PATTERN DI SICUREZZA E AUTENTICAZIONE · MODULO 5

Federazione con OpenID Connect

Guida della lezione · Guida al lab / esame

LearnWisej

Guida della lezione

OAuth contro OpenID Connect, l'authorization code flow con PKCE attraverso gli handler cookie e OpenID Connect di ASP.NET Core registrati in Startup.cs prima di app.UseWisej(), la convalida dei token, la mappatura dell'account per issuer e subject nella sessione Wisej.NET, il ripiego sul login locale e la disconnessione da entrambe le sessioni. Che cosa significa per uno sviluppatore che mette in sicurezza i percorsi di accesso, identità e autorizzazione di un'app Wisej.NET di produzione — e come affrontare questo modulo.

Focus di produzione

OAuth contro OpenID Connect, l'authorization code flow con PKCE attraverso gli handler cookie e OpenID Connect di ASP.NET Core registrati in Startup.cs prima di app.UseWisej(), la convalida dei token, la mappatura dell'account per issuer e subject nella sessione Wisej.NET, il ripiego sul login locale e la disconnessione da entrambe le sessioni.

1. Il problema in produzione

I partner di AccessOps hanno già un account presso la propria organizzazione, con la propria policy multifattore e il proprio processo per chi lascia l'azienda, quindi la route dei partner delega l'autenticazione a un provider OpenID Connect invece di mantenere una seconda password. Ora l'identità viaggia attraverso il browser dell'utente, dove ogni parametro può essere contraffatto, riprodotto o scambiato. Wisej.NET 4.1.7 non ha alcun componente di federazione, quindi AccessOps collega al proprio host gli handler standard di ASP.NET Core e consegna il loro risultato verificato alla sessione Wisej.NET.

2. Concetti chiave

Concetto	Perché è importante
OAuth contro OpenID Connect	OAuth autorizza un client a chiamare un'API protetta; OpenID Connect aggiunge uno strato di identità, un ID token che dice al client chi ha effettuato l'accesso.
authorization code + PKCE	Il browser trasporta solo un codice di breve durata; il server lo riscatta sul canale secondario con un code verifier che non ha mai lasciato il server, quindi un codice rubato è inutile.
state e nonce	Lo state lega il callback al tentativo del browser che lo ha avviato; il nonce lega l'ID token a quella stessa richiesta, quindi nessuno dei due può essere riprodotto in un altro accesso.
ID token e access token	L'ID token è una prova di identità per AccessOps stesso; l'access token appartiene al resource server per cui è stato emesso e non dimostra nulla su chi sta alla tastiera.
issuer + subject	La coppia convalidata `iss` e `sub` è l'unica chiave stabile di un account esterno; indirizzi email e nomi visualizzati possono essere riassegnati.
callback esatto, authority fissata	Il provider invia i codici solo a un redirect URI registrato, e l'handler carica chiavi ed endpoint solo da un'unica authority configurata, quindi nessuno dei due può essere dirottato.
handle di passaggio	Un valore casuale monouso in un cookie HttpOnly che permette al caricamento della pagina Wisej.NET di raccogliere un'identità convalidata; il browser non contiene mai un token.

3. Il filo della lezione

Un codice, due prove: il flusso con PKCE, state e nonce

Un partner fa clic su *Sign in with your work account*. La pagina di AccessOps chiama `IExternalSignIn.Begin()`, che porta la scheda all'endpoint ASP.NET Core ordinario `/account/signin`; quell'endpoint emette un challenge per `OpenIdConnectDefaults.AuthenticationScheme`. A questo punto l'handler crea tre segreti: uno state casuale, protetto e associato a un cookie di correlazione in questo browser; un nonce, conservato in un cookie del nonce; e un code verifier PKCE il cui hash SHA-256 viaggia verso il provider come challenge S256. Il partner effettua l'accesso presso il provider; AccessOps non vede mai la password.

Il provider riporta il browser a `/signin-oidc` con un codice e lo state. L'handler controlla prima che lo state si possa decifrare e corrisponda al cookie di correlazione di *questo* browser. Poi riscatta il codice presso il token endpoint sul canale secondario con il client secret e il verifier originale, dimostrando che chi riscatta è chi ha avviato il tentativo. Infine convalida l'ID token: firma rispetto alle chiavi pubblicate dal provider, issuer, audience, durata e nonce. Solo allora esiste un'identità.

Tieni separati i due token. L'ID token risponde a *chi ha effettuato l'accesso, presso quale provider, per questo client*. Un access token è indirizzato a un resource server, per esempio un'API della directory dei partner, e

AccessOps lo invia solo lì, dal server. Far accedere qualcuno a partire da un access token decodificato salta tutti i controlli descritti sopra.

Startup.cs: gli handler prima di app.UseWisej()

Un'applicazione Wisej.NET 4 è un'applicazione ASP.NET Core, quindi la federazione si configura in `Startup.cs` con il pacchetto `Microsoft.AspNetCore.Authentication.OpenIdConnect` 10.0.14: `AddAccessOpsOpenIdConnect()` registra gli handler, e `MapAccessOpsAccountEndpoints()` chiama `app.UseAuthentication()` e mappa gli endpoint dell'account, tutto prima di `app.UseWisej()`, così che a `/signin-oidc` e `/signout-callback-oidc` risponda il middleware di autenticazione.

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie()
    .AddOpenIdConnect(o ⇒
    {
        o.Authority = oidc["Authority"];        // fissata, discovery su HTTPS
        o.ClientId = oidc["ClientId"];
        o.ClientSecret = oidc["ClientSecret"];  // Oidc__ClientSecret a runtime
        o.ResponseType = "code";
        o.UsePkce = true;
        o.SaveTokens = false;                   // i token restano sul server
        o.Scope.Clear(); o.Scope.Add("openid"); o.Scope.Add("profile");
        o.Events = new OpenIdConnectEvents { OnTokenValidated = HandOffToWisej };
    });
// app.UseAuthentication(); endpoint; poi app.UseWisej();
```

`Authority` arriva dalla configurazione ed è l'unico posto da cui vengono recuperate la discovery e le chiavi di firma; `RequireHttpsMetadata` resta attivo fuori dal lab. La registrazione presso il provider elenca un unico callback HTTPS esatto che termina con il `CallbackPath` dell'handler, quindi dietro un proxy che termina il TLS l'origine canonica deve sopravvivere (`app.UseForwardedHeaders()`), altrimenti l'handler costruisce un redirect URI `http://` che il provider rifiuta. Il client secret arriva come `Oidc__ClientSecret` dall'ambiente o dagli user secrets, che vengono caricati solo quando `GenerateAssemblyInfo` è true (il template Wisej lo imposta a false). Tieni disattivato `MapInboundClaims` così che `iss` e `sub` conservino i loro nomi di protocollo, e allinea `ResponseMode` al provider; il valore predefinito è `form_post` .

Da un token convalidato a un principal Wisej.NET

La richiesta di callback non ha una sessione Wisej.NET, e `Application.RefreshSessionId()` raggiunge il browser solo durante il caricamento di una pagina. Per questo `OnTokenValidated` non fa accedere nessuno: costruisce una `ExternalIdentity` dall'issuer e dal subject convalidati, la parcheggia in `ExternalHandoffStore` sotto un handle casuale, scrive quell'handle nel cookie HttpOnly `AccessOps.Handoff` , chiama `HandleResponse()` così che non venga creata alcuna sessione cookie dell'host, e reindirizza alla pagina iniziale.

Al caricamento di quella pagina, in `Program.Main` o in `Application.ApplicationRefresh` , `WisejExternalSignIn.TryCompleteOnPageLoad()` legge l'handle con `Application.Cookies[ ... ]` , fa scadere il cookie e riscatta l'handle una sola volta. `ExternalIdentityMapper` cerca issuer e subject in `external_logins` , mai per email, rifiuta un account disabilitato o un tenant che AccessOps non ha

approvato, e restituisce l'account locale con il suo tenant e i ruoli presi dai dati di AccessOps. Poi il `GrantStore` del Modulo 4 emette esattamente un grant di login, e `ApplicationPrincipal.Establish( ... )` imposta `Application.User` su un `ClaimsPrincipal` che porta account, tenant e grant, e ruota l'ID di sessione. Nulla di ciò che stava nel token è diventato un permesso.

Il modulo con password del Modulo 4 resta sotto *Service accounts only*: `LoginService` accetta account di servizio nominati come `svc.reports` e dà a ogni account interattivo l'errore generico, così che il ripiego non sia mai una seconda porta d'ingresso.

Rifiuti prima del grant, e due disconnessioni

Ogni fallimento deve fermarsi prima che venga eseguito `GrantStore`. Un callback che porta lo state di un altro tentativo (fixture F06) non si riesce a decifrare, e l'handler solleva `OnRemoteFailure`; AccessOps registra un motivo presente nella allowlist, mai la query del callback, e reindirizza alla vista di accesso. Un callback riprodotto (F07) trova il proprio cookie di correlazione già consumato e il proprio codice già riscattato, e un handle riprodotto non trova nulla nello store, quindi il tentativo ha comunque un solo grant in totale. Un'identità valida proveniente da un tenant non approvato (F08) supera i controlli di protocollo e viene rifiutata dal mapper senza alcun principal. Un rifiuto del provider arriva attraverso `OnAccessDenied` o `OnRemoteFailure` e porta alla vista di accesso con un percorso per riprovare.

La disconnessione chiude due sessioni, e ciascuna viene verificata separatamente. In locale, AccessOps revoca il grant, così che un'esportazione ritentata da una seconda scheda aperta venga rifiutata, poi chiude la sessione Wisej.NET. Il browser visita poi `/account/signout`, che restituisce `Results.SignOut` per gli schemi cookie e OpenID Connect; l'handler lo invia all'end-session endpoint del provider e lo riporta indietro attraverso `/signout-callback-oidc`. La prova che ha funzionato è che l'accesso successivo chieda di nuovo le credenziali. Nessuna delle due dimostra che i refresh token siano stati revocati; verifica la revoca, dove è disponibile, come terzo risultato.

> **Regola di sicurezza**
>
> Tieni separate le identità e decidi sul server. Il principal dell'host, il principal applicativo di Wisej.NET e il token Windows effettivo sono tre fatti diversi, e nessuno di essi è un permesso. Ogni lettura protetta controlla il grant corrente, il tenant e il record al confine del servizio prima che il repository venga eseguito, e un pulsante nascosto, un claim di ruolo o una scheda disconnessa non dimostrano nulla finché quel controllo non rifiuta.

> **Obiettivi di apprendimento**
>
> Alla fine di questo modulo sarai in grado di:

- ✓ Spiegare l'authorization code flow di OpenID Connect con PKCE, state e nonce, e a che cosa servono rispettivamente l'ID token e l'access token.
- ✓ Configurare gli handler cookie e OpenID Connect di ASP.NET Core in Startup.cs prima di app.UseWisej() con un callback registrato esatto, scope minimi, un'authority fissata e segreti caricati a runtime.

- ✓ Mappare issuer e subject convalidati su un account e un tenant locali di AccessOps, stabilire il principal della sessione Wisej.NET a partire da dati attendibili e mantenere un ripiego sul login locale per gli account di servizio.
- ✓ Rifiutare uno state contraffatto, un callback riprodotto e un tenant non approvato prima che venga creato qualsiasi grant, e disconnettersi dalla sessione locale e da quella del provider come esiti separati e verificati.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.

Lab pratico

Aggiungi ad AccessOps la federazione dei partner: registra l'app presso un provider OpenID Connect di test e un tenant, configura gli handler cookie e OpenID Connect in Startup.cs prima di app.UseWisej() per l'authorization code flow con PKCE, un callback HTTPS esatto, l'origine canonica preservata attraverso il proxy e il client secret caricato a runtime, e consegna l'identità convalidata alla sessione Wisej.NET, dove un mapper degli account cerca issuer e subject, assegna tenant e ruoli dai dati di AccessOps e crea esattamente un grant di login. Mantieni il login locale del Modulo 4 come ripiego per gli account di servizio, esegui un accesso riuscito, poi usa le fixture usa e getta per cambiare lo state, riprodurre un callback già consumato, accedere da un tenant non approvato e simulare un rifiuto del provider, mostrando che in nessuno dei quattro casi viene creato un grant dell'app, e concludi con una disconnessione che revoca il grant locale e chiude la sessione del provider.

Punto di partenza

Apri AccessOps come l'hai lasciato dopo il Modulo 4: la pagina di accesso della route dei partner con il suo modulo con password, il `GrantStore` che ogni esportazione controlla e `ApplicationPrincipal` che stabilisce il principal di sessione al caricamento di una pagina. Usa una registrazione del provider non di produzione e un tenant di test usa e getta (va bene un identity provider di test locale), e le fixture riservate F06, F07 e F08 per i casi negativi.

Approccio suggerito

- ✓ Registra AccessOps presso il provider di test con un unico callback HTTPS esatto che termina con `/signin-oidc`, solo gli scope `openid profile email`, e un client secret che imposti come `Oidc__ClientSecret` nell'ambiente o negli user secrets, mai in `appsettings.json`.
- ✓ Scrivi `AddAccessOpsOpenIdConnect()` in `Startup.cs`: schema cookie più OpenID Connect con `ResponseType` code, `UsePkce`, `SaveTokens` a false e un' `Authority` fissata; mappa `/account/signin` e `/account/signout` e chiama `app.UseAuthentication()` prima di `app.UseWisej()`.
- ✓ In `OnTokenValidated` parcheggia issuer e subject in `ExternalHandoffStore`, imposta il cookie HttpOnly `AccessOps.Handoff` e chiama `HandleResponse()`; al caricamento della pagina lascia che `TryCompleteOnPageLoad()` lo riscatti una volta, esegua `ExternalIdentityMapper`, crei un grant e chiami `ApplicationPrincipal.Establish`.
- ✓ Esegui il percorso di successo, poi F06 (state di un altro tentativo), F07 (lo stesso callback due volte), F08 (un tenant non approvato) e un rifiuto del provider, contando i grant dopo ciascuno e registrando solo una correlazione oscurata e il motivo presente nella allowlist.
- ✓ Disconnettiti, riprova un'esportazione da una seconda scheda aperta, poi accedi di nuovo e registra se il provider chiede le credenziali; disegna il diagramma di sequenza ripulito a partire da ciò che hai osservato.

Deliverable richiesti

- ✓ Configurazione degli handler cookie e OpenID Connect in Startup.cs con i segreti fuori dal sorgente
- ✓ Mapper degli account per issuer e subject che crea un grant per ogni accesso
- ✓ Ripiego sul login locale per gli account di servizio
- ✓ Cinque esiti della transazione: un successo e quattro rifiuti senza grant
- ✓ Diagramma di sequenza ripulito e disconnessione che copre entrambe le sessioni

Domande di autovalutazione

- ✓ Un attaccante copia l'URL di callback di un partner, codice e state compresi, e lo apre nel proprio browser qualche secondo dopo. Quali due controlli indipendenti lo rifiutano, e che cosa proteggerebbe ancora AccessOps se il codice non fosse ancora stato riscattato?
- ✓ L'email di un partner cambia presso il suo provider e un nuovo assunto lì riceve il vecchio indirizzo. Che cosa fa il tuo mapper al prossimo accesso di ciascuna delle due persone, e perché un mapper basato sull'email sbaglierebbe?
- ✓ Dopo la disconnessione, l'utente successivo su una macchina condivisa rientra subito senza password. Quale dei tuoi due esiti di disconnessione è fallito, e quale prova lo avrebbe scoperto prima del rilascio?

Regola di sicurezza

Tieni separate le identità e decidi sul server. Il principal dell'host, il principal applicativo di Wisej.NET e il token Windows effettivo sono tre fatti diversi, e nessuno di essi è un permesso. Ogni lettura protetta controlla il grant corrente, il tenant e il record al confine del servizio prima che il repository venga eseguito, e un pulsante nascosto, un claim di ruolo o una scheda disconnessa non dimostrano nulla finché quel controllo non rifiuta.