SECURITY & AUTH PATTERNS · MODULE 6

Passkeys with the WebAuthn Extension

Lesson Guide · Lab / Exam Guide

LearnWisej

Lesson Guide

WebAuthn relying party design, enrollment from a verified account, registration and authentication ceremonies with the Wisej.NET WebAuthn extension, server-side validation of every assertion, a concurrency-safe credential store and account recovery. What this means for a developer securing the sign-in, identity and authorization paths of a production Wisej.NET app — and how to read this module.

Production focus

WebAuthn relying party design, enrollment from a verified account, registration and authentication ceremonies with the Wisej.NET WebAuthn extension, server-side validation of every assertion, a concurrency-safe credential store and account recovery.

1. The production problem

AccessOps partners already sign in with a local account (Module 4) or their own identity provider (Module 5). A passkey adds a credential that cannot be phished or replayed on another site, but only if the server checks everything the browser hands back. The Wisej.NET WebAuthn extension runs the browser ceremonies and verifies one signature; the challenge, origin, relying party, owner, counter and credential store are AccessOps code, and each is a place where a sign-in can be wrongly accepted.

2. Key concepts

Concept	Why it matters
relying party ID	The domain, without scheme or port, that a credential is scoped to; changing it later forces every user to register again.
origin allowlist	The exact scheme, host and port values accepted in `ClientData.Origin`; anything else is refused even with a valid signature.
credential ID and public key	What the server stores per credential; the private key and any biometric sample never leave the authenticator.
user handle	An opaque, stable account identifier sent as the user entity's `id` at registration and returned as `UserHandle`.
one-time challenge	Random bytes per ceremony, bound to its type, account and expiry, compared with `ClientData.Challenge` and consumed so a captured response cannot be used twice.
assertion policy	The checks around `WebAuthn.Validate`, which proves only that the signature matches a stored public key.
compare-and-swap commit	One conditional update that applies a verified sign-in only if the row still has the version that was read, returning Updated, Conflict or NotFound.

3. Lesson narrative

One relying party, exact origins, and what the server keeps

A WebAuthn credential belongs to a relying party ID, a bare domain such as `partners.accessops.example`. An origin is narrower: `https://partners.accessops.example`, scheme, host and port included. AccessOps registers with an RP ID equal to the partner sign-in host and keeps an allowlist of exact origins, not a pattern. For the lab, the documented secure context is `https://localhost` on the lab port with the RP ID `"localhost"`; WebAuthn refuses plain HTTP anywhere else. The host choice matters in this extension: `WebAuthn.GetAsync` passes no RP ID to the browser, so the browser uses the page's own domain and a credential registered for a parent domain may not be offered on a sub-host. Register and sign in on the same host.

The server keeps, per credential: the credential ID, the serialized public key (`JSON.Stringify(publicKey)`, restored with `new PublicKey(json)`), the algorithm, the RP ID, the account and its user handle, the last signature counter, the `BackupEligibility` flag, a label and timestamps. AccessOps generates the user handle, never an e-mail address, and keeps it ASCII: the client sends the `id` one byte per character and decodes `UserHandle` as UTF-8. The application never receives a private key, a fingerprint or a face image.

Two ceremonies from server-side C#

The package is `Wisej-4-WebAuthn` 4.1.7, namespace `Wisej.Ext.WebAuthn`. It is one static class with four methods: no components, events, store or challenge management. `IsUserVerifyingPlatformAuthenticatorAvailableAsync` reports Windows Hello, Touch ID or Face ID; a

security key can still work when it returns false. Registration is `CreateAsync` with seven required arguments: a standard base64 challenge of at least 16 random bytes, a `RelyingParty`, a `PublicKeyCredentialUserEntity`, the algorithms, an `AuthenticatorSelectionCriteria`, a timeout in milliseconds and an `AttestationConveyancePreference` (`None` in AccessOps). Offer only `COSEAlgorithmIdentifier.ES256` and `RS256`: `Validate` returns false for the other ECDSA curves and throws for EdDSA. How this package serializes the attachment value is unverified, so AccessOps does not rely on it to restrict authenticators.

Enrollment starts only from a verified account session: the partner signed in recently through Module 4 or Module 5 and the current grant passes. `CreateAsync` performs no attestation check and does not fill the authenticator flags or the RP ID hash, so the response proves little about the person; the session does. AccessOps compares the challenge, checks that `ClientData.Type` is `"webauthn.create"` and the origin is on the allowlist, then stores `AuthenticatorData.PublicKey`. Both async calls throw an `Exception` with the browser's message on cancel, timeout or mismatch, so they sit in `try` / `catch` and the screen shows fixed text with `allowHtml: false`: `AlertBox` and `MessageBox` default it to true.

Validate every assertion on the server

Sign-in calls `GetAsync` with a new challenge. Passing `null` for the allowed credentials lets the partner pick a discoverable passkey, and `UserHandle` then names the account; an account-first form passes that account's `PublicKeyCredentialDescriptor` list instead. The response does not say which credential signed, so AccessOps tries the keys stored for that user handle under this RP and checks everything else itself:

```
var r = await WebAuthn.GetAsync(ticket.Base64, null, 60000);
bool fresh = _challenges.TryConsume(ticket.Id, r.ClientData.Challenge);
var flags = AuthenticatorFlags.Read(r.AuthenticatorData.Base64); // 4.1.7: UserVerified
reads a reserved bit
var cred = _store.ForUserHandle(Rp.Id, r.UserHandle).FirstOrDefault(c ⇒
    WebAuthn.Validate(c.Key, r.AuthenticatorData.Base64,
                      r.ClientData.Base64, Es256Signature.ForValidate(r.Signature)));
bool ok = fresh && cred ≠ null
    && r.ClientData.Type == "webauthn.get"
    && AllowedOrigins.Contains(r.ClientData.Origin)
    && r.AuthenticatorData.RPIDHash.SequenceEqual(RpIdHash)
    && flags.UserPresent && flags.UserVerified
    && CounterPolicy.Accepts(cred.SignCount, r.SignatureCounter);
if (!ok) { _audit.PasskeyDenied(ticket.Id); return; }
var outcome = _store.CommitUse(cred, r.SignatureCounter); // Updated | Conflict | NotFound
```

The challenge is consumed first, so a failed attempt burns it too. `GetAsync` sends no user-verification option, so the browser default applies and user verification must be checked rather than assumed. Two defects in the 4.1.7 extension decide how. Its client script sets `AuthenticatorData.UserVerified` from bit 1 of the flags byte, a reserved bit that is always zero, so the property is false even when a Windows Hello prompt verified the person; AccessOps therefore reads the flags byte itself from `AuthenticatorData.Base64` (byte 32 of the signed authenticator data: bit 0 user present, bit 2 user verified, bit 3 backup eligible) and uses the property only as a cross-check for presence. The second defect is in `WebAuthn.Validate`: it reads the DER-encoded ES256 signature by fixed offsets, so a genuine

signature whose `r` or `s` value is shorter than 32 bytes, roughly one in 128, fails verification; AccessOps re-encodes both integers as exactly 33 bytes before calling Validate, which changes neither value nor what the signature proves, and a malformed signature still fails. Both workarounds live in two small classes, `AuthenticatorFlags` and `Es256Signature`, and should be removed when a fixed extension ships: both defects were reported upstream on 2026-11-10 and are fixed in the extension's source; the fix reaches a package version later than 4.1.7. The counter policy is written down: many authenticators always report zero, so zero followed by zero is accepted, while a non-zero counter that does not increase is refused and recorded as an anomaly. The extension exposes `BackupEligibility` but no backup-state flag, so AccessOps records eligibility and makes no decision that depends on whether a passkey is currently synced. Only when the commit returns Updated does the Module 4 hand-off run: park the verified account, reload, call `Application.RefreshSessionId()` on the load, set `Application.User` and create exactly one login grant that every protected read still checks.

A credential store that keeps revocation final

The `PasskeyCredentialStore` is AccessOps's own class over one table with a unique constraint on RP ID and credential ID, so a credential cannot be enrolled twice or under a second account. Its commit is one conditional statement: update the counter and last-used time where the RP, the credential ID, the owner, the row identity and the version all match what the verifier read. One affected row is Updated. Zero rows is classified, never retried blindly: NotFound if the row is gone, otherwise Conflict. A credential deleted while a sign-in was in flight stays deleted, and one re-enrolled with the same ID gets a new row identity, so a stale update cannot land on it. Reading, comparing in C# and writing in a second unguarded step is exactly the race to avoid.

The review page lists, renames and deletes only the signed-in user's own credentials, taking the owner from the session principal, never from a grid cell. Recovery after a lost device is a written plan: a second enrolled passkey, or a Module 4 or Module 5 sign-in plus a recent-verification step before a new credential is enrolled, the lost credential deleted, and every change written to the account history that support can audit. Recovery never bypasses the verified-session rule.

Security habit

Keep the identities apart and decide on the server. The host principal, the Wisej.NET application principal and the effective Windows token are three different facts, and none of them is a permission. Every protected read checks the current grant, the tenant and the record at the service boundary before the repository runs, and a hidden button, a role claim or a signed-out tab proves nothing until that check denies.

Learning objectives

By the end of this module you'll be able to:

- ✓ Choose the relying party ID and exact allowed origins for AccessOps and explain what the server stores — credential ID, public key, user handle — and what it never receives.
- ✓ Run passkey registration and authentication from server-side C# with the Wisej.NET WebAuthn extension, starting enrollment only from a verified account session.

- ✓ Validate every assertion on the server — one-time challenge, origin, RP ID, user presence and verification, credential owner and signature — and apply a documented counter and backup policy.
- ✓ Build a credential store whose updates use an atomic compare-and-swap so a deleted or revoked credential stays revoked, and plan device-loss recovery.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.

Hands-on lab

Add passkeys to AccessOps with the Wisej-4-WebAuthn extension on a stable HTTPS hostname or a documented localhost secure context: check browser support, sign in as account A through the Module 4 or Module 5 flow and enroll a labeled credential with CreateAsync, then sign out and authenticate with GetAsync and server-side Validate, map the local account and recheck the current grant before showing records. Store credentials in a PasskeyCredentialStore keyed by RP ID and credential ID with a unique constraint and a compare-and-swap commit that returns Updated, Conflict or NotFound, add a credential review page where a signed-in user renames or deletes only their own credentials, and reject account A's credential for account B, an unapproved origin, a replayed challenge and a cancelled prompt with no new grant, then read a credential twice in a store integration test and show a stale update and a delete-before-commit fail.

Starting point

Open the cumulative AccessOps solution from Module 5: the partner route already has the local sign-in with its `GrantStore`, the OpenID Connect federation and the `ApplicationPrincipal` hand-off that rotates the session ID on a page load. Add the `Wisej-4-WebAuthn` 4.1.7 package to the AccessOps project and use the two partner fixtures, account A and account B, in the same tenant.

Suggested approach

- ✓ Fix the relying party first: write down the RP ID, the exact allowed origins and the secure context (the HTTPS host, or `https://localhost` with RP ID `"localhost"`), and run registration and sign-in on that same host.
- ✓ Build the `PasskeyCredentialStore` table and its conditional commit before any UI, and write the store integration test that reads a row twice, commits once and then attempts the stale update and the delete-before-commit.
- ✓ Add enrollment to the signed-in account screen only: issue a one-time challenge bound to the account, call `WebAuthn.CreateAsync` with ES256 and RS256, check the challenge, type and origin, and store the labeled credential.
- ✓ Add "Sign in with a passkey" to the partner sign-in page: `WebAuthn.GetAsync`, consume the challenge, find the owner's key with `WebAuthn.Validate` (passing the ES256 signature through `Es256Signature.ForValidate`), check type, origin, RP ID hash, user presence and user verification from the signed flags byte (`AuthenticatorFlags.Read`, not the `UserVerified` property) and the counter, commit, then run the Module 4 grant hand-off.
- ✓ Run the five outcomes and the two store races, add the credential review page, and write the device-loss recovery plan with who approves a re-enrollment and where it is audited.

Required deliverables

- ✓ Passkey enrollment and sign-in with the WebAuthn extension

- ✓ Server-side assertion validation with one-time challenges
- ✓ PasskeyCredentialStore with atomic compare-and-swap commits
- ✓ Five outcomes: one valid sign-in and four rejections, with store race tests
- ✓ Credential review page and account recovery plan

Student review questions

- ✓ `WebAuthn.Validate` returned true for a response whose challenge was already consumed. Which check refuses it, and what would an attacker have needed to produce that response?
- ✓ A partner deletes a lost laptop's passkey while a sign-in with it is waiting on the commit. Walk through what the conditional update returns and why no grant is created.
- ✓ Your RP ID is the parent domain and the sign-in page moves to a sub-host. Why does this extension's `GetAsync` fail to find the credential, and what would you change?

Security habit

Keep the identities apart and decide on the server. The host principal, the Wisej.NET application principal and the effective Windows token are three different facts, and none of them is a permission. Every protected read checks the current grant, the tenant and the record at the service boundary before the repository runs, and a hidden button, a role claim or a signed-out tab proves nothing until that check denies.