SECURITY & AUTH PATTERNS · MODULE 3

Windows Impersonation and Delegation

Lesson Guide · Lab / Exam Guide

LearnWisej

Lesson Guide

Request identity versus effective token, scoped impersonation on .NET Framework and with WindowsIdentity.RunImpersonated on ASP.NET Core, the Wisej.NET impersonate setting and its ImpersonationBegin and ImpersonationEnd hooks, token reversion, and the Kerberos second hop. 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

Request identity versus effective token, scoped impersonation on .NET Framework and with WindowsIdentity.RunImpersonated on ASP.NET Core, the Wisej.NET impersonate setting and its ImpersonationBegin and ImpersonationEnd hooks, token reversion, and the Kerberos second hop.

1. The production problem

AccessOps keeps confidential documents on Windows file storage whose ACLs were written for people, not for the web server. The intranet route already knows the staff member, so it is tempting to impersonate everywhere and let Windows decide. That turns one file read into a process-wide identity change and fails confusingly once the file sits on a second server.

2. Key concepts

Concept	Why it matters
request identity	The `WindowsIdentity` that IIS Windows Authentication attached to the HTTP request; it says who called, but by itself it opens nothing.
effective token	The token the thread presents when Windows opens an object, read with `WindowsIdentity.GetCurrent()`; normally the pool identity; the user only inside an impersonation scope.
scoped impersonation	Running one bounded operation under the user's token and reverting at the end, even when it throws.
`impersonate` setting	The Default.json switch behind `Configuration.Impersonate` (default `false`) that makes Wisej.NET raise its impersonation events around request threads and `Application.StartTask` tasks.
`ImpersonationBegin` / `ImpersonationEnd`	Process-wide `HandledEventHandler` events; on .NET Framework the default handling impersonates and later reverts, and setting `Handled` suppresses that default.
ACL proof	Evidence that access came from the token's groups against a real ACL: one file only the pool can read, one only a named lab user can read.
second hop	The web server reusing the user's identity against another machine, which needs Kerberos delegation to an allowed target and a backend ACL that grants the user.

3. Lesson narrative

Who called is not who opens the file

A Windows-authenticated request carries two facts that are easy to merge. The request identity is what the host authenticated: on ASP.NET Core it reaches the Wisej.NET session as `Application.User`, whose `Identity` is a `WindowsIdentity` under IIS Windows Authentication. The effective token is what the executing thread presents to the operating system, and on a normal request that is still the application pool identity. Windows never asks for a name when it opens a file; it compares the token's user SID and group SIDs with the file's ACL. The AccessOps application subject from Module 1 is a third, separate fact.

That is why the lab proves access with ACLs, not names: one folder only the application pool can read, one only a named domain lab user can read. Before impersonation the user-only read must fail and the pool-only read succeed; inside the scope the outcomes flip. If both identities read both files, check inherited ACLs and local administrator membership. Record `WindowsIdentity.GetCurrent()` with its `Name`, `User` SID and `ImpersonationLevel`, and dispose the identity it returns.

.NET Framework, the impersonate setting and the thread hooks

On .NET Framework four switches sound alike and do different things: the IIS Windows Authentication role service, System.Web `<authentication mode="Windows" />`, System.Web `<identity impersonate>`, and the Wisej.NET `impersonate` key in Default.json. The lab keeps System.Web impersonation off and uses

`using (identity.Impersonate()) { … }` around one synchronous file read, with the identity taken from `Application.UserIdentity` (the request's logon identity, which exists only in the .NET Framework build).

With `"impersonate": true`, which is read once and cached for the whole process, Wisej.NET raises `ImpersonationBegin` at the start of each request thread and each task started with `Application.StartTask`, and `ImpersonationEnd` when it finishes. On .NET Framework, unless a handler sets `e.Handled = true`, Begin calls `UserIdentity.Impersonate()` and End disposes that context. The events are raised around the request thread: `ImpersonationBegin` fires before `ThreadBegin`, and `ImpersonationEnd` before `ThreadEnd`. (Older documentation says the opposite.) They are static, process-wide events: attach them once at application start, and if you only observe them leave `Handled` false so the framework cleans up what it started. A raw `Task.Run` gets none of this.

The lab compares the setting off and on for a click and an `Application.StartTask` body, with WebSocket enabled and in HTTP-only mode, each in a fresh session.

A bounded RunImpersonated read on ASP.NET Core

On .NET 8 and .NET 10 the two events still fire when `impersonate` is true, but Wisej.NET does nothing by default: modern .NET has no `WindowsIdentity.Impersonate()` and no impersonation context to keep open between a Begin and an End. The supported pattern is a scope that owns its own reversion. AccessOps wraps it in a `ScopedDocumentReader` that receives a document ID, never a path:

```
[SupportedOSPlatform("windows")]
public byte[] Read(HttpContext context, int documentId)
{
    if (context.User.Identity is not WindowsIdentity identity || !identity.IsAuthenticated)
        throw new UnauthorizedAccessException();
    // policy first: grant, tenant and record, then a canonical path under the root
    string path = _catalog.ResolveApprovedPath(documentId);

    return WindowsIdentity.RunImpersonated(identity.AccessToken, () ⇒
    {
        using (var current = WindowsIdentity.GetCurrent()) _trace.Record("inside", current);
        return File.ReadAllBytes(path);
    });
}
```

The browser never chooses a path; the catalog maps the ID to a file under one approved root only after the application decision allows the read. `RunImpersonated` restores the previous token when the delegate returns or throws, which is what fixture F03 proves: request an approved document whose file is missing, let `FileNotFoundException` escape, and the "after" row must show the pool identity again. The denied ACL case uses a file that exists, so "not found" never masquerades as "not allowed". Keep the scope synchronous; use `RunImpersonatedAsync` only when the awaited work completes inside the request, and never store the token in session state or hand it to long-lived work.

Run the reader where the request token is live: an authorized endpoint mapped before `app.UseWisej()`, or code reaching the current `HttpContext` through `IHttpContextAccessor`. A Wisej event over WebSocket sees the principal captured when the socket opened, and the host owns that token's lifetime, so the lab records that case rather than relying on it.

Classifying the second hop

A read that works on a local NTFS folder and fails against `\\files01\confidential` or a SQL integrated login is usually not an impersonation bug. Ask three questions. First hop: did the browser reach IIS with Kerberos at all? A Negotiate header alone does not prove it; `klist` on the client should show a ticket for the site's HTTP SPN, and NTLM cannot be delegated. Delegation: is the application pool account allowed to delegate, by constrained delegation, to that exact target SPN, and is the user account not marked as sensitive? Backend authorization: does the share, NTFS or SQL permission actually grant this user? Fixture F04 is that case; change code only after all three answers are in.

Then decide. Delegated user access keeps the backend ACL authoritative but needs constrained delegation configured for every target, never unconstrained. A narrowly privileged service identity moves the decision into AccessOps: the service checks the grant, the tenant and the record, and an audit entry names the person on whose behalf it read. The decision record says which one AccessOps uses and why.

Be honest about where each proof comes from. Impersonated reads, ACL outcomes, the .NET Framework comparison and the second hop need IIS with Windows Authentication and domain accounts: they are verified in the lab environment only. On a developer machine the PolicyHarness proves what comes first: the application decision that must allow the read before any impersonation scope opens.

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:

- ✓ Explain the difference between the Windows identity attached to a request and the effective token used at a file operation, and prove access with an ACL rather than a user name.
- ✓ Configure scoped impersonation on .NET Framework and compare the Wisej.NET impersonate setting, its ImpersonationBegin and ImpersonationEnd events and Application.StartTask across HTTP and WebSocket events.
- ✓ Implement a bounded WindowsIdentity.RunImpersonated read on ASP.NET Core with a server-selected canonical path, and verify the token reverts after an exception.
- ✓ Classify a second-hop failure — SPN, constrained delegation target or backend ACL — and decide between delegated user access and a restricted service identity with explicit audit.

Lab / Exam Guide

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

Hands-on lab

Add a confidential-documents reader to AccessOps with two folders, one readable only by the application pool and one only by a named domain lab user, and a ScopedDocumentReader built from the WindowsScopedRead (ASP.NET Core) or Net48ScopedRead (.NET Framework) helper that opens a file chosen by server-side ID. Record the app subject, request identity and effective identity before, inside and after the scoped read, force an exception on a missing approved fixture and show the ordinary identity resumes, compare the Wisej.NET impersonate setting off and on for a click and an Application.StartTask in HTTP and WebSocket modes on .NET Framework, and, where a second server exists, classify a remote SMB or SQL integrated failure by first hop, delegation and backend authorization. Impersonated reads, ACL outcomes and the second hop are verified only in the lab environment with IIS Windows Authentication and domain accounts; on a developer machine the PolicyHarness proves the application decision that must allow the read before any impersonation scope opens.

Starting point

Open AccessOps as Module 2 left it: the intranet site on IIS with Windows Authentication, the Module 1 resource policy behind the document services, and the PolicyHarness running green. In the lab environment, create the two fixture folders under one approved root and grant the pool account and the domain lab user their separate read rights; on a developer machine, work in the PolicyHarness and the reader's code until you reach that environment.

Suggested approach

- ✓ Add a document catalog that maps server-side document IDs to canonical paths under the approved root, and make `ScopedDocumentReader` call the resource policy before it resolves any path.
- ✓ Wrap the read in `WindowsIdentity.RunImpersonated` (or `using (identity.Impersonate())` on .NET Framework) and record `WindowsIdentity.GetCurrent()` before, inside and after, next to the app subject and the request identity.
- ✓ Read the pool-only and the user-only fixture outside and inside the scope, then request an approved document whose file is missing and confirm the "after" row shows the pool identity again (F03).
- ✓ In the .NET Framework sandbox, set `"impersonate": true`, attach logging handlers to `ImpersonationBegin` and `ImpersonationEnd` once at start-up, and repeat the observations for a click and an `Application.StartTask` with WebSocket on and off.
- ✓ Where a file server or SQL Server is available, reproduce F04 and record the first hop, the delegation target and the backend ACL separately before writing the migration decision.

Required deliverables

- ✓ ScopedDocumentReader with a server-resolved canonical path

- ✓ Before, inside and after identity and outcome matrix
- ✓ Forced-failure reversion trace
- ✓ Impersonate setting and StartTask comparison across HTTP and WebSocket
- ✓ Second-hop classification and migration decision for the remote resource

Student review questions

- ✓ Your trace shows the lab user's name inside the scope, yet both fixture files open in every row. What does that tell you about the evidence, and what do you check first?
- ✓ On .NET 10 the `ImpersonationBegin` handler logs on every click. Why does the file read still run as the pool, and what would have to change for it not to?
- ✓ The remote share denies only through AccessOps while the same user opens it from their desktop. Which of the three second-hop checks does that rule out, and which two remain?

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.