unknown* set to Reject. Test it with **Test a session**.

Cheat sheet KEEP THIS

- **Posture**: agent on Windows and Linux, **802.1X only**, **Enterprise**. States: Compliant, Non-compliant, Not reporting, Awaiting first report.
- Roll-out: **remediation VLAN → Monitor policy → enroll → watch Overview → Enforce.**
- Agent sends only pass/fail. Fleet token limited to the device's own adapters.
- **MDM**: Intune and Jamf Pro, matched by **MAC** or by **certificate serial or Azure AD device ID**. Stale = **unknown**. Worst answer wins.
- Re-authorize on change: **off** by default, first sync never disconnects, limited per sync, hour and device.
- Posture has the last word over an access rule that admits. Add a Reject rule to refuse instead.

- Read more: Endpoint posture (https://demo.authcha.cloud/docs/posture), MDM compliance (https://demo.authcha.cloud/docs/mdm).