Awesome—here’s the **what** and the **why** behind the hardened Web.config we set up, plus a few careful trade-offs and gotchas to watch for in a WebForms/IIS environment.

# **What I changed**

## **1) Remove stack-leaking banners**

<system.web>

<httpRuntime enableVersionHeader="false" />

</system.web>

<system.webServer>

<httpProtocol>

<customHeaders>

<remove name="X-Powered-By" />

<remove name="X-AspNet-Version" />

<!-- (optional if you use MVC anywhere)

<remove name="X-AspNetMvc-Version" /> -->

</customHeaders>

</httpProtocol>

<!-- (Optional fallback) Hide "Server" via URL Rewrite if registry isn’t allowed -->

<rewrite>

<outboundRules>

<rule name="Remove Server Header">

<match serverVariable="RESPONSE\_Server" pattern=".+" />

<action type="Rewrite" value="" />

</rule>

</outboundRules>

</rewrite>

</system.webServer>

### **Why**

* Server, X-Powered-By, X-AspNet-Version (and sometimes X-AspNetMvc-Version) make it easier to fingerprint your exact stack and version. That speeds up targeted exploits.
* enableVersionHeader="false" prevents ASP.NET from emitting X-AspNet-Version.
* customHeaders <remove …> strips common banners added by IIS/ASP.NET.

The *most reliable* way to hide **Server** is the **HTTP.sys registry switch**:  
  
 reg add "HKLM\SYSTEM\CurrentControlSet\Services\HTTP\Parameters" /v DisableServerHeader /t REG\_DWORD /d 1 /f

iisreset

* I included the URL Rewrite fallback because sometimes registry edits aren’t allowed; the registry method is preferable.

## **2) Add core browser hardening headers**

<system.webServer>

<httpProtocol>

<customHeaders>

<add name="X-Frame-Options" value="SAMEORIGIN" />

<add name="X-Content-Type-Options" value="nosniff" />

<add name="Strict-Transport-Security" value="max-age=31536000; includeSubDomains" />

<add name="Content-Security-Policy" value="default-src 'self'" />

<add name="Referrer-Policy" value="no-referrer-when-downgrade" />

<add name="Permissions-Policy" value="geolocation=(), microphone=(), camera=()" />

<add name="X-XSS-Protection" value="1; mode=block" />

</customHeaders>

</httpProtocol>

</system.webServer>

### **Why (and key trade-offs)**

* **HSTS (Strict-Transport-Security)** forces HTTPS after first visit; prevents SSL-strip attacks.  
  + Chosen: 1 year with includeSubDomains. Only use this if **all** subdomains support HTTPS.
  + If you might break legacy subdomains, start with a smaller max-age (e.g., a week) and then increase.
* **CSP (Content-Security-Policy)** is your **#1 defense** against XSS.  
  + Chosen baseline: default-src 'self' to allow only same-origin by default.
  + WebForms often emits inline scripts/styles. If your pages break, you’ll either:  
    - temporarily allow 'unsafe-inline' for script-src/style-src while you refactor, or
    - move inline JS to external files and/or adopt nonces/hashes (harder in WebForms but possible).

Safer rollout path: use **Report-Only** first:  
  
 <add name="Content-Security-Policy-Report-Only" value="default-src 'self'" />

* + Then tighten and switch to enforcing once violations are resolved.
* **X-Content-Type-Options: nosniff** blocks MIME sniffing (stops browsers from executing non-JS as JS, etc.).
* **X-Frame-Options: SAMEORIGIN** blocks clickjacking.  
  + Modern alternative in CSP: frame-ancestors 'self'. You can use both for coverage.
* **Referrer-Policy: no-referrer-when-downgrade** avoids leaking full URLs to non-HTTPS targets.  
  + You can go stricter (e.g., strict-origin-when-cross-origin) if analytics don’t need full referrers.
* **Permissions-Policy** (replaces the old Feature-Policy) disables powerful features you don’t use (geo/mic/camera).
* **X-XSS-Protection: 1; mode=block** is kept only for legacy browsers.  
  + Modern Chromium/Safari ignore it; it’s *not* a substitute for CSP/encoding. We include it for backward compatibility in locked-down environments.

# **Why these choices fit WebForms/IIS specifically**

* **WebForms + ViewState**: XSS risk tends to come from rendering unencoded user input in server controls. CSP plus strict output encoding mitigates this. (Keep ValidateRequest on and use proper HtmlEncode.)
* **Inline scripts/styles**: WebForms controls frequently render inline JS/CSS. A strict CSP breaks these. That’s why we start with a **tight default** but expect to tune script-src/style-src with nonces/hashes or a temporary 'unsafe-inline'.
* **IIS “Server” header**: Best hidden at the **HTTP.sys** layer (registry). Outbound rewrite is a fallback.
* **Cluster without internet**: Everything here is **self-contained** (no CDNs required). If you do use CDNs, you must list them in CSP (script-src https://cdn.example.com 'self').

# **How to verify (offline, in Postman)**

1. Send a GET to a few .aspx endpoints.
2. Confirm **present**: X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security, Content-Security-Policy, Referrer-Policy, Permissions-Policy, (optional) X-XSS-Protection.
3. Confirm **absent**: Server, X-Powered-By, X-AspNet-Version (and X-AspNetMvc-Version if applicable).

Optional Postman Tests script:  
  
 pm.test("Security headers present", function () {

pm.response.to.have.header("X-Frame-Options");

pm.response.to.have.header("X-Content-Type-Options");

pm.response.to.have.header("Strict-Transport-Security");

pm.response.to.have.header("Content-Security-Policy");

pm.response.to.have.header("Referrer-Policy");

pm.response.to.have.header("Permissions-Policy");

});

pm.test("No banner leakage", function () {

pm.expect(pm.response.headers.has("Server")).to.eql(false);

pm.expect(pm.response.headers.has("X-Powered-By")).to.eql(false);

pm.expect(pm.response.headers.has("X-AspNet-Version")).to.eql(false);

});

## **Postman Tests Script**

Paste this into the **Tests tab** of your request in Postman:

// --- Check Security Headers ---

pm.test("Security headers are present", function () {

var headers = pm.response.headers;

pm.expect(headers.has("X-Frame-Options")).to.eql(true);

pm.expect(headers.has("X-Content-Type-Options")).to.eql(true);

pm.expect(headers.has("Strict-Transport-Security")).to.eql(true);

pm.expect(headers.has("Content-Security-Policy")).to.eql(true);

pm.expect(headers.has("Referrer-Policy")).to.eql(true);

pm.expect(headers.has("Permissions-Policy")).to.eql(true);

pm.expect(headers.has("X-XSS-Protection")).to.eql(true);

});

// --- Ensure banners are hidden ---

pm.test("Server banners are NOT exposed", function () {

var headers = pm.response.headers;

pm.expect(headers.has("Server")).to.eql(false);

pm.expect(headers.has("X-Powered-By")).to.eql(false);

pm.expect(headers.has("X-AspNet-Version")).to.eql(false);

pm.expect(headers.has("X-AspNetMvc-Version")).to.eql(false);

});

// --- Check Content-Security-Policy rules ---

pm.test("CSP allows only intended sources", function () {

var csp = pm.response.headers.get("Content-Security-Policy");

pm.expect(csp).to.include("default-src 'self'");

pm.expect(csp).to.include("script-src 'self' 'unsafe-inline'");

pm.expect(csp).to.include("img-src 'self' http://armanent http://drdointranet data:");

});

# **Common pitfalls & tips**

* **Load balancer / reverse proxy**: Some devices re-add a Server header. If you still see it after hardening IIS, remove it on the LB as well.
* **CSP breakage**: Start with Report-Only, inspect violations, then enforce. Expect to whitelist your own sub-paths or CDNs if any.

**Cookies**: For defense-in-depth, also add:  
  
 <system.web>

<httpCookies httpOnlyCookies="true" requireSSL="true" sameSite="Strict" />

<!-- If using Forms Auth:

<authentication mode="Forms">

<forms requireSSL="true" slidingExpiration="true" cookieless="UseCookies" protection="All" />

</authentication> -->

</system.web>

* This pairs well with HSTS and prevents cookie theft over HTTP.
* **CSP advanced hardening** (optional later):  
  + script-src 'self' 'nonce-<dynamic>' (requires generating a nonce per request and adding it to allowed inline scripts).
  + Add frame-ancestors 'self' and drop X-Frame-Options eventually.
  + Consider object-src 'none', base-uri 'self', img-src 'self' data: (if you embed data URLs), etc.

If you want, tell me whether your app uses **inline scripts**, **CDNs**, or **iframes**—I’ll craft a **CSP tailored to your pages** that protects against XSS without breaking functionality.

Perfect 👍 thanks, Aditya — that gives me clarity:

* ❌ You don’t want **any server banners** visible (Server, X-Powered-By, X-AspNet-Version).
* ✅ You have **inline scripts** (from WebForms).
* ✅ You have **images** coming from:  
  + Your own cluster domains (armanent, drdointranet/...)
  + Project’s own /images/... folders.

So let’s build you a hardened config tailored to this.

# **🔹 1. Hiding All Server Banners**

### **In Web.config**

<system.web>

<!-- Prevent X-AspNet-Version -->

<httpRuntime enableVersionHeader="false" />

</system.web>

<system.webServer>

<httpProtocol>

<customHeaders>

<!-- Remove common banners -->

<remove name="X-Powered-By" />

<remove name="X-AspNet-Version" />

<remove name="X-AspNetMvc-Version" />

</customHeaders>

</httpProtocol>

<!-- Outbound rule to wipe Server header -->

<rewrite>

<outboundRules>

<rule name="Remove Server Header">

<match serverVariable="RESPONSE\_Server" pattern=".+" />

<action type="Rewrite" value="" />

</rule>

</outboundRules>

</rewrite>

</system.webServer>

### **Best Method (IIS / HTTP.sys registry)**

For IIS itself:

reg add "HKLM\SYSTEM\CurrentControlSet\Services\HTTP\Parameters" /v DisableServerHeader /t REG\_DWORD /d 1 /f

iisreset

This guarantees Server: never appears in responses.

# **🔹 2. Security Headers (With Inline Script & Cluster Images in Mind)**

<system.webServer>

<httpProtocol>

<customHeaders>

<!-- Core protections -->

<add name="X-Frame-Options" value="SAMEORIGIN" />

<add name="X-Content-Type-Options" value="nosniff" />

<add name="Strict-Transport-Security" value="max-age=31536000; includeSubDomains" />

<add name="Referrer-Policy" value="strict-origin-when-cross-origin" />

<add name="Permissions-Policy" value="geolocation=(), microphone=(), camera=()" />

<add name="X-XSS-Protection" value="1; mode=block" />

<!-- CSP tailored for your case -->

<add name="Content-Security-Policy"

value="default-src 'self';

script-src 'self' 'unsafe-inline';

img-src 'self' http://mentra http://pokeintranet data:;

style-src 'self' 'unsafe-inline';

frame-ancestors 'self';

object-src 'none';

base-uri 'self';" />

</customHeaders>

</httpProtocol>

</system.webServer>

# **🔹 3. Why This CSP?**

* default-src 'self' → by default, load only from same origin.
* script-src 'self' 'unsafe-inline' → allows inline scripts **(required for WebForms)**.  
  + ⚠️ Safer long-term: migrate inline JS to external .js files and replace with nonces instead of 'unsafe-inline'.
* img-src 'self' http://armanent http://drdointranet data:  
  + allows images from your app itself, your **cluster domains**, and data: URLs (useful for embedded base64 logos).
* style-src 'self' 'unsafe-inline' → WebForms also tends to emit inline CSS, so we allow it.
* frame-ancestors 'self' → prevents other sites from iframing your pages (anti-clickjacking).
* object-src 'none' → blocks Flash/ActiveX (rarely needed today).
* base-uri 'self' → prevents attackers from overriding <base> tag to trick URL resolution.

# **🔹 4. Verification in Postman (Cluster)**

1. Restart IIS (iisreset).
2. Open Postman → call http://yourclusterapp/Login.aspx.
3. In **Response Headers**, you should now see:  
   * ✅ Security headers listed.
   * ✅ CSP matching above.
   * ❌ No Server, X-Powered-By, X-AspNet-Version.

✅ This config **hides banners completely** and gives you a **practical CSP** that works with inline scripts + cluster-hosted images.  
 Later, if you want **maximum XSS protection**, we can remove 'unsafe-inline' and replace it with a **nonce-based CSP** (more secure, but requires code changes in WebForms).

👉 Do you want me to prepare a **stricter CSP upgrade path** (step-by-step) so you can gradually move from 'unsafe-inline' to a fully locked-down CSP without breaking your WebForms pages?

Perfect 👌 Let’s break this down very clearly. You’re asking about **security headers** (like the ones we added in Postman or web.config) and how they protect your **WebForms app**. I’ll explain **each header**, its **purpose**, and give you **examples** that fit your case (IIS/ASP.NET WebForms, Chrome/Firefox usage, inline scripts, images from cluster/project files, etc.).

## **🔐 1. Strict-Transport-Security (HSTS)**

👉 Forces browsers to **always use HTTPS** instead of HTTP.

* Without this: A user might type http:// and their traffic could be hijacked.
* With this: Browser remembers to always use HTTPS.

Example response header:

Strict-Transport-Security: max-age=31536000; includeSubDomains

* max-age=31536000 → 1 year
* includeSubDomains → applies to all subdomains (armanetwork/..., drdointranet/...).

✅ If your cluster already runs HTTPS, this **prevents downgrade attacks**.

## **🧑‍💻 2. X-Content-Type-Options**

👉 Stops browsers from “guessing” file types.

* Without this: If you upload an image but it has hidden JS, browser might treat it as script.
* With this: Browser **trusts only the declared Content-Type** (e.g., image/png).

Example response header:

X-Content-Type-Options: nosniff

✅ Protects against attacks via **images, CSS, JS served from project files**.

## **📦 3. X-Frame-Options**

👉 Stops your app from being **embedded in iframes** (clickjacking).

* Without this: Attacker could open your app inside their site and trick users into clicking invisible buttons.
* With this: Your app can’t load in an iframe (unless you allow).

Example response header:

X-Frame-Options: SAMEORIGIN

* SAMEORIGIN → allows only your domain.
* DENY → blocks all framing.

✅ Helps if your app is internal (drdointranet) and should not be embedded outside.

## **🖼️ 4. Content-Security-Policy (CSP)**

👉 Controls where scripts, images, styles can load from.

* Since you said you have **inline scripts** and images from **cluster/project**, we must allow those.
* CSP blocks XSS by stopping attacker-injected scripts from running.

Example response header:

Content-Security-Policy: default-src 'self'; img-src 'self' armanet drdointranet; script-src 'self' 'unsafe-inline'

Breakdown:

* default-src 'self' → all resources load from your own domain.
* img-src 'self' armanet drdointranet → allow images from your cluster servers.
* script-src 'self' 'unsafe-inline' → allow inline scripts (since you already use them).  
   *(better is to remove 'unsafe-inline' and put scripts in .js files, but I kept it for compatibility).*

✅ This is your **main XSS defense** for Chrome/Firefox.

## **🛡️ 5. X-XSS-Protection (⚠️ legacy, only IE/old Edge)**

👉 Used to tell browser’s **built-in XSS filter** what to do.  
 Example:

X-XSS-Protection: 1; mode=block

* 1 = enable
* mode=block = don’t load page if XSS detected.

⚠️ Chrome/Firefox have **deprecated** this (they ignore it). But doesn’t harm to include.

## **🕵️ 6. Referrer-Policy**

👉 Controls how much referrer info is sent when navigating away.  
 Example:

Referrer-Policy: no-referrer-when-downgrade

* Sends referrer only on HTTPS → HTTPS.
* Doesn’t leak sensitive URLs when going to HTTP.

✅ Protects internal app paths (/drdointranet/xyz.aspx) from leaking outside.

## **🚫 7. Server Header Removal**

👉 By default, IIS sends:

Server: Microsoft-IIS/10.0

* Attackers can use this to target known IIS vulnerabilities.
* Best practice: **remove or hide it**.

In IIS:

Remove-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter "system.webServer/security/requestFiltering" -name "removeServerHeader"

✅ Now attackers can’t fingerprint your server easily.

# **🔎 Example: How headers protect in Chrome/Firefox**

Imagine:

* You host a WebForms app at https://drdointranet/app/.
* Someone tries to inject <script>alert(1)</script> into an input field.

With headers:

* **CSP** blocks it (script not from 'self').
* **X-Content-Type-Options** ensures that uploaded images aren’t executed as JS.
* **X-Frame-Options** stops clickjacking.
* **HSTS** ensures user is never redirected to http://.
* **Referrer-Policy** hides sensitive internal URLs when clicking outside links.
* **No Server banner** hides your IIS version.

👉 Would you like me to now **write a Postman test script** that verifies each of these headers in the response, so you can directly check whether they’re working in your cluster?