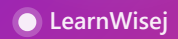


Protecting Data, Uploads, Downloads and Handlers

Lesson Guide · Lab / Exam Guide



Lesson Guide

Authorizing file uploads and downloads, safe custom HTTP handlers, input validation, output encoding, injection and cross-site risks in a server-side UI. What this means for a developer securing a production Wisej.NET app — and how to read this module.



Production focus

Authorizing file uploads and downloads, safe custom HTTP handlers, input validation, output encoding, injection and cross-site risks in a server-side UI.

1. The production problem

The confidential documents area is the part of AccessOps an attacker actually wants: salary letters, disciplinary files, signed contracts, board packs. Every other screen moves rows between grids; this one moves bytes across the boundary to a browser you do not control, and back. Identity and permissions were settled in the earlier modules, and this one is about the places where a file, a URL or a string walks around them.

2. Key concepts

Concept	Why it matters
server-streamed download	Calling <code>Application.Download</code> from server code streams a file to the browser during an event you already authorized, so it has no address to bookmark, share or guess.
static file exposure	Anything under the web root is served before a line of your code runs, so the file's permissions are exactly "whoever knows the URL".
Uploaded event	The <code>Upload</code> control raises it on the server with the incoming file streams — the only place a file can be inspected before it joins your system.
content signature check	Comparing a file's first bytes with the format's known header, because the extension and the declared content type are supplied by the client.
quarantine storage	Holding the accepted upload outside the web root under a generated name until the scan is clean, and only then linking it to the owning department.
custom HTTP handler	A server class registered with the application that receives raw requests outside the event flow, so it has no session identity and must authenticate and validate everything itself.
signed expiring link	A URL carrying a document id, an expiry and an HMAC computed over both with a server-held key, so the values cannot be edited and the link stops working on its own.

3. Lesson narrative

A download with no address

The safest thing about a Wisej.NET download is that there is nothing to link to. Server code opens a stream, calls `Application.Download` with that stream and the file name the browser should save, and the bytes travel down the existing connection as an attachment. The transfer happens inside an already authenticated session, in an event handler that has already run your checks: no path in the markup, no URL in the history, nothing to forward.

That property is easy to throw away. Putting `/files/contract-4471.pdf` in a link hands the feature to the web server, which serves it to anyone at all, and sequential names invite the next caller to try 4472. In AccessOps the files live outside the published folder under names the application generates, and the only way in is `DocumentService.Download`.

The order inside that service method matters. Load the document by id through a parameterised query, treating the id as untrusted input because it came from a grid row in a browser. Ask the permission rules from Module 3 — department ownership and manager scope included — whether this caller may read this record, and throw if not. Write the audit entry. Only then open the stored file and stream it. The button handler only calls the service, so the same rule holds from the grid, an approval screen or a scheduled report.

```

public void Download(int documentId)
{
    var doc = _repository.FindById(documentId);           // parameterised
    if (doc == null || !_permissions.CanRead(_session.User, doc))
        throw new SecurityException("Document not available.");

    _audit.RecordDownload(_session.User.Id, doc.Id);

    using var file = _storage.OpenRead(doc.StoredName); // outside the web root
    Application.Download(file, doc.DisplayName);
}

```

An upload is untrusted input that happens to be a file

The `Upload` control posts the selected files to the server and raises `Uploaded`, where your handler receives the file streams. Everything the event tells you about them came from the browser: the name, the extension, the declared content type, the size. A hostile client can send an executable called `payslip.pdf`, or a name full of parent-directory segments that walks out of the folder you intended.

So the handler is a gate with five checks, in order. Enforce a maximum size in the control and in the server request limits. Check the extension against an allow-list of what the feature needs, never a deny-list of what you fear. Read the first bytes and confirm they match that format's signature, so a renamed executable is rejected even though its extension looked fine. Write the file outside the web root under a name your code generates, keeping the user's original name as a display value. Finally, quarantine it, run the malware scan, and add it to the department's documents only when the scan comes back clean.

- Size and extension rejected before the stream is read to the end.
- Signature checked against the allow-listed formats, not the declared type.
- Stored name generated by the server; the client name is never a path.

Every rejection is a message the user can act on — "PDF and Word documents up to 20 MB" — plus an audit entry naming the caller, the file and the reason. Silence is worse than refusal.

Handlers stand outside the session

Sometimes a link has to work without the portal: a supplier needs the signed contract, a manager forwards one document to an auditor. That is a custom HTTP handler, a server class registered with the application that receives the raw request. It is not part of a Wisej.NET session, so nothing about the caller has been established. Whatever the handler will answer, it will answer to the whole internet.

AccessOps issues a share link as a document id, an expiry and a signature — an HMAC over the id and the expiry with a key held only on the server. The handler parses the id as an integer, recomputes the HMAC from the query string values, compares the two in fixed time, checks the expiry, and only then loads the record and streams the file. Editing the id or pushing the expiry out changes the signature, so the request is refused. Expired and forged links get the same short response, both logged with the caller's address so a scan for valid ids becomes visible.

The rest of the handler rules follow from the same fact: validate every input rather than trusting a caller you cannot see, return no detail in the error response, keep the lifetime short — one hour in the lab — and

rate-limit the endpoint so documents cannot be enumerated at leisure.

Injection in a server-side UI

Server-side controls remove a great deal of ordinary web risk, because your C# sets properties on objects rather than concatenating markup. They do not remove injection, which follows the data. Every query in AccessOps uses parameters — EF Core parameterises LINQ automatically, raw SQL takes explicit parameters — so a search box containing a quote and a semicolon is a search term, not a statement. Concatenation into SQL is the one shortcut this course never allows.

The second path is output. Most Wisej.NET controls render a caption as text, but a property that is allowed to carry markup will render it, so a label with `AllowHtml` enabled and an employee-supplied name in it becomes a script host. Keep those properties off for user content, or encode the value before assigning it; the file names you echo back after an upload came from the browser too.

The third path is JavaScript interop. `Application.Eval` and `Application.Call` send script to the client, and building that script by pasting in a user string is the same mistake as building SQL by hand. Pass values as arguments to a function that already exists on the page, and let the client-side code treat them as data.



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:

- ✓ Authorize every download through a service and stream files with `Application.Download` rather than exposing paths or URLs.
- ✓ Validate uploads by size, type and content, store them outside the web root and scan or quarantine them.
- ✓ Write custom HTTP handlers and endpoints that authenticate the caller and validate every input.
- ✓ Prevent injection and script risks: parameterised queries, encoded output, safe HTML in labels and safe JavaScript interop.

Lab / Exam Guide

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



Hands-on lab

Secure the AccessOps documents feature: uploads limited by size and type with content sniffing, stored outside the web root with a random name and linked to the owning department, downloads that go through DocumentService authorization and Application.Download, a custom handler for a signed share link that expires in one hour, and a review that every query is parameterised and every user-provided string is encoded before display.

Starting point

You open the AccessOps solution as it stood at the end of Module 4: a working login with an external identity provider, a SessionContext carrying the verified identity, and a PermissionService that resolves roles, claims and the department and manager scope rules. The Documents screen already lists the confidential documents for the current department, and it already has an Upload control and a Download button — both of which currently do the naive thing, writing files into a folder under the web root and streaming back whatever file name the grid row happens to hold.

Suggested approach

- ✓ Move the document store out of the published folder to a configured path outside the web root, give every stored file a server-generated name, and keep the user's original file name in the database as a display value only.
- ✓ Rewrite the Uploaded handler as a gate: enforce the size limit, check the extension against an allow-list, read the leading bytes and compare them with the format signature, then write the accepted file to the quarantine folder and link it to the owning department only after the scan passes.
- ✓ Put every download behind DocumentService: load the record by id with a parameterised query, call the permission check for the caller and that record, write an audit entry, then stream the file with Application.Download and leave the button handler with nothing but the service call.
- ✓ Add the share-link handler: generate a URL carrying the document id, an expiry one hour ahead and an HMAC over both, then have the handler recompute the signature with a fixed-time comparison, reject expired or tampered links with the same short response, and log every refusal.
- ✓ Walk the feature for injection: confirm every query uses parameters, confirm no user-supplied string reaches an HTML-capable property or an Application.Eval call unencoded, and finish with the forged download attempt from an account in another department.

Required deliverables

- ✓ Validated upload pipeline with quarantine folder
- ✓ Authorized download through the service

- ✓ Signed, expiring share-link handler
- ✓ Parameterised query and output encoding review
- ✓ Test with a forged download request

Student review questions

- ✓ Your upload handler checks the extension against an allow-list and the file passes. Explain what an attacker can still send, and what the signature check adds that the extension check cannot.
- ✓ A manager forwards a share link to somebody outside the company before it expires. Which parts of your design limit the damage, and what would you have to add to revoke that single link without changing the key for everyone?
- ✓ Your download button is hidden for users without the read permission, and the forged request test still passes. Explain precisely which line of code caused it to pass, and why the hidden button had nothing to do with it.



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.