Course Outline

- Fundamentals of OAuth 2
- Fundamentals of OpenID Connect
- How web APIs use OAuth 2
- How to do identity federation with OIDC
- Overview of a Java implementation of OIDC called MITREid Connect
- Integrating OIDC into web apps
- Using OIDC for SSO
Protocol Overview
What is OAuth?
Delegated Authorization

- How can a resource owner authorize a piece of client software to access protected data on their behalf?
- How can we scale security decisions to work on the internet?
- How can we make a system that’s both secure enough for data protection and usable enough for average end users?
The players

Resource Owner & User Agent

Client

Protected Resource
The goal

Connect the Client to the PR
How do we do that?

(Traditionally)
Steal the keys (assume SSO)
Ask for the keys
Developer key
Is there a better way?
The OAuth approach

- Client gets the user to authorize the client
- User presents proof of that authorization grant to the client
- Client presents authorization grant to server to get a token
- Token represents authorized access for a specific client on behalf of a specific user in a specific context
The OAuth players

Resource Owner & User Agent

Authorization Server

Client

Protected Resource
Ever seen these?

Authorize a new application

The following application is asking for authorization to access your account:
MyMII 2.0

Authorize this application

Request for permission

famousity is requesting permission to do the following:

Access my basic information
Includes name, profile picture, gender, networks, user ID, list of friends and any other information I've shared with everyone.

Send me email
famousity may email me directly at erezmazor@hotmail.com • Change

Post to my Wall
famousity may post status messages, notes, photos and videos to my Wall

Access my data any time
famousity may access my data when I'm not using the application

Access my Profile information
Facebook status

Logged in as Erez Mazor (Not you?)

Allow Leave app
Then you’ve used OAuth!
OAuth

- Started in late 2006 to connect sites using OpenID for logins
- Version 1.0(a) standardized in IETF: RFC5849
- Version 2.0 modularized concepts, added explicit extensibility, and removed major pain points of 1.0
  - Standardized in IETF: RFC6749, RFC6750
  - Continued extension development today
How?
Endpoints

Authorization Endpoint

Token Endpoint

AS
Front Channel

Uses HTTP redirects through the browser
Back Channel

Uses direct HTTP connections
OAuth “flows”
Different methods for getting a token
The Authorization Code Flow

- UA (User Agent)
- AE (Authorization Endpoint)
- TE (Token Endpoint)
- AS (Authorization Server)
- PR (Resource Server)

Diagram showing the flow of interactions between these components.
Front–channel only flows
The Implicit Flow

UA → AE

AS

PR
Back-channel only flows

UA

AS

C

PR
The Client Credentials Flow
The Password Flow
OAuth 2.0 design philosophy

- Keep things as simple as possible (but no simpler)
  - Especially for the client
- Otherwise, developers will:
  - Continue to use password anti-patterns
  - Develop their own “OAuth-like” protocol
  - Forego security altogether
Security Vs. Usability

What security folks say to do

What users actually do
An application with 0% security and 100% functionality is still an application, but an application with 100% security and 0% functionality is useless.

– Eve Maler
Do I have an access token?

Call the API

Did it work?

Good!

Do I have a refresh token?

Refresh your access token

Did I get an access token?

Authorize a new access token

No Good!
Remember these?

Authorize a new application

The following application is asking for authorization to access your account:
MyMII 2.0

Authorize this application

Request for permission

famousity is requesting permission to do the following:

Access my basic information
Includes name, profile picture, gender, networks, user ID, list of friends and any other information I’ve shared with everyone.

Send me email
famousity may email me directly at erezmazor@hotmail.com. Change

Post to my Wall
famousity may post status messages, notes, photos and videos to my Wall

Access my data any time
famousity may access my data when I’m not using the application

Access my Profile information
Facebook status

Logged in as Erez Mazor (Not you?)

Allow Leave app
Involving end users

- End users help make security decisions
  - They have context of what they’re trying to accomplish
  - It’s the best way to scale to internet size

- More research needs to be done here
  - Especially in usability
TOFU
It’s good for you!
Trust on First Use

- First time through, ask:
  - “You’ve never allowed this before. This is what I can say about them, is that OK?”

- Subsequent times through:
  - “I’m reasonably sure this is the same thing that you’ve said OK to before, let it through”
It should look like this:

**Whitelist**
Trusted partners, business contracts, customer organizations, trust frameworks

**Graylist**
User-based trust decisions
Follow TOFU model, keep logs

**Blacklist**
Very bad sites we don’t want to deal with, ever
Whitelist
Trusted partners, business contracts, customer organizations, trust frameworks

Graylist
User-based trust decisions
Follow TOFU model, keep logs

Blacklist
Very bad sites we don’t want to deal with, ever

Organizations decide these

End-users decide these
Layers of user trust

- Security must be usable by regular people
- We need multiple models, together
  - It’s a continuum
- Let organizations decide:
  - What organizations/sites to trust automatically
  - Who to sue if something goes wrong
  - Who to block completely
- Let users decide:
  - If they trust things the organization is silent about
  - (It’s easy to forget about this one)
Now back to protocols...
Avoiding password proliferation
  ◦ User’s credentials never go to the client

API protection
  ◦ Hundreds of thousands of sites, projects, and systems … and growing

Mobile access to server systems

Authentication (sign-on) protocols
  ◦ Facebook Connect, Log In With Twitter, etc.
So, OAuth is a sign-on protocol?
So, OAuth is a sign-on protocol?

No, it isn’t.
So, OAuth is a sign-on protocol?

No, it REALLY isn’t.
A delicious metaphor

Chocolate VS Fudge

Delicious on its own

Versatile ingredient
  - Useful in many circumstances

Can be used to make fudge
What OAuth tells you

- Somebody authorized the Client
- Hints at what the client is authorized to do (scope) and for how long (expiration)
A confection with several ingredients
Can be made with chocolate
  • But needs more than just chocolate
  • Could be made without chocolate
What OAuth *doesn’t* tell you

- Who the user is
- If the user is still there
- What to call the user
- How the user authenticated in the first place
Create an identity API, protect it with OAuth
  ◦ Authorization Server becomes Identity Provider
  ◦ Client becomes Relying Party
Standardized user info coming from this API
  ◦ Name, email, picture, etc.
Session management
  ◦ Is the user still logged in?
  ◦ Request log out
Communicate authentication information
  ◦ Step up to high levels of authentication
Keep compatibility with basic OAuth2
OAuth2 + Identity API = OpenID Connect
New generation identity protocol

- OpenID Connect (OIDC) is built on experience with OpenID 2, OAuth, SAML, Facebook Connect, etc.
- Developed by the OpenID Foundation
  - http://openid.net/connect
What do you mean by “identity”?

- Authentication
- Authorization
- Profile information
- Verifiable claims
- Personal data
- Biometrics
Distributed
◦ Don’t know ahead of time who all the players are
◦ Internet collaboration

Federated
◦ Know who the players are, but not what exactly they’ll be doing
◦ Cross-organizational collaboration

Local
◦ Control everything within a single stack
◦ Within-app (username/pass)
◦ Within-enterprise (LDAP)
What is distributed identity?

Identity system in which:
- Users present some set of claims
- Claims can be verified by a remote party (Identity Provider, or IdP)
- Remote party might not be known before the transaction begins
Advantages of distributed ID

- Decentralized system
  - Scale to full internet size
- Multiple types and levels of trust for different transactions
- Interoperable, standardized protocols
- End-user credentials don’t leak to applications
- Better user experience
  - If done well
Disadvantages of distributed ID

- Feeling of loss of control
  - “Can I really trust someone else’s security?”
  - (But you’re always trusting someone else)

- Potentially horrible user experience
  - If not done well
Two layers of credentials

- Primary credential
  - Information the user can present directly to a system in order to authenticate

- Derived (federated) credential
  - Information asserted to a system by a third party on behalf of a user
Identity providers (IdP)
  ◦ Translation service between primary credential and derived credential

Users talk to the IdP
  ◦ One place to manage and control primary credentials

Applications talk to the IdP
  ◦ Using a standardized protocol
  ◦ Easier to talk to multiple IdPs
Back to OIDC
OAuth 2 authorization
  ◦ Authorization Server becomes Identity Provider
  ◦ Client becomes Relying Party

JSON Web Tokens
  ◦ Structured token format

Can work in fully-distributed mode
  ◦ Dynamic discovery and registration
  ◦ Self-issued identities

“Make the simple things simple, make the difficult things possible.”
The players

Resource Owner & User Agent

Authorization Server

Client

Protected Resource
The players (with OpenID Hats)

- End User & Web Browser
- Identity Provider
- Relying Party
- Identity API
What else it can do

- Higher levels of assurance
  - Signed and encrypted requests
  - Signed and encrypted responses
- Fine-grained claims management
- Distributed and aggregated claims
- Self-issued identities
- IdP-initiated login
  - Kicks off the standard flow “remotely”
- Can get very complex if you want it to
  - “SAML with curly braces”
A protocol proving ground

- OAuth 2 in the wild
- Real-life interoperability testing
- Real deployments, large and small
- Generalization of protocols
  - OIDC Discovery → Webfinger
  - OIDC Registration → OAuth 2 Dynamic Client Registration
  - JWT Claims
    - Subject, audience, authorized presenter
How can simplicity be secure?
OAuth and OpenID Connect Protocol Flow
OAuth limits information known by each party in the protocol

Security problems usually come from leaking information between parties

We will demonstrate this using the authorization code flow

- Widest separation of different roles
## Information in play

<table>
<thead>
<tr>
<th>Username</th>
<th>Authorization Code</th>
<th>Issuer</th>
<th>Redirect URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>Access Token</td>
<td>Authorization Endpoint</td>
<td>Client ID</td>
</tr>
<tr>
<td>Session 1</td>
<td>Refresh Token</td>
<td>Token Endpoint</td>
<td>Client Secret</td>
</tr>
<tr>
<td>Session 2</td>
<td>ID Token</td>
<td>User Info Endpoint</td>
<td>Client Registration</td>
</tr>
</tbody>
</table>
Initial State

UA

AS

C

PR
End-user initiates Client action

UA → C → AS → PR
Discover
Registration
Client redirects User to AS
User authenticates to AS
User authorizes Client

UA

AS

C

PR
AS issues Authorization Code

UA

AS

C

PR
AS redirects User to Client
Client sends code to AS
AS issues token(s)
AS issues token(s)
Client accesses PR

UA

AS

C

PR
OIDC clients are OAuth clients

- Do regular OAuth 2 with predefined scopes
  - Advanced control with extra parameters

- Get back access token
  - Use it to fetch user information
  - Maybe use it to call other services also

- Get back an ID token
  - Parse and validate it to figure out who’s there now
How to Parse and Validate an ID Token

If you don’t have a library to do that for you
The anatomy of an ID token

- Uses JSON Web Token (JWT) for structure
  - Set of claims about what it’s representing
  - Fully structured JSON object

- Uses JSON Object Signing & Encryption (JOSE)
  - Protect the set of claims during transmission

- Uses compact format for transmission
  - Can be put into HTTP headers, query parameters, form parameters, HTML fields, JSON strings...
We’ll concentrate on signed tokens today

More common and simpler
Example Signed JWT

```
{"typ":"JWT", "alg":"HS256"} +
{"iss":"joe", "exp":1300819380, "http://example.com/is_root":true} + (signature) =
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJtZDE1NiJ9eyJpc3MiOiJqb2Ui
eyJleHAiOjEzMD44MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ij0cnVl
mNvbS9pc19yb290Ijp0cnVl
```

88
Step 1: Split on “.”

- If you have 3 pieces, it’s a JWS
  - Header, payload, signature
- If you have 5 pieces, it’s a JWE
  - Header, encrypted key, initialization vector, ciphertext, authentication tag
Step 2: Parse the header

- Base64 URL Decode the first part from step 1
- Parse the result as a JSON object
- Claims represent information on how this object was put together
  - `alg`: Algorithm used to sign this object (this tells us how to validate the signature)

```
{ "alg": "RS256" }
```
Step 3: Parse the payload

- Base64 URL Decode the second part from step 1
- Parse the result as a JSON object

```json
{
    "iss": "https://idp.example.org/",
    "sub": "ZXCV-2341.21",
    "exp": 987341049600,
    "iat": 987341049000,
    "aud": "KZ-F33-FA53-M"
}
```
### ID Token Claims

<table>
<thead>
<tr>
<th>Claim Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>iss</td>
<td>URL of the server that issued this</td>
</tr>
<tr>
<td>sub</td>
<td>Unique (at iss) identifier of the user</td>
</tr>
<tr>
<td>exp</td>
<td>This token is no good after this timestamp</td>
</tr>
<tr>
<td>iat</td>
<td>This token was created at this timestamp</td>
</tr>
<tr>
<td>aud</td>
<td>List of clients this token is meant for</td>
</tr>
<tr>
<td>auth_time</td>
<td>The end-user authenticated at this timestamp</td>
</tr>
<tr>
<td>nonce</td>
<td>Copy of a client-supplied string value, used to protect against replay attacks</td>
</tr>
</tbody>
</table>
Step 4: Get the key

- If it’s a symmetric signature (HS*), use your own client secret
- If it’s an asymmetric signature (RS*, ES*), find the server’s public key
  - Look up the server’s discovery document, find the “jwks_uri” member, fetch that URL
  - Keys will be in a JWK Set (“jwks”) format

```json
{  "keys": [  
    {  
    }  
  ] }
```
Step 5: Validate the signature

- Base64 URL Decode the third part from step 1
  - This byte array is your signature

- Take the first and second parts from step 1 and put them together with a “.”
  - NOTE: use the exact strings as sent over the wire, do NOT re-encode them

- Validate the signature according to the algorithm stated in the header
Is “iss” an issuer we trust?
Is our client ID in the “aud” list?
Is “exp” a timestamp in the future?
Is “iat” a timestamp in the past?
Are “iat” and “auth_time” not too far in the past for our use case?
Is “nonce” the same value that we sent in our original request?
Step 7: User is logged in

- Combine “iss” and “sub” claims to create a globally unique identifier for this user
  - Methods vary by platform
  - MITREid Client uses:
    - `ImmutableMap.of("iss", issuer, "sub", subject)`
Scopes in OAuth
What is a “scope”?

- Loosely defined by OAuth specs
  - Space-separated list of strings, order independent
- Clients can request scopes to be approved by the resource owner
- Auth Server can communicate scopes back to the client along side access token
How are scopes used?

- Each scope represents a bundle of permissions
- Each action a client wants to take is associated with a set of permissions (and set of scopes)
- Each token is associated with a set of scopes
- Resource Server compares the set of scopes in the token with the set of scopes needed to fulfill the request to determine authorization
Scopes in OIDC

- Most scopes represent a bundle of claims
  - Name, homepage, picture: profile
  - Physical address: address
  - Email address: email
  - Telephone: phone

- Some turn on special functionality
  - Authentication subject: openid
  - Allow client to access profile when the user isn’t around anymore: offline_access
Advanced OAuth
Strongly Binding Bearer Tokens

- Combine client authentication to resource server with presentation of a token
  - Limits what a client can do on each request
    - (no more god-like powers just for having client credentials, still need user authorizations)
  - Limits the useful context of a token

- Bind a token to a TLS client identifier
  - In a mutual-TLS environment, OAuth provides authorization and delegation on top of existing security structures
Signed HTTP Messages

- Not yet an IETF standard (Maybe Someday?)
- Token is provided with public and private components
- Signature is calculated over the token and some parts of the HTTP message
- Token identifier and signature are sent to the resource server
- Resource server checks signature to provide holder-of-key assurance
Connecting the Authorization Server to the Protected Resource
OAuth deals with these parts
But what about this part?
Structured Tokens

- OAuth doesn’t define what goes into the token string itself
- Define a parseable format for moving data within the token: JSON Web Tokens (JWT)
- Clients and protected resources can verify the token through signatures (JOSE)
  - http://datatracker.ietf.org/wg/jose/
Example JWT

```
{"typ":"JWT", "alg":"HS256"} + 
{"iss":"joe", "exp":1300819380, "http://example.com/is_root":true} + (signature) =
```

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ.dBjftJeZ4CV
P-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```
Token introspection

- Unstructured or opaque tokens
  - “I have a token, what is it good for?”
- Token in, JSON out

```json
{
  "valid": true,
  "client_id": "s6BhdRkqt3",
  "scope": ["read", "write", "dolphin"],
  "subject": "2309fj32kl",
  "audience": "http://example.org/protected-resource/*"
}
```
Service Chaining
What if your PR isn’t so simple?

UA

AS

C

PR1

PR2

?
PR1 sends token to AS
AS mints new token for PR2
PR1 calls PR2 with new token

UA

AS

C

PR1

PR2
Federated Introspection

- All tokens are JWTs
- PR parses token
  - Validate signature
  - Check to see if issuer is trusted
- Send token to issuer’s introspection endpoint
  - Check scopes, subject, client identifier
Server and client built on Spring Security

Supports key features:
- Signed tokens
- Request objects
- Authorization code and implicit flows

General purpose OAuth2 server
- Flexible scope definitions
- Dynamic registration
- Token introspection
- Token chaining
Interoperability testing

- Interoperability testing with working group
  - Nomura Research Institute (PHP client)
  - OIDC–PHP (PHP Client)
  - IBM (Java client)
  - Nov Matake (Ruby client and server)
  - OIDC test suite (Python)
  - … and others

- Pending Interoperability event hosted by MIT Kerberos Consortium
Enterprise features

- Enterprise-friendly platform (Java Spring)
- Administration consoles
- Programmable API
- Modern UI
- Event and action logging
- Pluggable primary user authentication
- General-purpose OAuth 2.0 service
Welcome!
OpenID Connect is a next-generation protocol built on top of the OAuth2 authorization framework. OpenID Connect lets you log into a remote site using your identity without exposing your credentials, like a username and password.

About
This OpenID Connect service is built from the MITREid Connect Open Source project started by The MITRE Corporation.

Contact
For more information or support, contact the administrators of this system.

Current Statistics
There have been 1 user of this system who have logged in to 1 site, for a total of 1 site approval.
<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Mobile</th>
<th>Email</th>
<th>Address</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>Test Client</td>
<td><img src="image" alt="phone" /></td>
<td><img src="image" alt="openid" /></td>
<td><img src="image" alt="offline_access" /></td>
<td><img src="image" alt="profile" /></td>
</tr>
<tr>
<td>e4287521-0656-457e-9e4f-059c873313fb</td>
<td>adkikcfghkld.</td>
<td><img src="image" alt="openid" /></td>
<td><img src="image" alt="phone" /></td>
<td><img src="image" alt="email" /></td>
<td><img src="image" alt="profile" /></td>
</tr>
<tr>
<td>45172aeb-cf00-4da6-b016-adc2283bbee80</td>
<td>Simple Web App</td>
<td><img src="image" alt="openid" /></td>
<td><img src="image" alt="phone" /></td>
<td><img src="image" alt="email" /></td>
<td><img src="image" alt="profile" /></td>
</tr>
</tbody>
</table>
Edit Client

Main

Client name: edkckdfghjkld
Human-readable application name

Client ID: e4257621-0056-467e-9e4f-059c
Unique identifier. If you leave this blank it will be automatically generated.

Redirect URI(s): http://

Description: Type a description
Human-readable text description

Logo: http://
URL that points to a logo image, will be displayed on approval page

Terms of Service: http://
URL for the Terms of Service of this client, will be displayed to the user

Policy: http://
URL for the Policy Statement of this client, will be displayed to the user

Home Page: http://
URL for the client's home page, will be displayed to the user

Application Type: Native

Contacts: new contact

Save Cancel
Approve New Site

**Caution:**
This client was dynamically registered and has very few other users on this system.

Do you authorize "Simple Web App" to sign you into their site using your identity?

**Redirect URL:** http://nivens.richer.org:8080/simple-web-app
/openid_connect_login

**Access to:**
- Log in using your identity
- Basic profile information
- Email address
- Physical address
- Telephone number

**Remember this decision:**
- Remember this decision until I revoke it
- Remember this decision for one hour
- Prompt me again next time

[Authorize] [Deny]
<dependency>
  <groupId>org.mitre</groupId>
  <artifactId>openid-connect-parent</artifactId>
  <version>1.1.0</version>
  <type>pom</type>
</dependency>

*Also: openid-connect-common, openid-connect-client, openid-connect-server, and openid-connect-server-webapp*
Main Project: Java OIDC

- OpenID-Connect-Java-Spring-Server
  - (I shouldn’t be allowed to name projects)
- Maven project with several sub-modules
  - openid-connect-server: server library
  - openid-connect-client: client library and filter
  - openid-connect-common: components common to both client and server
  - openid-connect-server-webapp: deployable copy of the server (configuration, database schema, web resources)
Project components

- Per-server overlays (not public)
- Hosted on GitHub
- Open Source, owned by SpringSource

MITREid Connect
Open Source Project

Server A  Server B  ...

SECOAUTH

Spring Security  Spring

Java
Major Java libraries in use

- Spring Security OAuth (SECOAUTH)
- Nimbus-JOSE-JWT
- Guava
- GSON
- Spring 3.1
- Spring Security 3.1
Other Components
Example Server Overlay

- example-server-overlay
- Shows how to create a custom overlay of the server webapp
  - All dependencies and structure required for an overlay are demonstrated here
- Contents of my-openid-connect-server are injected into resulting WAR file
Simple Web App

- simple-web-app
- Minimal example application with client filter set up in context
  - Uses OIDCAuthenticationFilter for app login
- Example beans for each type of client component
  - Pre-configured to connect to the demo server with dynamic registration and discovery
Key Generator

- json-web-key-generator
- Java app that generates public/private keypairs and prints them in JWK Set (.jwks) format
- Output can be used by server and client libraries via the KeyStore class
Account Chooser

- account-chooser
- Simple JavaScript application that demonstrates how to centralize user choice between multiple IdPs
- Can be configured with multiple buttons
Demo Server

- Build openid-connect-server-webapp to get a self-contained demonstration server
  - openid-connect-server-webapp.war
  - Install into a servlet container (like Tomcat)
  - Run in embedded jetty server

- Default configuration:
  - Issuer: http://localhost:8080/openid-connect-server-webapp/
  - One RSA 1024 key (prepackaged)
  - In-memory HSQL database
    - Configured with two users and one client
    - Cleared and re-initialized on every restart
Demo Server: Users

- **Admin user:**
  - Username: admin
  - Password: password
  - Can use administration console (edit clients, scopes, whitelists, blacklists)

- **Regular user:**
  - Username: user
  - Password: password
  - Can authorized clients
  - Can use dynamic registration self-service console
Demo Server: Client

- Client ID: client
- Client secret: secret
- Redirect URIs: http://localhost/, http://localhost:8080/
- Name: “Test Client”
- Scopes: openid, profile, phone, email, address, offline_access
Building and Editing the MITREid Connect Project
Download from Git

- All source control handled in Git
  - Primary branch: master
  - Legacy branch: 1.0.x

- If you are editing: always work in a new branch
  - Always
Build with Maven

- Always build from the root of the project
  - Building from sub-projects is not supported
- Maven will pull all dependencies automatically
- Use command: mvn package
  - Or: mvn clean package
Created artifacts

- openid-connect-server/target/openid-connect-server.jar
- openid-connect-common/target/openid-connect-common.jar
- openid-connect-client/target/openid-connect-client.jar
- openid-connect-server-webapp/target/openid-connect-werver-webapp.war
Creating a custom OIDC server using MITREid
Customizing MITREid Connect

- The MITREid Connect server is designed to be customized and connected to an existing user store
- Overlay local configurations and custom classes on the existing WAR file
Maven Overlays

- Take a previously-built WAR file
- Explode the contents of the WAR
- Combine these with local files (classes, configuration, other elements)
  - Local files overwrite those in the source WAR when the name & path is the same
- Package the results as a new WAR file
The server ships with a single “test” key that needs to be replaced for real instances
  ◦ All tokens are signed with this key

The json-web-key-generator program will create keys in the appropriate format
  ◦ Example: -t RSA -i rsa1 -s 1024 -S

Overlay:
  ◦ src/main/resources/keystore.jwks
  ◦ src/main/webapp/WEB-INF/crypto-context.xml
Customizing the Issuer

- The “issuer” value MUST be set to the URL that clients will access the server from
  - Defaults to http://localhost:8080/openid-connect-server-webapp/
  - Java/Tomcat aren’t smart enough to reliably autodetect this, so we put it in configuration

- Overlay:
  - src/main/webapp/WEB-INF/server-config.xml
  - (You can also change the display name and logo of the server from here)
Customizing the database

- Defaults to an in-memory HSQL store
- File-based HSQL:
  - Change JDBC URL to file-based example, point at a writeable directory
- MySQL:
  - Change JDBC URL to MySQL
  - Change EclipseLink connector properties to MySQL
- Overlay:
  - src/main/webapp/WEB-INF/data-context.xml
Customizing user authentication

- Defaults to a username/password against local users stored in a set of database tables
- Can be replaced with anything that Spring Security can use
  - Normal users require “ROLE_USER”
  - Administrators also require “ROLE_ADMIN”

- Overlay:
  - src/main/webapp/WEB-INF/user-context.xml
Connecting to a user store

- **User Info Service**
  - Provides user profile information for requests to the UserInfoEndpoint
  - Needs to be able to key off of the Spring Security Principle used in the primary authentication

- Declare service as `@Primary` to be autowired

- **Overlay:**
  - `src/main/webapp/WEB-INF/user-context.xml`
MITREid Connect Server Management Application
No HTML form processing

- Java server sends JavaScript and HTML templates to the browser
- Browser calls RESTful API
  - JSON objects in and out
- JavaScript processes objects
- Browser renders appropriate template
  - No page reloads (if they can be avoided)
Application Structure

JavaScript App ➔ Controller ➔ Template ➔ HTML

Java Server ➔ RESTful API ➔ Data Store
Our APIs

- Client management: /api/clients
- Approved sites: /api/grant
- System scopes: /api/scopes
- Whitelisted sites: /api/whitelist
- Blacklisted URIs: /api/blacklist
Why build this way?

- We needed to build an API anyway
  - Allow external (OAuth-protected) control
  - Wanted it to be RESTful

- Traditional forms would require additional processing code on the server
  - Why write the same code twice?

- Better user experience
  - More responsive web design
Front end libraries

- **Bootstrap**
  - UI components, layout, and visual look and feel

- **Backbone**
  - MVC structure, object management, routing

- **Underscore**
  - Templates, JavaScript utilities

- **JQuery**
  - JavaScript utilities
JavaScript source layout

- `src/main/webapp/resources/`
  - JavaScript application code: `/js/`
  - HTML templates: `/template/`

- **Classes and templates are split up:**
  - `admin`: main application, router, general and miscellaneous templates and classes
  - `client`: client registration
  - `dynreg`: dynamic registration self-service
  - `grant`: approved apps
  - `scope`: system scopes
  - `whitelist`: whitelisted apps
Using MITREid Connect’s Client Code
Structure of an OIDC client

- Find out which Issuer to talk to
- Get the server configuration for that Issuer
- Get a client configuration for us at that Issuer
- Create an Authorization Request
- Receive an Authorization Response
  - Basic clients: callback contains a code, send to token endpoint to get tokens
  - Implicit clients: callback contains tokens
- Parse and validate ID Token
- Send Access Token to User Info Endpoint to get user information
Issuer Services

- **Static**
  - Always returns the same issuer

- **Third-party**
  - Parses URL parameters to look for an issuer (as defined in OIDC)
  - Forwards user to an “account chooser” URL if no issuer is found

- **Webfinger**
  - Parses URL parameters to look for an identifier, uses Webfinger to discover the issuer

- **Hybrid**
  - Looks for third party login parameters, then webfinger identifier, forwards to login page if not found
Server Configuration Services

- **Static**
  - Contains a map from issuer URLs to server configuration objects

- **Dynamic**
  - Does OIDC Discovery on given issuer URL
  - Fetches and caches server’s published configuration

- **Hybrid**
  - Looks up issuer in statically configured map, does discovery if config isn’t found in there
Client Configuration Services

- **Static**
  - Contains a map from issuer URLs to registered client configuration objects

- **Dynamic Registration**
  - Does OIDC Dynamic Client Registration for each unknown issuer
  - Stores registered client information via a Registered Client Service

- **Hybrid**
  - Checks statically configured map, then does dynamic registration if config isn’t found
Options Services

Default

- Adds optional request parameters such as “display” and “prompt” to every request
Authorization URL Services

- **Plain**
  - Constructs a URL with all request elements set as query parameters on the URL

- **Signed**
  - Constructs a request object with all request elements set as members of the request object
  - Signs the request object with the client’s configured key
  - Requires configuration to set up a signing service and key publisher

- **Encrypted**
  - Constructs a request object with all request elements set as members of the request object
  - Encrypts the request object with the server’s configured key
Add Spring Security

- Add the Spring Security filters and dependencies
- Add the MITREid Connect components:
  - OIDCAuthenticationFilter
  - OIDCAuthenticationProvider
Listen on /openid_connect_login
Begins login process
  ◦ Handles discovery and registration
  ◦ Sends redirect to authorization endpoint
Acts as redirect_uri for callbacks
  ◦ Takes in authorization code
  ◦ Calls token endpoint
Returns an Authentication object
  ◦ Validates ID Token
  ◦ Saves ID Token, Access Token, and Refresh Token
Authentication Provider

- Takes in OIDC Authentication object from filter
- Loads UserInfo from UserInfoEndpoint
  - Uses Access Token to call server
- Gives each user a GrantedAuthority based on issuer and subject and ROLE_USER
  - Can be translated to ROLE_ADMIN or other roles using a custom NamedAuthoritiesMapper
Authentication Token

- Spring Security authentication element
  - Not to be confused with OAuth tokens!
- Contains issuer/sub pair as Principal
- Contains string copies of token values
  - ID Token
  - Access Token
  - Refresh Token
- Contains reference to UserInfo (if found)
Using OIDCAuthenticationToken

- Inject into a Spring @Controller as OIDCAuthenticationToken, Principal, or Authentication
- Call .get UserInfo() to access user’s profile information for display or processing
- Use only combination of issuer/subject as unique user identifier
  - Do NOT use anything from User Info!
DynamicClientRegistrationService uses an in-memory store by default
- Meaning: every time the client servlet is redeployed it will re-register itself at the server (this is bad)

Configure with a RegisteredClientService to store client registrations between boot
- Ideally save it to your application’s data store somewhere
- Simple file-on-disk version included with library
RHEX

MITRE id

research prototype

Clipper

openPDS

Case Studies

MPN

MITRE Partnership Network
Case Study: RHEEx
RESTful Health Exchange
What is RHEX?

- A RESTful system for exchange of health data
  - Best practices around HTTP protocols
  - Links to protected data

- Distributed user base
  - Medical practitioners in different organizations
  - Links to files on remote systems
Primary Care Physician Patient Records

Link to record with hData

Fetch Record with OAuth

Authenticate With OpenID

Consulting Physician’s System

Consulting Physician Identity Provider
Beyond RHEX: Blue Button

- Blue Button+ initiative on health record exchange
- Using OAuth 2 for protecting records in RESTful API
- Using dynamic discovery and registration
OpenID for MITRE users

- OpenID 2.0 prototype running since 2009
  - OpenID Connect available since 2012
- Backed by MITRE’s identity infrastructure
  - If you are a MITRE person, you have an OpenID
- Usable on any site that supports OpenID
- Nearly–single–sign–on experience with minimal integration efforts
  - Many web apps supported OpenID 2.0 natively
MITREid by the numbers

- Over **6800** people have used it at least once
  - Ease of developer integration
  - Ease of user experience

- Users have accessed over **400** distinct sites
  - Handshake is largest, whitelisted by admins
  - Most sites are not whitelisted
  - Serves the “long tail” of user needs
Number of users per site

*logarithmic scale
Corporate Firewall

DMZ

Intranet

External OP

Database

Corporate SSO

Internal OP

User Data
External UA, External RP

- Corporate Firewall
- DMZ
- Intranet
- External OP
- Database
- Two-Factor Signon
- Internal OP
- User Data
Case Study: Clipper Web Services
Adapting a legacy application
Trans-lingual web browsing and searching project from MITRE

Originally built as a monolithic application, being redeveloped as a set of services

- Modular architecture
- HTTP interconnections
Advanced OAuth

- Token introspection
  - Resource servers call back to authorization server to validate tokens at runtime

- Token chaining
  - Resource servers need to call each other
  - Downslope tokens for minimal access
User login

- User Interface
- Dictionary Service
- Search Service
- OpenID Connect
- AuthN Provider
Access a service

User Interface → Dictionary Service → OpenID Connect → AuthN Provider

Search Service
Chaining a service request

User Interface

Dictionary Service

Search Service

OpenID Connect

AuthN Provider

188
Turtles all the way down...
What is the MPN?

- Services and applications to help MITRE interact with external partners
- Loosely-coupled architecture
  - Based on open standards
  - User-driven experience
  - Opt-in for applications
CAC Proxy

Identity Clearinghouse

Authorization
- Rule Based Decision Engine
- Handshake
- CommShare Partners
- Center and lab capabilities
- Identify additional capabilities

External Partners

Notification Service
- PuSH-EE
- ActivityStreams Aggregator

Content APIs
- MPN Portal
  - Apache Rave
  - Apache Shindig
- OpenSocial Data Source

Apps “opt-in” to services

Cloud Applications

Permission Management Authorization
- UMA

User Info Endpoint

External Partners

CommShare Partners

Center and lab capabilities

Apps “opt-in” to services

Open ID Connect

Apps “opt-in” to services

OpenSocial Data Source

External Partners

External Partners

External Partners

External Partners
Making life easy

- Leverage what’s working on the internet
  - Build on existing code and libraries
- Give users a consistent experience
  - One account across applications
  - No password at MITRE if we can avoid it
- Let developers focus on their applications
- Make it easy to do the right thing
MITREid Connect Interop Testing Server
https://mitreid.org/
MIT OpenID Connect Server

https://oidc.mit.edu/
Try it out!

- Open to all MIT account holders
- Log in with username/password or Kerberos tickets
  - (Assuming your browser is set up for Kerberos)
- Can be used on any valid OIDC site
- Not quite production ready
  - Some things might break, some things might change
Conclusions
OpenID Connect Flow Diagrams
OpenID Connect
Authorization Code Flow

Client

Request Authorization

A

Authenticate End-User

Obtain Authorization

B

Direct End-User to Client

Request Access Token and ID Token

C

Validate Client

Validate Authorization Code

Grant Access Token

D

Access Protected Resources

Identity Provider

Service Provider

Grants Authorization

Request includes
response_type: code
scope: openid (MAY include additional scopes)
client_id
direct_url
state (required, recommended)
nonce
display (optional)
prompt (optional)
request (optional)
request_url (optional)

Service Provider

Grants Access Token and ID Token

JSON response object (200 OK) includes
access_token
id_token
token_type: Bearer
expires_in (optional)
refresh_token (optional)
scope (optional; SHOULD be included if the scope granted differs from the scope requested)

Client Requests Authorization

A

Request includes
response_type: code
scope: openid (MAY include additional scopes)
client_id
direct_url
state (required, recommended)
nonce
display (optional)
prompt (optional)
request (optional)
request_url (optional)

©2012-The MITRE Corporation. All rights reserved.
OAuth 2 Authorization
Authorization Code Flow

Client

Direct Resource Owner to Service Provider

Service Provider

Authenticate Resource Owner

Obtain Resource Owner Authorization

Direct Resource Owner to Client

Obtain Authorization Code

Request Access Token

Validate Client

Exchange Access Code for Access Token

Validate Authorization Code

Grant Access Token

Access Protected Resources

Access Protected Resources

A Client sends Authorization Request

Request includes
- response_type: token
- client_id
- redirect_uri (optional; may be pre-configured with service provider)
- scope (optional)
- state (recommended)

B Service Provider grants Authorization

Redirection URI (302 Found) includes
- code
- state (required if state was sent in the request; must be equal to what was received)

C Client requests Access Token

Request includes
- grant_type: authorization_code
- code
- redirect_uri (optional; may be pre-configured with service provider. Must match value sent in step A if provided)

D Service Provider grants Access Token

JSON response object (200 OK) includes
- access_token
- token_type
- expires_in (optional)
- refresh_token (optional)
- scope (optional; SHOULD be included if the scope granted differs from the scope requested)

If the client is confidential, it must authenticate with the Authorization Server in this request.
OpenID Connect
Implicit Flow

Client

<table>
<thead>
<tr>
<th>Request Authorization</th>
</tr>
</thead>
</table>
A                      

Identity Provider

<table>
<thead>
<tr>
<th>Authenticate End-User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain Authorization</td>
</tr>
</tbody>
</table>
B                      

Request includes
- response_type: id_token
- scope: openid (MAY include additional scopes)
- client_id
- redirect_uri (required IFF the client has pre-configured more than one value with the service provider)
- state (optional, recommended)
- nonce
- display (optional)
- prompt (optional)
- request (optional)
- request_uri (optional)

Service Provider

<table>
<thead>
<tr>
<th>Grants Authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
B

Redirection URI (302 Found) includes
(url-encoded in fragment)
- access_token
- id_token
- token_type: Bearer
- expires_in (optional)
- refresh_token (optional)
- scope (optional; SHOULD be included if the scope granted differs from the scope requested)
- state (required IFF state was sent in the request; must be equal to what was received)

©2012 - The MITRE Corporation. All rights reserved.