ScrambleID Architecture: One Identity Fabric Across Eight Surfaces

Fundamentals / Last updated 2026-06-11 / https://www.scrambleid.com/learn/scrambleid-architecture-identity-fabric

In one sentence: ScrambleID is an **identity fabric**: a shared set of cryptographic primitives and telemetry that every surface reuses (web, voice, people, and frontline for humans; agent, machine, bot, and workload for non-humans), so attackers cannot "route around" one strong control by switching surfaces.

TL;DR (canonical)

- The core join keys are **SUID** (user) and **ZID** (device).
- The core challenge primitive is a **Dynamic Identifier (DID)** wrapped as a signed **QID** for QR flows.
- The core safety mechanism is **binding**: origin binding (WebAuthn) and session/channel binding (websocket/call session).
- Every surface emits consistent events so monitoring and policy can be shared.

The primitives (what is truly shared)

Primitive	Meaning	Used by
SUID	canonical user identifier	all eight surfaces
ZID	canonical device identifier	the human surfaces (Web, Voice, People, Frontline); non-human surfaces register key-pairs
Device key	private key stored in secure hardware	mobile, workstation, terminal, non-human runtime
DID	short-lived, single-use challenge bound to a session	Web, Voice, People, Frontline
QID	signed QR envelope carrying DID + validation metadata	Web, Frontline, some People flows
WebAuthn assertion	origin-bound public-key proof	Web, Frontline
JWT client assertion	non-human proof of key possession	Agent, Machine, Bot, Workload

If a vendor has different primitives per surface, you do not have an identity fabric - you have a product bundle.

Where ScrambleID sits in the stack (vs IdP/MFA/IDV/PAM)

Teams often ask "is this an IdP?" or "is this identity verification?" The clean answer: ScrambleID is an **omnichannel authentication and assurance** layer that can sit alongside (or in front of) your existing identity systems.

Category	What it typically owns	ScrambleID role	You still need
Identity Provider (IdP)	directory, SSO federation, sessions	can act as an IdP for certain apps or as a step-up / verifier	directory & HR provisioning (often)
MFA app / OTP	factor prompts, device enrollment	replaces weak prompts with bound proofs (WebAuthn/DID/QID)	device management policy (enterprise)
IDV / KYC	proofing / doc checks, sanctions, AML	uses IDV outputs as provenance inputs, not as the day-to-day auth rail	regulated identity proofing when required
PAM	privileged session brokering, vaulting	gates privileged actions with step-up + dual control	vault + session recording (if needed)
Contact center scripts	caller handling flows	provides caller authentication primitives that replace KBA	routing and agent tooling

Common deployment patterns

- **Augment existing IdP:** keep Okta or Microsoft Entra ID for SSO, use ScrambleID for phishing-resistant step-up and cross-channel confirmation.
- **Contact center hardening:** deploy ScrambleID Voice to stop vishing, then extend to ScrambleID Web and ScrambleID People for consistent policy.
- **Assurance overlay:** use ScrambleID provenance + telemetry to unify fraud, IAM, and support operations.

Component model (conceptual)

```
flowchart LR
  subgraph Surfaces
    Web[ScrambleID Web]
    Voice[ScrambleID Voice]
    People[ScrambleID People]
    Frontline[ScrambleID Frontline]
    Agent[ScrambleID Agent]
    Machine[ScrambleID Machine]
    Bot[ScrambleID Bot]
    Workload[ScrambleID Workload]
  end

  subgraph Core
    Identity[Identity Core\nSUID/ZID + policies]
    Challenge[Challenge Rails\nDID/QID + binding]
    Certs[Cert & Key Distribution\nJWKS + thumbprints]
    Telemetry[Audit & Telemetry\nEvents + exports]
  end

  Surfaces ⟶ Identity
  Surfaces ⟶ Challenge
  Challenge ⟶ Certs
  Identity ⟶ Telemetry
  Challenge ⟶ Telemetry

  subgraph Controls
    Overwatch[Overwatch\nRisk decisions]
    XFactor[XFactor\nStep-up chains]
    Lockstep[Lockstep\nDual control]
    CoT[Circle of Trust\nTrust signals]
  end

  Telemetry ⟶ Overwatch
  Identity ⟶ XFactor
  Identity ⟶ Lockstep
  Identity ⟶ CoT
```

```
Overwatch ⟶ XFactor
Overwatch ⟶ Lockstep
```

The Controls plane (Overwatch, XFactor, Lockstep, Circle of Trust) is in development; the diagram shows the target architecture, not what ships today.

Binding (why the system is phishing-resistant)

Binding is what prevents "approve from anywhere".

- **Web:** WebAuthn binds the proof to the relying party origin.
- **Cross-device web login:** the DID is bound to the initiating session (often via a websocket channel id) and a typed code binds human intent.
- **Voice:** the DID confirmation is bound to a specific live call session id.

If binding fails, the system fails closed for that session and logs the mismatch.

Session binding cryptography (canonical specification)

A DID or QID alone is just a number. What makes a confirmation safe to act on is **binding**: the cryptographic guarantee that this specific confirmation came from a specific user, on a specific enrolled device, for a specific session and intent, within a specific time window.

Without binding, an attacker who intercepts a DID can replay it. With binding, the same DID from session A is invalid in session B even if the attacker holds the value.

This section is the authoritative reference for how binding works across every ScrambleID surface. Articles that invoke "session binding" link back here.

What binds together

Three things travel together in every confirmation and are validated atomically by the verifier:

1. **Proof of identity.** A signature from a device-bound private key (WebAuthn assertion for browsers, ZID device key for app-confirmed flows).
2. **Proof of session.** A server-issued, opaque, single-use session token that the verifier created when the session began. The session token is unguessable (cryptographically random, ≥128 bits of entropy) and tied to the verifier's session record.
3. **Proof of intent.** A structured description of what the user is approving: the action verb, the target resource, the requesting application, the audience for any token that gets minted, and an absolute expiry. Intent is included in the signed payload so a confirmation for one action cannot be reused for a different action.

If any of the three is missing, mismatched, or expired, the verifier rejects the confirmation. The check is atomic: there is no "two of three" partial credit.

Confirmation payload structure

Every device-signed confirmation carries this structure:

```
{
  "did": "DID-...",
  "session_id": "sess_<opaque>",
  "intent": {
    "action": "authenticate",
    "resource_id": "app:billing-portal",
    "rp_origin": "https://billing.example.com",
    "audience": "https://api.example.com",
    "issued_at": 1735776000,
    "expires_at": 1735776090
  },
  "device_proof": {
    "zid": "ZID-...",
    "key_id": "kid_...",
    "alg": "ES256"
  }
}
```

The device signs the entire payload with the private key tied to `key_id`. The signature is fresh per confirmation. There is no reusable shared secret anywhere in this exchange.

Expiry is **absolute**, not relative. The verifier rejects any confirmation where `now > expires_at + clock_skew_tolerance` or `now < issued_at - clock_skew_tolerance`. Recommended skew tolerance: 5 seconds. Production deployments must run NTP or an equivalent time-sync service.

The atomic verification check

When a confirmation arrives at the verifier, all of the following must hold or the confirmation is rejected:

1. **Signature validates** against the device's registered public key for `key_id`.
2. `zid` **is enrolled** to the user (`suid`) the session expects, in the active state (not revoked, not retired, not in a grace window for other purposes).
3. `session_id` **matches** an open server-side session record.
4. `did` **matches** the challenge bound to that session.

5. **Session has not been consumed.** A confirmation can only succeed once per session. The verifier marks the session consumed atomically with the success response.
6. **Intent is unchanged** from what was registered when the session was issued. The verifier compares fields, not just hashes.
7. **Time window is valid** within the configured skew tolerance.
8. **Origin or call-context binding** is satisfied: for web, the WebAuthn assertion's RP ID matches; for voice, the IVR-supplied call session id matches the session record; for people verification, the verifier's session matches the originator.

A failure on any of these is an audit event and contributes a signal to the risk layer.

Per-channel binding mechanics

Every channel uses the same primitives. What changes is what `session_id` represents and how `intent` is populated.

Web (browser, same device): Session binding is to the WebAuthn assertion's origin and challenge. The browser-issued challenge IS the session token; the relying party validates the origin and signature in one step.

Web (cross-device QR(DID)): Session binding is to a server-side session record created when the relying party initiated the QR flow. The session id travels in the QID payload (signed). The browser holds an open channel (websocket or long-poll) keyed on the same session id. The user types a short numeric code on the app to bind human intent. On success, the verifier emits the result only on the open channel keyed to the originating session.

Voice (IVR): Session binding is to the live call session id supplied by the IVR platform (Twilio CallSid, Genesys interaction id, NICE CXone contact id). The verifier records `call_session_id` as part of the session record at session-start. The DID confirmation must reference the same `call_session_id`. On success, the verifier issues an intercept callback to the IVR which redirects the live call. A confirmation for call A cannot redirect call B because the session record cross-checks `call_session_id`.

People (verifier-initiated Trust Check): Session binding is to the Trust Check session id created by the verifier. The presenter's app must include the same session id in the signed share payload. The renderer on the verifier's side validates that the signed card's session id matches the Trust Check it initiated.

Desktop (workstation login): Session binding is to the desktop client's local session handle plus the device's enrolled `zid`. For shared workstations, the session record includes a `workstation_id` so a confirmation from device A cannot resolve a session opened on device B.

M2M (machine-to-machine): Sessions are not typed-by-user the same way. Binding is to a JWT client assertion (RFC 7523), an audience claim equal to the token endpoint, and a unique `jti`.

Sender-constrained tokens (mTLS via RFC 8705 or DPoP via RFC 9449) bind issued access tokens to the requesting client's key thereafter.

What this defeats and what it does not

Session binding defeats:

- **Approve-anywhere replay.** A confirmation captured from session A cannot be used to complete session B.
- **AiTM relay.** An attacker who proxies the user's browser session sees the QID, but cannot complete the confirmation because the confirmation must originate from the user's enrolled device, signed with the device's private key, and the verifier's session record is bound to the legitimate browser channel that the relying party opened.
- **Cross-channel session hijack.** A confirmation from a voice call cannot resolve a web login session, and vice versa, because the `intent.action` and `session_id` differ.
- **Quishing.** A QR copied to a fake page is valid only if scanned, but the confirmation flows back to the original session via the legitimate verifier, never to the attacker. The attacker holds a screenshot, not a session.

Session binding does **not** defeat:

- **A fully compromised device.** If an attacker has the user's device and the device's PIN/biometric (or the OS-level key protection has been bypassed), the attacker can produce signatures the verifier accepts. Mitigation lives in device key lifecycle (next section): rapid revocation on suspected compromise.
- **Social engineering of the user.** A determined adversary on the phone with the user can pressure them to approve the legitimate session of the attacker's choice. The user is approving real intent that the verifier renders accurately. Mitigation lives in the step-up layer (XFactor, in development: anti-fatigue, intent display, friction on high-risk actions) and the risk layer (Overwatch, in development: signal correlation that flags suspicious patterns).
- **Server-side compromise.** If the verifier is compromised, the session records can be manipulated. Mitigation lives outside this section: hardened key storage, audit logs, separation of duties for verifier infrastructure.

These limits are inherent. Honest acknowledgment of what cryptographic binding can and cannot do is part of why this design works in production.

Implementation guidance: session binding

For engineers building against this model:

- **Generate session ids cryptographically.** At least 128 bits of entropy from a CSPRNG. Never reuse, never derive from user input, never embed user identifiers.

- **Persist session records server-side.** Memory-only state breaks under load balancing. Use a strongly consistent store (a transactional database or Redis with appropriate guarantees). Single-use enforcement requires atomic `MARK_CONSUMED` semantics; do this in a single transaction with the success response.
- **Validate every field.** Do not trust the client to send the right session id; the verifier owns the session record and matches against it. Reject on mismatch, do not "best effort" repair.
- **Race conditions matter.** Two concurrent confirmation attempts for the same session must result in one success and one rejection, never two successes. Use database constraints or Redis SETNX, not application-level checks.
- **Clock skew must be bounded.** A 30-second confirmation TTL with 60 seconds of clock skew is functionally a 90-second TTL. Run NTP, monitor drift, alert if drift exceeds 1 second on production verifier hosts.
- **Log binding failures separately from successes.** A binding failure is a signal: either an attack attempt or a real bug. Either way, the SOC needs to see them.

Recommended TTLs

These are starting defaults. Tune per use case based on UX measurement and threat model.

Channel	Session TTL	DID TTL	Notes
Web QR(DID), same-room scan	60 seconds	60 seconds	Tight TTL keeps replay window minimal
Web QR(DID), cross-device messaging link	5 minutes	60 seconds session-extended on user activity	Link can travel; DID is per-attempt
Voice IVR	90 seconds	90 seconds	Allows for read-aloud + app open + confirm
People (in-person)	60 seconds	60 seconds	Live session, tight
People (messaging link)	24 hours	60 seconds session-extended on join	Link can travel; consent is single-use
Desktop login	30 seconds	30 seconds	Workstation already in trusted physical context
WebAuthn (any)	Per RFC, browser-managed	N/A (challenge-driven)	The browser owns the timing here

The article on Dynamic Identifiers (DID/QID) covers DID lifecycle in more depth. This section is the authoritative source for *how the binding works*; that article is the authoritative source for *what a DID is*.

Device key lifecycle (canonical specification)

"Device-bound keys" appear throughout the corpus. This section specifies the full lifecycle: enrollment, storage, rotation, revocation, recovery, and compromise handling. Articles that invoke device-bound keys link back here.

Where keys live

The device's private key is generated and stored inside hardware-backed secure storage. The exact mechanism depends on the platform:

- **iOS:** Secure Enclave. Keys generated via the Apple Secure Enclave coprocessor; private key material never leaves the SE. Biometric or passcode unlock is required to invoke the key.
- **Android:** Hardware-backed Keystore via the Trusted Execution Environment (TEE) or, where available, a dedicated security chip (StrongBox on supported devices). Biometric or PIN gates key use.
- **Windows:** TPM 2.0. Windows Hello for Business issues platform credentials backed by the TPM. The key cannot be exported.
- **macOS:** Secure Enclave (Apple Silicon) or T2 chip on Intel Macs. Touch ID or password gates key use.

In all cases:

- The private key cannot be extracted by software running on the device, including a privileged process or the OS itself.
- A signature operation requires user verification (biometric, PIN, or password) per the configured policy.
- The OS attests the key's hardware backing during enrollment so the verifier can record the assurance level.

This is what makes a confirmation a meaningful proof of device possession. The device must be physically present and unlocked by the user.

Enrollment

Enrollment is the moment the verifier records `(suid, zid, public_key, attestation)`. It happens once per device per user.

A typical enrollment flow:

1. The user authenticates to the verifier through an existing trusted path (e.g., a managed onboarding flow with identity proofing, or an in-app step-up from a previously enrolled device).
2. The user's app generates a new key pair inside the secure hardware. The platform produces an attestation that proves the key is hardware-backed.

3. The app sends the public key, the attestation, and a fresh nonce signed with the private key to the verifier.
4. The verifier validates the attestation against the platform manufacturer's root (Apple, Google, Microsoft), records the public key against the user's `suid` with a fresh `zid`, and notes the assurance level.
5. The device is now enrolled and can produce confirmations the verifier will accept.

Enrollment is the most security-sensitive moment in the lifecycle. If enrollment is weak (the trusted path was easy to social-engineer), every subsequent confirmation inherits that weakness. Production deployments should require identity proofing for first-device enrollment and step-up from an existing enrolled device for additional-device enrollment.

Rotation (planned)

Keys rotate when the user upgrades their device, when the OS-level key generation parameters change, or when policy requires periodic rotation.

The pattern:

1. The user's new device or new key pair completes enrollment as a fresh `zid`.
2. Both the old `zid` and the new `zid` are valid for an **overlap window** (recommended default: 24-72 hours, policy-configurable up to 14 days).
3. During the overlap window, the user can complete confirmations from either device. The verifier accepts both.
4. After the overlap window, the old `zid` is automatically retired. Confirmations from the old device are rejected.

The overlap window matters because:

- Users move at human pace, not technical pace. They may not migrate their auth flows the moment they unbox a new phone.
- In-flight confirmations (an open IVR call, an open web session) need to complete with the key the user actually has in their hand at that moment.
- A hard cutover during a peak traffic window causes confirmation failures to spike, which a SOC must distinguish from an actual attack.

Recommended monitoring during rotation:

- Track the percentage of confirmations using the old `zid` per day. The number should fall toward zero as the overlap window closes.
- Alert if a meaningful fraction (>5%) of confirmations still use the old `zid` close to the end of the overlap window. This usually indicates a stuck client or a user who never received the new device.
- Distinguish "old `zid` rejected after overlap" (expected) from "old `zid` accepted post-overlap" (a bug; should never happen).

Revocation (compromise or loss)

Revocation is the immediate retirement of a `zid` outside the planned rotation flow. The trigger is suspected compromise: a lost device, a stolen device, a reported account takeover attempt, a risk-layer signal that flags abnormal device behavior, or an admin action initiated by the user, the user's IT team, or the SOC.

Revocation has two distinct timing requirements:

Decision latency: how long it takes the verifier to record the revocation. Recommended target: under 60 seconds from trigger to recorded state.

Propagation latency: how long it takes every active session (web, voice, desktop, M2M consuming tokens minted by this device) to see the revocation. Recommended target: under 60 seconds for online sessions, with a hard ceiling at the maximum cached-session TTL configured in your environment.

Propagation is the harder of the two. The verifier records the revocation immediately, but downstream systems may have cached state. Implementation patterns:

- **Token introspection on each request** for high-risk APIs. Every request hits the verifier; revocation is honored within one round-trip.
- **Short token TTLs (≤5 minutes) with refresh** for medium-risk APIs. Revocation is honored at the next refresh.
- **Long token TTLs with revocation event push** for low-risk APIs that cannot afford the introspection round-trip. The verifier emits revocation events to relying parties, which invalidate cached state. This pattern requires a guaranteed-delivery event channel and at-least-once handling on the receiver.

The right pattern depends on the latency budget for each surface. For high-stakes flows (admin actions, payouts, M2M for sensitive resources), introspection-on-each-request is the safe default. For low-stakes flows, eventual consistency in seconds is acceptable.

In-flight handling during revocation:

- An open web session whose `zid` is revoked should be terminated at the next backend roundtrip. Show the user a clear "session ended for security; please re-authenticate" message; do not silently log them out without explanation.
- An open voice call whose `zid` was confirmed pre-revocation can complete the current task at the IVR's discretion (a partially-completed call may need to drop, a fully-verified call may finish). Future confirmations on the same call require re-verification.
- An in-flight M2M token request signed by a revoked key fails immediately; the requesting workload must surface this as a fatal error and trigger its own incident response.

Recovery

Recovery is what happens after revocation when the user needs to re-establish authentication. There are two paths.

Replacement-device recovery (warm path). The user obtains a new device and re-enrolls. The trusted path for re-enrollment is the most security-critical decision in the recovery design:

- If the user has another enrolled device, that device authorizes the new enrollment via XFactor (step-up from the existing enrolled device). This is the strongest recovery path.
- If the user has no other enrolled device, recovery requires identity proofing of the user (real-time video verification, government-ID scan, supervisor approval) before a new `zid` is issued.
- Recovery should never fall back to KBA. KBA is exactly the attack surface being eliminated.

Account-recovery (cold path). The user has lost all enrolled devices and any other recovery factor. This is the rarest and highest-risk path. Production deployments should treat cold-path recovery as a manual, audited operation with mandatory dual approval. The decision tree lives in the Recovery and Fallback Playbook.

Compromise scenarios

What if the device key itself is compromised? In practice, key compromise without device compromise is rare because the key never leaves hardware. The realistic scenarios:

Lost or stolen device, attacker has the device but not the unlock factor. The key cannot be used without the biometric or PIN. Time-to-detect is the user reporting the loss. Mitigation: revocation as above, force re-enrollment.

Lost or stolen device, attacker has the unlock factor (shoulder-surfed PIN, lifted fingerprint, coerced biometric). The key can be used until revoked. Time-to-detect is the user noticing or the risk layer flagging anomalous behavior. Mitigation: revocation, plus risk-layer tuning to catch anomalous patterns (new geography, new IP block, sudden velocity).

Device root or jailbreak with attestation bypass. The platform attestation we relied on at enrollment is now bypassable. The attacker can extract or use the key. Mitigation: periodic re-attestation of enrolled devices, refusing to issue new sensitive sessions to devices that fail re-attestation.

Platform-level vulnerability (Secure Enclave / TEE bypass). The hardware boundary we trusted has been broken. This is rare but historically has happened. Mitigation: vendor-issued advisories trigger forced re-enrollment for affected device generations.

For each scenario, the response is the same shape: detect, revoke, re-enroll through a stronger trusted path. The corpus does not pretend any of this is a happy path. It is incident response, and a deployment that treats it as such recovers cleanly.

Implementation guidance: device keys

- **Enroll public keys against `(suid, zid)` with attestation metadata.** Record the assurance level (hardware-backed, software-backed, attestation chain validated). Cite this metadata in audit logs.
- **Track rotation overlap state explicitly.** A `zid` has states: `pending`, `active`, `rotating`, `revoked`, `retired`. Transitions are audit events.
- **Make revocation atomic across the verifier's data plane.** If revocation lands on one verifier replica but not another (eventual consistency in your data store), an attacker can briefly use the revoked key. For high-stakes deployments, write through a synchronous primary and read from the same primary for revocation checks.
- **Provide a self-service revocation surface.** Users discovering loss should not have to call support to revoke. A web button (gated by a different enrolled device or by identity proofing) is faster.
- **Run a periodic compromise-detection sweep.** Inactive devices, devices with unusual usage patterns, devices reporting OS versions with known critical CVEs: surface them for review.

Recommended SLAs

These are starting targets. Tune per use case.

Operation	Recommended target
Enrollment latency, normal path	< 30 seconds end-to-end
Re-enrollment from existing device, with XFactor	< 60 seconds
Cold-path recovery with identity proofing	Hours to a business day; this is a feature, not a bug
Revocation decision latency	< 60 seconds from trigger
Revocation propagation, online sessions	< 60 seconds
Revocation propagation, M2M tokens via introspection	One round-trip
Revocation propagation, M2M tokens via TTL refresh	≤ 5 minutes
Rotation overlap window	24-72 hours default, up to 14 days configurable

These SLAs assume a healthy verifier and network. Plan separately for degraded modes; see Overwatch fail-safe guidance.

Certificate and JWKS distribution (why QID can rotate safely)

Signed QR challenges only work if scanning devices can validate signatures during rotation.

A typical pattern:

- QID includes a **certificate thumbprint**.
- device fetches the public certificate by thumbprint once (cacheable).

- subsequent scans verify locally.

This is what makes large-scale rotation possible without breaking flows.

How each surface reuses the fabric

Web

- emits DID/QID and runs websocket-bound confirmation
- optionally runs WebAuthn
- returns SAML/OIDC assertions

Voice (IVR)

- the registered-number fast path sends a push to the enrolled device; the IVR speaks a DID as the fallback
- the app confirms with device-bound keys and the call is intercepted to success
- emits the same session, did, and outcome events

People

- verifier starts a Trust Check
- presenter consents and shares a one-time card view
- attributes are rendered with provenance

Frontline

- shared-workstation and point-of-sale login uses device keys or WebAuthn, with per-person attribution instead of shared PINs
- the same session and policy model covers kiosks, clean rooms, and counters

Agent

- AI agents authenticate with asymmetric keys (JWT client assertions), never static API keys
- access is scoped per agent and per task, with the same event taxonomy for audit

Machine

- service-to-service calls prove key possession with short-lived JWT assertions
- tokens can be sender-constrained where replay is high-risk

Bot

- RPA bots, scheduled jobs, and automations replace standing service-account credentials with the same short-lived assertions
- each bot gets its own identity, scope, and audit lineage

Workload

- containers, functions, and VMs prove what they are at runtime instead of holding long-lived instance credentials
- the same issuance, revocation, and telemetry rails apply

Where trust and risk fit

Trust and risk are not separate systems; they sit on top of shared telemetry and identities. The four components below are in development; this is the design they're being built to.

- **Overwatch** correlates events and triggers actions.
- **XFactor** adds step-up chains.
- **Lockstep** adds multi-party approvals.
- **Circle of Trust** adds trust context (enterprise tiers, verified brands, personal edges).

What to ask in an architecture review

- How are confirmations bound to sessions and origins?
- What is the certificate rotation story for signed QR artifacts?
- What are the canonical identifiers across surfaces?
- What is the event schema, and can it be exported to a SIEM?
- Where are weak fallbacks (OTP, KBA, email links) and can we disable them?

Key Takeaway

ScrambleID's architecture uses four core identifiers (SUID for users, ZID for devices, DID for session challenges, QID for QR envelopes) unified across all eight surfaces. The identity fabric means the human surfaces (web, voice, people, frontline) and the non-human surfaces (agent, machine, bot, workload) share the same proof primitives, telemetry schema, and policy engine, enabling cross-surface correlation and consistent assurance levels.

FAQ

What is an "identity fabric"?

A shared set of identities, cryptographic primitives, and telemetry reused across surfaces instead of building separate authentication stacks per surface.

What is the minimum set of primitives ScrambleID reuses?

SUID/ZID identities, DID/QID challenge rails, binding mechanisms, and unified telemetry.

Why does reuse matter?

Because it reduces "dark corners" where attackers switch channels to find weaker controls.

References (public)

- WebAuthn specification (W3C): https://www.w3.org/TR/webauthn/
- NIST Digital Identity Guidelines (SP 800-63-4): https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-4.pdf

Related reading

- Dynamic Identifiers (DID/QID)
- Overwatch: Risk Engine
- XFactor Step-Up