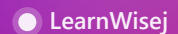


Hardening Checklist and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

HTTPS and security headers, cookie flags, session settings, dependency updates, rate limiting, deployment hardening and the AccessOps security review. What this means for a developer securing a production Wisej.NET app — and how to read this module.



Production focus

HTTPS and security headers, cookie flags, session settings, dependency updates, rate limiting, deployment hardening and the AccessOps security review.

1. The production problem

AccessOps now verifies passwords against a salted hash, resolves permissions in a service, streams documents only through `DocumentService` and audits every approval. None of that helps if the first request travels in clear text, if a script can read the session cookie, or if the build ships a package with a published advisory. The last mile belongs to the host: the certificate, the response headers, the cookie flags, the request limits, and the version of everything in the dependency graph.

2. Key concepts

Concept	Why it matters
HSTS	A <code>Strict-Transport-Security</code> header telling the browser to refuse plain HTTP for this host for a stated time, so a later visit is never intercepted on the redirect.
Content-Security-Policy	A header listing the origins a page may load scripts, styles and frames from; it turns an injected script into a blocked request instead of code running in a live session.
frame-ancestors	The CSP directive (<code>X-Frame-Options</code> is the older equivalent) that stops another site framing AccessOps and tricking a manager into clicking Approve.
cookie flags	<code>Secure</code> keeps the session cookie off plain HTTP, <code>HttpOnly</code> keeps it away from JavaScript, and <code>SameSite</code> stops it riding a request another site started.
request limits	Host-level caps on body and header size that reject an oversized upload before a line of AccessOps code runs.
token bucket	A counter that grants an attempt only while it holds a token and refills at a fixed rate, so traffic may burst a little and then slows to a crawl.
dependency audit	A repeatable check of every direct and transitive package, and of the runtime, against published advisories — recorded with a date and a decision.

3. Lesson narrative

HTTPS is the host's job, HSTS is the browser's

TLS is terminated outside your C#: by IIS with a bound certificate, by Kestrel configured with one, or by the reverse proxy in front of both. Your part is to make plain HTTP useless. Redirect every HTTP request at the host, then send `Strict-Transport-Security` with a long `max-age` so the browser stops attempting the insecure scheme at all. Add `includeSubDomains` only once every AccessOps subdomain really serves TLS, and treat `preload` as a decision you cannot quickly undo. One deployment detail catches people out: behind a proxy the request reaches the application unencrypted, so unless forwarded-headers handling is enabled the app believes it is insecure and marks nothing `Secure`.

Headers are the cheapest controls in this course, and they are set once for the pipeline rather than per screen. `X-Content-Type-Options: nosniff` stops a browser guessing that an uploaded document is really HTML. `Referrer-Policy: strict-origin-when-cross-origin` stops a document identifier leaking into another site's logs. `frame-ancestors 'none'` refuses framing outright. Set them where every response passes, including the signed share-link handler from Module 5 — a separate URL, and the one people forget.

Content-Security-Policy needs care here, because the browser is running the Wisej.NET client. That client, its theme resources and your widget scripts come from the same origin, so `default-src 'self'` covers them; the layout engine writes inline styles, so `style-src` usually needs `'unsafe-inline'`; and interop that builds client code at runtime forces you to widen `script-src`. Deploy as `Content-Security-Policy-Report-Only` first, drive every screen, read the violations, then enforce.

```

app.UseForwardedHeaders();           // the proxy terminated TLS, not us
app.UseHttpsRedirection();
app.UseHsts();                       // Strict-Transport-Security
app.Use(async (ctx, next) =>
{
    var h = ctx.Response.Headers;
    h["X-Content-Type-Options"] = "nosniff";
    h["Referrer-Policy"] = "strict-origin-when-cross-origin";
    h["Content-Security-Policy-Report-Only"] = AccessOpsCsp;
    await next();
});
// the Wisej.NET handler runs after this, so every response carries the headers

```

Cookies, session limits, and a button that is not an endpoint

Once login succeeds the session cookie is the credential, so it gets all three flags. **Secure** means it never travels without TLS. **HttpOnly** means a successful script injection in a custom widget still cannot read it. **SameSite** set to **Lax** or **Strict** means a request another site started does not carry it — the cheapest defence against a forged approval. On ASP.NET Framework these are settings on the cookie and session-state elements of **Web.config**; on ASP.NET Core they are cookie policy options. Verify them in the browser's developer tools, not by re-reading the config.

Session lifetime has two halves. Wisej.NET reads its own settings from the application's JSON configuration file beside the start page, and that is where the framework's session timeout belongs: the backstop that reclaims an abandoned session and its server-side state. It is not the idle logout you built in Module 2, because the client keeps its connection alive and an abandoned browser can look like traffic — idle must be measured from the last real user event, and explicit logout must dispose the **SessionContext** in **Application.Session** so the back button reaches a login page. Request limits are the third setting: cap the body and header size at the host so an oversized upload is rejected before **DocumentService** is entered.

Rate limiting splits along an important line. The share-link handler is a real URL, so middleware can limit it directly — a token-bucket limiter partitioned by client address, answering 429 without revealing whether the link existed. The login button is not a URL: every Wisej.NET event, from a grid scroll to a Sign in click, reaches the same application endpoint inside an existing session, so middleware cannot tell them apart. Login rate limiting therefore lives in the service, as a bucket keyed by the submitted user name and the client address and checked before the hash is verified. A refusal returns the same generic message as a wrong password, is audited with its correlation ID, and counts against a shared store rather than a static field when AccessOps runs on several servers.

Patched, and provably so

Run **dotnet list package --vulnerable --include-transitive** and read all of it: the package you never chose, pulled in three levels down, is the usual finding. Add **--outdated** and **--deprecated** for what is merely stale or abandoned. Recent .NET SDKs also check advisories during restore, which turns the audit into build output rather than a task somebody remembers. Put it in the pipeline so a new advisory fails a build instead of surfacing in a penetration test.

Packages are only part of the graph. The .NET runtime, the web host and the container base image each need a patch cadence and a recorded version. Wisej.NET itself is a versioned dependency: pin it, read the release notes before moving, and retest the theme and custom widgets after a major upgrade. Third-party JavaScript in the web root is invisible to the package graph, so list each file by hand with its version and origin. Then write the audit down — date, tool, findings, a decision for each (upgrade, mitigate, or accept with an expiry) and a next review date.

The pre-release review, module by module

The capstone deliverable is a review, not a feature. Take the control placement map from Module 1 and walk the course forwards, checking the app rather than the intention, and test as an attacker instead of as a user: request another department's document as an ordinary employee, replay a share link after its hour, call an approval method as somebody who is not the manager, submit twenty bad passwords in ten seconds.

- Module 2 — lockout, hashing, verified identity, idle timeout, logout that disposes state.
- Module 3 — every service method guarded, department and manager rules below the UI, the forged request refused.
- Module 4 — PKCE, ID token and nonce validation, claim mapping, sign-out ending both sessions.
- Module 5 — upload validation, downloads through the service, signed expiring links, parameterised queries, encoded output.
- Module 6 — audit rows for sensitive commands, correlation IDs in the logs, no secrets or personal data in either.
- Module 7 — HTTPS and HSTS, the header set, cookie flags, session and request limits, rate limiting, a dated audit.

Every finding gets a severity, an owner and either a fix or an accepted risk with an expiry; a checklist with unexplained blanks is not complete. Sign-off means one thing: every boundary drawn in Module 1 now has a control that a service, not a screen, enforces.



Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.



Learning objectives

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

- ✓ Configure HTTPS, HSTS, secure cookie flags and security headers for a Wisej.NET deployment.
- ✓ Tune session settings, request limits and rate limiting to resist abuse.
- ✓ Keep dependencies and the runtime patched with a repeatable process.
- ✓ Run a pre-release security review against a checklist and deliver the AccessOps capstone.

Lab / Exam Guide

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



Hands-on lab

Complete the AccessOps capstone: enforce HTTPS with HSTS and set security headers, mark cookies Secure and HttpOnly with a strict SameSite policy, set session timeout and request size limits, add rate limiting on login and share links, run a dependency audit and record the results, then walk the full checklist from this course against the app and fix every finding.

Starting point

Open the AccessOps solution as it stands after Module 6: a login page with logout, a `SessionContext` holding the verified identity, a `PermissionService` guarding `DocumentService` and `ApprovalService`, external sign-in through OpenID Connect, an upload and download pipeline with a signed share-link handler, and an `AuditService` writing correlation-stamped rows. Nothing in the host configuration has been touched yet: the app answers on plain HTTP in development, its cookies carry no flags, and no package has been checked against an advisory.

Suggested approach

- ✓ Bind a certificate and redirect HTTP to HTTPS at the host, add `Strict-Transport-Security`, and confirm the forwarded-headers handling if a reverse proxy sits in front of AccessOps.
- ✓ Add one place in the pipeline that sets `X-Content-Type-Options`, `Referrer-Policy`, `frame-ancestors` and a Content-Security-Policy, starting in report-only mode until every AccessOps screen and the share-link handler load clean.
- ✓ Mark the session cookie `Secure` and `HttpOnly` with a strict `SameSite` policy, set the Wisej.NET session timeout in the application configuration file, and cap the request body size at the host.
- ✓ Put a token bucket keyed by user name and client address inside the login service, and a middleware limiter on the share-link URL, auditing every refusal with its correlation ID.
- ✓ Run `dotnet list package --vulnerable --include-transitive`, record findings with a decision and an owner, then walk the seven-module checklist against the running app and fix what it finds.

Required deliverables

- ✓ HTTPS, HSTS and security header configuration
- ✓ Cookie and session hardening settings
- ✓ Rate limiting on login and share links
- ✓ Dependency audit report
- ✓ Completed security review checklist with fixes

Student review questions

- ✓ Middleware can rate limit the share-link handler but cannot rate limit the Sign in button. What is different about the two requests in a Wisej.NET app, and where does the login limit have to live instead?
- ✓ Your Content-Security-Policy is enforcing and the grid renders unstyled. Which directive is the likely cause, and how would report-only mode have told you before the deployment did?
- ✓ The dependency audit lists a vulnerable package nobody in AccessOps references directly. How did it get into the build, and what are the two honest ways to close the finding?



Security habit

The browser is never trusted. Every decision that matters — who the user is, what they may do, which record they may see — is made on the server in a service, and the UI only reflects it. If a rule exists only as a hidden button, it does not exist.