Implementing Authorization for Applications & APIs
Authorization is hard!

• **Many approaches**
  – roles, permissions, resource-based, ACLs... (and permutations)
  – queries vs commands

• **No standard solution**
  – often very application specific
  – blurry line between authorization and business rules
  – XACML good example of failed attempt to standardize
Modern Applications

- Browser
- Native App
- Server App "Thing"
- Web App
- Web API
- Identity Provider
Identity != Permissions
Overloaded Security Token

```json
{
  "iss": "https://idsrv4",
  "exp": 1340819380,
  "aud": [ "api1", "api2" ],
  "amr": [ "password" ],
  "auth_time": 12340819300

  "sub": "182jmm199",
  "name": "Doug Ross",

  "role": [ "Approver", "Doctor" ],

  "permission": [ "DeleteData", "ManageCustomers", "ChangeTreatmentPlan" ]
}
```
Token Usage

- Patient Data API
- Oncology API
- Cardiac API
Permissions and Tokens

• **Separation of concerns**
  – authentication vs authorization
  – identity system does not have intimate knowledge of application specific authorization rules
    • "do authorization as close as possible to the resource you are trying to protect"

• **Tokens can be re-used at several places**
  – claims might have different meaning for each consumer
  – token & claim bloat

• **Permissions might change**
  – only way to update the data would be to get a new token
Identity + Permissions

== Authorization
Modern Applications

- Authorization Provider
- Identity Provider

- Web App
- Web API
- Browser
- Native App
- Server App "Thing"
- "Thing"

@leastprivilege / @brocklallen
Identity Provider

Authorization Provider

Admin UI

Get API specific permissions

Get client specific permissions

Authentication +
token request

Call APIs

Patient API

Oncology API

Cardiac API

@leastprivilege / @brocklallen
Our Vision

Authorization Provider

- Global roles (static / dynamic)
- App specific roles (static / dynamic)
- Roles to permission mappings
- resources & rules (future)

Management API

Client API

Client Library

Admin UI

{ "sub": "jd9j91199j1", "role": "Doctor", "contractor": "true" }
Client Integration Strategies

- **Provide a client library and call the authorization provider manually**
  - client library takes care of caching & refreshing

- **Augment the ClaimsPrincipal**
  - e.g. using claims transformation in pipeline

- **Use a specialized authorization API**
  - e.g. ASP.NET Core Authorization
public class TreatmentController : Controller
{
    private readonly AuthorizationProviderClient _client;

    public TreatmentController(AuthorizationProviderClient client)
    {
        _client = client;
    }

    public async Task<IActionResult> Update(TreatmentUpdateModel model)
    {
        var (roles, permissions) = await _client.GetAuthorizationAsync(User);
        // or
        var allowed = await _client.IsInRoleAsync(User, "Doctor");
        // or
        allowed = await _client.HasPermissionAsync(User, "PrescribeMedication");
    }
}
Augment the ClaimsPrincipal

- Inject roles and permissions into current principal
  - backwards compat with existing libraries

```csharp
[Authorize(Roles = "Doctor")]
public async Task<IActionResult> Update(TreatmentUpdateModel model)
{
    ...
}
```
ASP.NET Core Authorization

• **Very flexible authorization library**
  – created for ASP.NET Core, but has been back-ported by the community*

• **Introduces a policy-based framework**
  – decoupling authorization logic from business code
  – extensible
  – supports resource-based authorization

Authorization Policies

```csharp
services.AddAuthorization(options =>
{
    options.AddPolicy("PrescribeMedication", policy =>
    {
        policy.RequireAuthenticatedUser();
        policy.RequireClaim("permission", "PrescribeMedication");
    });
});
```

```csharp
[Authorize("PrescribeMedication")]
public IActionResult Update()
{
    // stuff
}
```
Programmatically using policies

```csharp
public class TreatmentController : Controller
{
    private readonly IAuthorizationService _authz;

    public TreatmentController(IAuthorizationService authz)
    {
        _authz = authz;
    }

    public async Task<IActionResult> Update()
    {
        var allowed = await _authz.AuthorizeAsync(User, "PrescribeMedication");
        if (!allowed) return Challenge();

        return View();
    }
}
```
...or from a View

```csharp
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService _authz

@if (await _authz.AuthorizeAsync(User, "PrescribeMedication"))
{
    <div>
        <a href="/treatment/update">Update</a>
    </div>
}
```
Policy Provider

• **Extensibility point that allows creating policies on the fly**
  – no need to create explicit "permission policies" anymore

```csharp
[Authorize("PrescribeMedication")]
public IActionResult Update()
{
    // stuff
}
```

// this policy is not statically defined
// but gets created dynamically
// check for permission claim

@leastprivilege / @brocklallen
public class AuthorizationPolicyProvider : DefaultAuthorizationPolicyProvider
{
    public async override Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        // check static policies first
        var policy = await base.GetPolicyAsync(policyName);
        if (policy != null) return policy;

        if (await _client.HasPermissionAsync(_contextAccessor.HttpContext.User, policyName))
        {
            return Allowed;
        }

        return Denied;
    }
}
Custom Requirements

```csharp
public class MedicationRequirement : IAuthorizationRequirement {
    public string MedicationName { get; set; }
    public int Amount { get; set; }
}
```
public class MedicationRequirementHandler : AuthorizationHandler<MedicationRequirement>
{
    private readonly AuthorizationProviderClient _client;

    public MedicationRequirementHandler(AuthorizationProviderClient client)
    {
        _client = client;
    }

    protected override async Task HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    MedicationRequirement requirement)
    {
        var user = context.User; var allowed = false;

        if (await _client.HasPermissionAsync(user, "PrescribeMedication"))
        {
            if (requirement.Amount < 10) allowed = true;
            else allowed = await _client.IsInRoleAsync(user, "Doctor");

            if (allowed || requirement.MedicationName == "placebo")
            {
                context.Succeed(requirement);
            }
        }
    }
}
Using a custom Requirement

```csharp
public async Task<IActionResult> Prescribe(int amount)
{
    var meds = new MedicationRequirement
    {
        MedicationName = "aspirin",
        Amount = amount
    };

    var allowed = await _authz.AuthorizeAsync(User, meds);
    if (!allowed) return Challenge();

    return View("Confirm");
}
```
Resource-based Authorization

Subject
- client ID
- subject ID
- scopes
- more claims

+ DI

Operation
- read
- write
- send via email
- ...

Object
- ID
- owner
- more properties

+ DI
Example: Patient Resource

```csharp
public class Patient
{
    public ICollection<string> Allergies { get; set; }
}
```
public class PatientHandler : AuthorizationHandler<PrescribeMedicationOperation, Patient>
{
    private readonly AuthorizationProviderClient _client;

    public PatientHandler(AuthorizationProviderClient client)
    {
        _client = client;
    }

    protected override async Task HandleRequirementAsync(AuthorizationHandlerContext context, PrescribeMedicationOperation medication, Patient patient)
    {
        // medication requirement logic
        if (!patient.Allergies.Contains(medication.MedicationName))
        {
            allowed = false;
        }
        if (allowed) context.Succeed(medication);
    }
}
public async Task<IActionResult> PrescribeMed(int amount, string medication)
{
    var operation = new PrescribeMedicationOperation
    {
        Amount = amount,
        MedicationName = medication
    };

    var patient = new Patient { Allergies = { "aspirin" } };

    var allowed = await _authz.AuthorizeAsync(User, patient, operation);
    if (!allowed) return Challenge();

    return View("Confirmation");
}
Conclusion

• Different levels of authorization lend to different abstractions