Security Headers: The Silent Gatekeepers of Web Applications

Table of Contents

1. Purpose of This Document

The purpose of this document is to provide a structured understanding of **HTTP Security Headers** and their importance in enhancing the security posture of web applications. This document serves as a reference for security teams, developers, and application architects who are responsible for ensuring that web applications are safeguarded against common threats through the proper configuration of these headers.

The goal is to create a clear and concise resource that not only lists the recommended security headers but also explains why they are necessary, what security benefits they offer, and how they contribute to the overall Defense strategy of an application. By consolidating this information into a single document, teams can maintain consistency in security practices across different environments and projects.

In the rapidly evolving digital landscape, web applications face a growing range of security risks — from cross-site scripting (XSS) and clickjacking to data leakage and cross-origin attacks. While application code and infrastructure security are essential, HTTP Security Headers provide an additional layer of protection that operates at the browser level, reducing the risk of exploitation without altering the underlying codebase. This document highlights the role of security headers as a preventive measure that complements other security mechanisms.

Another key purpose of this document is to promote awareness. Many security breaches occur not because of highly sophisticated exploits, but due to the absence of basic, well-established safeguards. Security headers fall into this category they are simple to implement, cost-effective, and powerful when configured correctly. However, they are often overlooked in development workflows or misconfigured due to lack of awareness. This document aims to close that gap by making the importance of these headers explicit and by encouraging their consistent use in all web applications.

2. Scope

This document applies to all web applications, APIs, and web-based services developed, deployed, or maintained by the organization. It is relevant for development teams, security engineers, system administrators, DevOps personnel, and quality assurance testers involved in building or maintaining application security.

3. Why Every Application Should Use Security Headers

Every web application, regardless of size or complexity, is a potential target for attackers. Cyberattacks are not limited to large corporations; small and medium-sized applications are often targeted because they are perceived as having weaker security. Security headers are lightweight, easy to implement, and have minimal impact on performance, making them one of the most cost-effective security measures available.

Without these headers, an application leaves critical aspects of its security to browser defaults, which may not be strict enough to withstand modern attack techniques. By adopting security headers as part of standard application deployment, organizations not only comply with recognized security best practices but also build user trust by demonstrating a commitment to safeguarding their data.

In short, security headers act as silent guardians invisible to end users but vital in defending against many common threats. They are not optional add-ons; they are essential building blocks of a secure and resilient web application.

4. Introduction to HTTP Security Headers

HTTP Security Headers are special pieces of information sent by a web server to a web browser as part of the **HTTP response**. They are not visible to the end user but play a critical role in defining how a browser should behave when interacting with a website. In simple terms, they are a set of **rules and instructions** from the server telling the browser what is safe and what is not.

These headers form an essential part of a web application's **defensive strategy**, as they can prevent a wide range of common attacks such as Cross-Site Scripting (XSS), Clickjacking, protocol downgrades, insecure resource loading, and sensitive data exposure.

Basic Concept

When a user visits a website, their browser sends a request to the server. The server responds with both the web content (HTML, CSS, JavaScript, etc.) and metadata in the form of HTTP headers. Some headers handle technical aspects like caching or content length, but security headers specifically tell the browser how to securely handle the site's resources and interactions.

Example:

- A security header can tell the browser not to run scripts from untrusted sources.
- It can instruct the browser to only connect to the website over HTTPS.
- It can prevent the site from being embedded inside other pages, blocking clickjacking attempts.

Why it is important

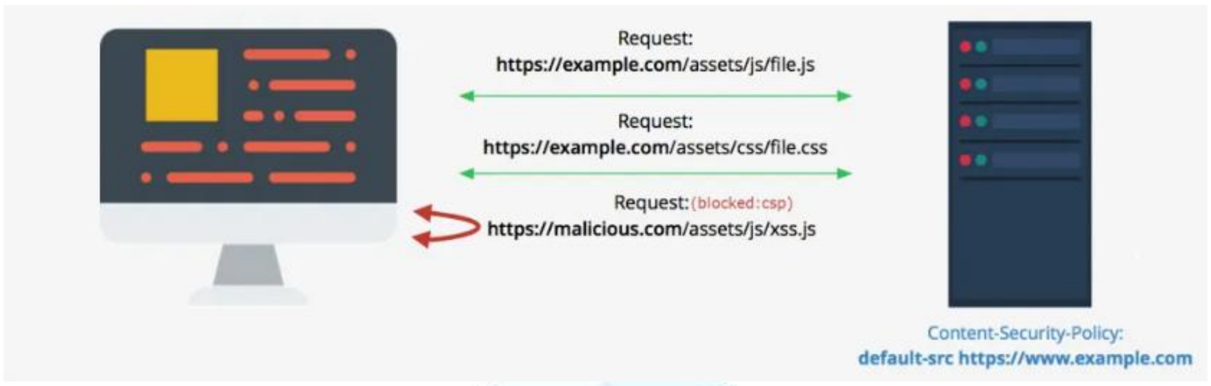
Without proper security headers, browsers use default behaviours that may not be secure. Attackers often exploit these defaults to inject malicious scripts, steal data, or trick users into interacting with harmful interfaces. Security headers reduce the attack surface by forcing strict security rules at the browser level even if an attacker manages to exploit part of the application.

5. Security Headers Included in This Document

1. Content-Security-Policy (CSP)
2. Strict-Transport-Security (HSTS)
3. X-Content-Type-Options
4. X-Frame-Options
5. X-XSS-Protection (Legacy, but included for awareness)
6. Referrer-Policy
7. Permissions-Policy (formerly Feature-Policy)
8. Cross-Origin Resource Sharing (CORS) Security Headers
9. Cache-Control (Security implications for sensitive data)
10. Cross-Origin-Resource-Policy (CORP)
11. Cross-Origin-Embedder-Policy (COEP)

Content-Security-Policy (CSP)

Content Security Policy (CSP) is a critical security control used to mitigate client-side attacks such as Cross-Site Scripting (XSS) and data injection.


Request:
https://example.com/assets/js/file.js
Request:
https://example.com/assets/css/file.css
Request: (blocked:csp)
https://malicious.com/assets/js/xss.js
Content-Security-Policy:
default-src https://www.example.com

Content-Security-Policy (CSP) is an HTTP security header that controls which resources a browser is allowed to load and execute on a web page. It acts like a whitelist, specifying trusted sources for scripts, styles, images, media, and other content.

CSP's primary goal is to **prevent content injection attacks**, especially **Cross-Site Scripting (XSS)**, by restricting unauthorized scripts from running in the browser.

What happens when we don't use CSP?

Without Content Security Policy, the browser executes all scripts, styles, and other resources loaded from the page. This includes content that may be injected by attackers through vulnerabilities in the application or through third-party components.

Risks include:

- Execution of **malicious JavaScript**, stealing user credentials, cookies, or session tokens.
- Loading of **untrusted resources** from external domains that can modify page behaviour.
- Users being tricked into performing unintended actions on the site.

Example Scenario:
An attacker injects **<script>** tags into a comment form or any input fields. Without CSP, the browser executes the script, potentially sending users' cookies to a malicious server.

How the risk is reduced using CSP

When CSP is implemented, the browser only allows content from **trusted sources** specified in the policy. Any scripts or resources not explicitly allowed are blocked.

- **Inline scripts** or malicious external scripts are prevented from executing.
- **Data theft** through XSS is reduced.
- **Control over third-party resources** ensures that only trusted or authorised domains are used.

Solution

CSP defines directives that specify allowed sources for different content types:

- default-src: Default sources for all content types.
- script-src: Sources allowed for JavaScript.
- style-src: Sources allowed for CSS.
- img-src : Sources allowed for images.
- connect-src: Sources allowed for AJAX/WebSocket requests.

Example Header Syntax:

Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.com; img-src 'self' data.

- 'self': Only allows content from the same origin.
- https://trusted.com: Allows scripts from a trusted external domain.
- data: Allows images encoded as inline data URIs.

Best Practices:

Recommended Values: Use 'self' for most resources, only include trusted external domains, avoid 'unsafe-inline'.

How Content Security Policy Works

CSP works by defining rules called “directives”. Each directive specifies what types of content are allowed and where they can come from. Think of it as a security whitelist for your web page.

#	Directive	Recommended Values	Unsafe / Dangerous Values
1	default-src	'self', https://trusted.com, https://cdn.example.com	*, 'unsafe-inline'
2	script-src	self', https://cdn.trusted.com,	*, 'unsafe-inline', 'unsafe-eval'
3	style-src	'self', https://fonts.googleapis.com	*, 'unsafe-inline'
4	img-src	'self', https://images.example.com,	*, untrusted domains
5	connect-src	'self', https://api.trusted.com	*, untrusted APIs
6	frame-ancestors	'self', https://partner.com	*, untrusted domains
7	media-src	'self', https://media.example.com	*, untrusted domains
8	font-src	'self', https://fonts.gstatic.com	*, untrusted domains
9	object-src	'none', 'self'	*, untrusted domains
10	form-action	'self', https://trusted-form.com	*, untrusted domains

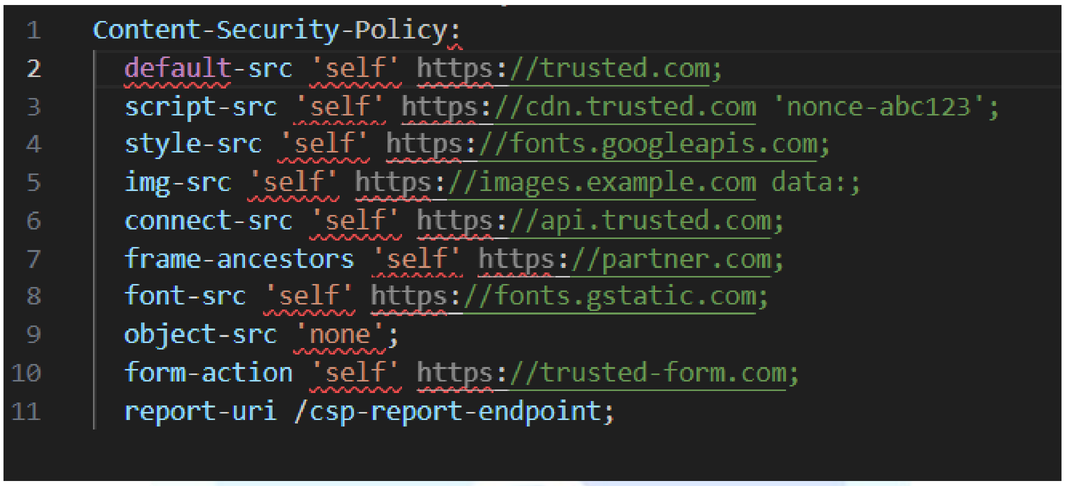
Content-Security-Policy (CSP) directives define which resources a browser is allowed to load and execute on a web page. Each directive controls a specific type of content, such as scripts, styles, images, or fonts. Using the recommended values ensures that only **trusted content sources** are loaded, reducing the risk of common client-side attacks.

For example, the **script-src** directive controls which JavaScript files can run. Using **'self'** or a trusted CDN prevents attackers from injecting malicious scripts through vulnerabilities like Cross-Site Scripting (XSS). If unsafe values like **'unsafe-inline' or *** are used, the browser may execute any script, potentially exposing sensitive data such as cookies, session tokens, or personal information. Similarly, **style-src** ensures that only safe CSS sources are applied, preventing attackers from injecting styles that could hide malicious content or trick users.

Other directives, like **img-src, connect-src,** and **font-src,** limit images, AJAX requests, and fonts to trusted sources. For instance, allowing images only from **'self'** and trusted domains stops attackers from embedding tracking pixels or malicious images that could compromise privacy. The **frame-ancestors** directive protects against clickjacking by allowing only specified websites to embed your content within **iframes**, reducing the risk of user manipulation.

While CSP is powerful, it has some limitations. Overly permissive policies, such as using * or 'unsafe-inline', can significantly weaken protection. Older browsers may ignore CSP, so additional security measures should still be in place. It is also important to note that CSP primarily defends against **client-side attacks**; server-side vulnerabilities like SQL Injection are not mitigated by CSP.

A Strong Content Security Policy Looks like:



```
Content-Security-Policy:
  default-src 'self' https://trusted.com;
  script-src 'self' https://cdn.trusted.com 'nonce-abc123';
  style-src 'self' https://fonts.googleapis.com;
  img-src 'self' https://images.example.com data:;
  connect-src 'self' https://api.trusted.com;
  frame-ancestors 'self' https://partner.com;
  font-src 'self' https://fonts.gstatic.com;
  object-src 'none';
  form-action 'self' https://trusted-form.com;
  report-uri /csp-report-endpoint;
```

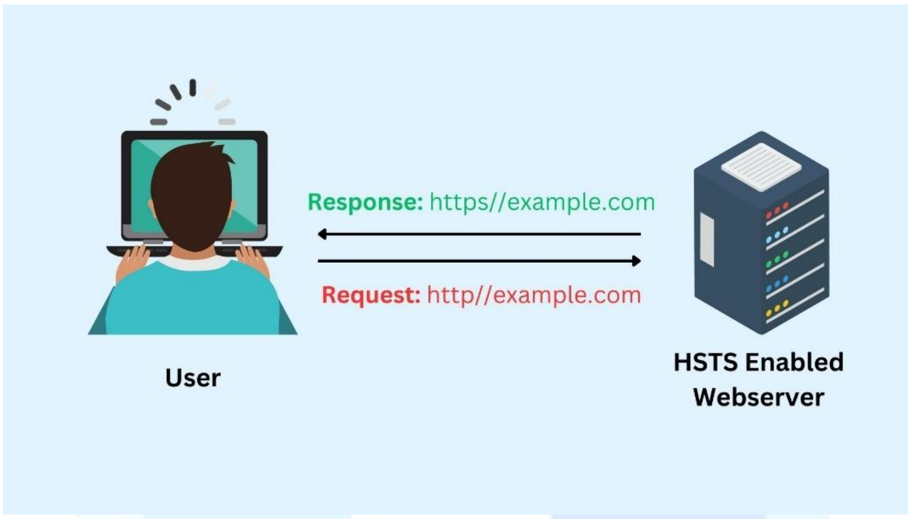
This example enforces strict content loading rules while allowing necessary external resources from trusted sources. It ensures that scripts, styles, images, and connections cannot be exploited by attackers, significantly reducing the risk of XSS, clickjacking, and data exfiltration.

In summary, implementing CSP correctly is essential for **modern web security**. Using safe directives, whitelisting trusted domains, and avoiding unsafe values ensures a robust defense against client-side attacks. Combining CSP with monitoring through reporting and other security headers strengthens your application's overall security posture.

Strict-Transport-Security (HSTS)

Strict-Transport-Security (HSTS) is an HTTP security header that instructs the browser to **only connect to a website using HTTPS**, never HTTP. Once a browser sees this header, it automatically converts all future requests to HTTPS, even if the user types http:// manually.

This ensures that all communication between the browser and server is **encrypted and secure**, protecting sensitive data like login credentials, session cookies, and personal information from being intercepted.



Why HSTS Header is important?

Without HSTS, users can accidentally access a website over unencrypted HTTP, which exposes data to attackers. This makes the site vulnerable to:

- **Man-in-the-Middle (MITM) Attacks:** Attackers intercept data sent over HTTP.
- **Session Hijacking:** Cookies or session tokens transmitted unencrypted can be stolen.

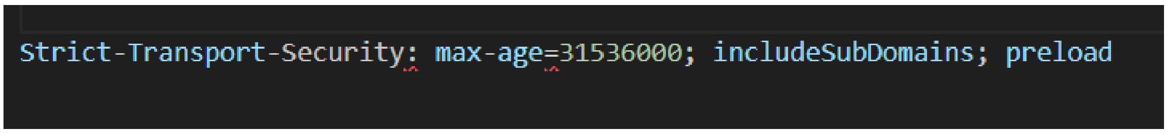
By enforcing HTTPS automatically, HSTS **prevents these attacks** and ensures users always connect securely.

How HSTS Header Works?

The server sends the HSTS header in its response to the browser. The header tells the browser:

- Only connect to this site using HTTPS for the next N seconds.
- Optionally, include all subdomains.
- Optionally, preload the site in the browser’s HSTS list so even the first visit is HTTPS.

Example Header Syntax:



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

- max-age=31536000 → Browser remembers to enforce HTTPS for 1 year.
- includeSubDomains → Applies HSTS to all subdomains.
- preload → Allows the domain to be included in browsers' HSTS preload lists.

This ensures that:

- Users never accidentally access the site over HTTP.
- All connections are encrypted, protecting sensitive data.
- Subdomains are also secure, reducing overall attack surface.

HSTS is a **critical security header** for modern websites. By enforcing HTTPS automatically, it prevents man-in-the-middle attacks, session hijacking, and SSL stripping. Implementing HSTS correctly, along with other security headers, provides a strong foundation for web application security.

X-Content-Type-Options

X-Content-Type-Options is an HTTP response header used by web servers to tell browsers **not to guess** (or "sniff") the MIME type of a file and to only trust the type declared in the Content-Type header.

A small header with a big job: it tells browsers **Do not guess the file type—only trust what the server declares.** This stops the browser from MIME type sniffing, which can lead to content-type confusion and script execution (XSS) from mislabelled files.

What is MIME type sniffing?

Browsers sometimes try to be helpful. If a response says Content-Type: text/plain but the bytes look like HTML/JS, older/lenient behaviour was to guess and render it as HTML/JS. That guessing is sniffing. If an attacker can make the browser misinterpret a file, they may be able to get their script executed.

Example:

- You upload notes.txt, but the content contains <script>…</script>.
- If the browser sniffs and decides "looks like HTML/JS", it may execute it—instant XSS.

no sniff disables that guessing for sensitive resource types.

What problems existed before this header?

- **Content-type confusion**: Malicious files mislabelled as harmless (e.g., JavaScript posing as text/plain or image/jpeg) could be executed.
- **Unsafe user uploads**: Attackers upload a file (e.g., avatar.jpg) that contains HTML/JS and then trick the app into including it as a script or stylesheet.
- **Missing/incorrect Content-Type**: If the server forgot to set Content-Type or used a generic one, browsers might sniff into a dangerous type and execute

Why is it important?

- **Blocks XSS via mislabelled resources**: Prevents the browser from executing JavaScript/CSS unless the MIME type matches.
- **Protects file upload/download features**: Critical wherever users can upload content or where files are proxied/served from storage (S3, CDN, etc.).

How it works?

When a response includes X-Content-Type-Options: no sniff, the browser **won't try to reinterpret** the resource's MIME type. For scripts and styles, if the Content-Type isn't a known executable style/script type, the browser **refuses to load/execute** it.

- **Scripts** must be served with a valid JavaScript MIME type (e.g., text/JavaScript, application/JavaScript).
- **Stylesheets** must be served with text/css.
- **If the type is wrong or missing**, the resource is blocked; the console shows a MIME-type mismatch error.

Attack Scenarios:

Scenario A — Malicious JS served as plain text

- /public/evil.txt contains: <script>fetch('/steal?c='+document.cookie)</script>
- Response headers: Content-Type: text/plain

Without no-sniff

<script src="/public/evil.txt"></script>

Some browsers may sniff and decide it's script-like, leading to execution (historically exploitable; behavior varied by browser/version).

With no-sniff

- Browser refuses to execute because the MIME type isn't a known JS type.
- Console: Refused to execute script from because its MIME type ('text/plain') is not executable.

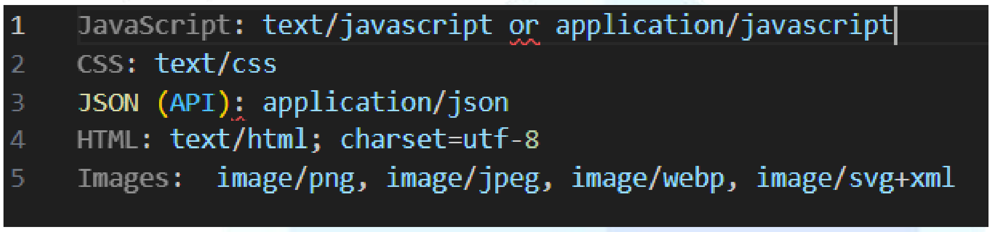
Scenario B — User upload polyglot (.jpg that is also HTML/JS)

- Attacker uploads avatar.jpg that is a **polyglot**: valid-looking JPEG bytes with embedded HTML/JS.
- App later includes it (accidentally) as a script: <script src="/uploads/avatar.jpg"></script>

Without no-sniff: Some user agents might misinterpret and execute.

With no-sniff: The resource is blocked because the declared/actual type isn't a recognized JavaScript type.

Correct MIME types:



```
JavaScript: text/javascript or application/javascript
CSS: text/css
JSON (API): application/json
HTML: text/html; charset=utf-8
Images:  image/png, image/jpeg, image/webp, image/svg+xml
```

Conclusion:

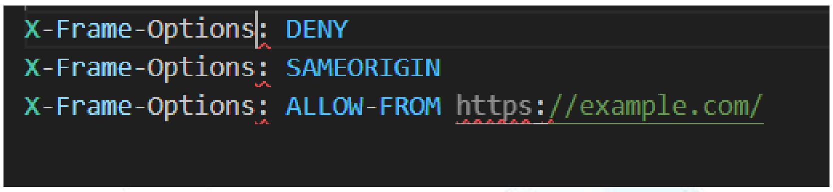
- Header: X-Content-Type-Options: nosniff
- Purpose: Stop MIME type sniffing; enforce strict type handling for scripts/styles.
- Protects against: Content-type confusion, some XSS vectors via mislabelled files.
- Requires: Correct Content-Type on all responses.
- Where to use: Everywhere—especially on sites with user uploads or any dynamic/static assets.

X-Frame-Options

X-Frame-Options is an HTTP response header that tells the browser **whether your website is allowed to be displayed inside a <frame>, <iframe>, <embed>, or <object> tag** on another site.

Its primary job is to **prevent clickjacking attacks**.

Typical values:



```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
X-Frame-Options: ALLOW-FROM https://example.com/
```

The problem before this header

Before X-Frame-Options existed:

- Any website could load your site in an invisible iframe.
- An attacker could overlay fake UI elements on top of the real buttons.
- The victim thinks they are clicking something harmless, but they are interacting with your site in the background — maybe transferring money, changing settings, or liking posts.
- This is called **clickjacking** (click hijacking).

Example:

1. Attacker builds a page that looks like a game.
2. They embed your banking site in an invisible <iframe> under the "Play" button.
3. When the victim clicks "Play," they click the hidden "Transfer Money" button.

Why it is important

- Protects sensitive actions from being tricked via hidden frames.
- Stops attackers from embedding your entire site for phishing or UI redressing.
- Acts as a **simple layer of defense** alongside Content Security Policy's frame-ancestors.

How it works

When a browser loads a page, it checks the X-Frame-Options header in the HTTP response:

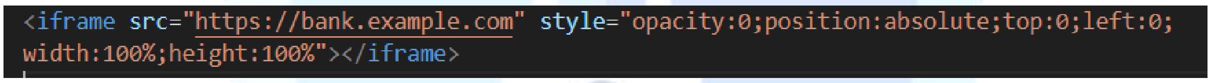
- **DENY** Never allow the page to be in a frame, even on the same site.
- **SAMEORIGIN** Allow the page to be framed only if the parent is from the same origin (same scheme, host, and port).
- **ALLOW-FROM URI** Allow the page to be framed only from a specific origin (note: not supported in all browsers).

How it solves the problem

- Prevents attackers from embedding your site in a malicious overlay.
- Stops the hidden "trick click" scenario.
- Forces the browser to respect your framing policy before rendering.

Example Scenarios

Without X-Frame-Options



```
<iframe src="https://bank.example.com" style="opacity:0;position:absolute;top:0;left:0;
width:100%;height:100%"></iframe>
```

- Browser loads your banking page invisibly.
- Attacker places fake buttons over it.
- User's clicks go to the banking page.

With X-Frame-Options: SAMEORIGIN

- Browser refuses to load https://bank.example.com in attacker's page.
- Console error: Refused to display 'https://bank.example.com' in a frame because it set 'X-Frame-Options' to 'SAMEORIGIN'.

Best Practices

- **If your page never needs framing** Use DENY.
- **If you frame your own pages** Use SAMEORIGIN.
- **Avoid ALLOW-FROM** unless necessary, and test carefully.
- Modern alternative: Use CSP header with frame-ancestors for more flexibility and wider future support

X-XSS-Protection

X-XSS-Protection is an **old security header** that was originally introduced by browsers (mostly Internet Explorer, Chrome, Safari) to enable or configure their **built-in Cross-Site Scripting (XSS) filter**.

It was designed to **detect reflected XSS attacks** and stop the page from loading malicious scripts.

Problem before this header

1. In the early 2000s, most browsers didn’t have any built-in defense against reflected XSS.
2. If a site was vulnerable to reflected XSS, the browser would happily execute the attacker’s JavaScript — allowing theft of cookies, keylogging, phishing overlays, etc.
3. If a website was vulnerable to reflected XSS (e.g.,? search=<script>…</script>), the browser would simply execute the script.
4. This allowed attackers to easily steal cookies, log keystrokes, or perform malicious actions.

Why it was important

Helped protect **users of vulnerable sites** that hadn’t yet fixed their XSS issues.

Understanding XSS

Cross-Site Scripting (XSS) is a vulnerability where attackers inject malicious scripts into web pages viewed by other users.

Types of XSS:

- Reflected XSS: Malicious script comes from the request and is reflected in the server’s response.
- Stored XSS: Malicious script is stored on the server (e.g., in a database) and served to many users.
- DOM-based XSS: Injection happens purely in the browser via client-side JavaScript.

The X-XSS-Protection header was created mainly to help mitigate reflected XSS.

How X-XSS-Protection works

When enabled, the browser's XSS filter:

1. Scans the rendered page for suspicious patterns (e.g., <script> tags injected from the URL).
2. If a pattern matches, the browser:
    - Sanitizes the suspicious code and renders the rest of the page (in 1 mode).
    - Blocks the entire page (in 1; mode=block mode).

Syntax & options

- X-XSS-Protection: 0 Disable the XSS filter.
- X-XSS-Protection: 1 Enable filter; sanitize suspicious scripts.
- X-XSS-Protection: 1; mode=block Enable filter; block page entirely if XSS detected.
- X-XSS-Protection: 1; report=<URL> (IE only) Send detection reports to the given URL.

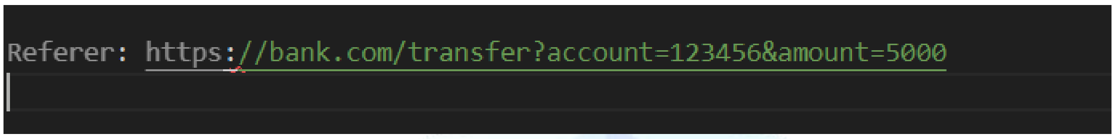
Header & Value	Meaning	Use / Behavior
X-XSS-Protection: 0	Disables the browser's XSS filter	No protection from browser-level XSS detection; not recommended unless the filter breaks functionality.
X-XSS-Protection: 1	Enables the XSS filter (sanitize mode)	Browser attempts to detect and neutralize reflected XSS attacks by removing suspicious script code but still renders the rest of the page.
X-XSS-Protection: 1; mode=block	Enables the XSS filter (block mode)	If XSS is detected, the browser blocks the entire page from loading, preventing any malicious script execution.
X-XSS-Protection: 1; report=<URL>	Enables filter and sends violation report (IE only)	Same as 1 but sends details of the detected XSS attempt to the specified <URL> for logging or analysis (only works in Internet Explorer).

Referrer-Policy

The Referrer-Policy HTTP header that controls how much of the **referrer information** is sent in the Referer header when a user navigates from one page to another, clicks a link, or loads a resource.

The Referer header (yes, it's spelled incorrectly due to a historical typo) contains the URL of the page that initiated the request. Without restrictions, browsers may send the **entire URL**, including sensitive data, to other sites.

Example of a Referer header:



```
Referer: https://bank.com/transfer?account=123456&amount=5000
```

Why This Header Is Needed

Before Referrer-Policy

1. Browsers would always send the full referrer URL, regardless of whether it contained sensitive information.
2. If the current URL contained query parameters with personal or confidential details, those details would be leaked to any third-party site or resource.

Example Risk:

User visits: https://bank.com/transfer?account=123456&amount=5000

User clicks a link to: https://example.com

Browser sends: **Referer: https://bank.com/transfer?account=123456&amount=5000**

This leaks account numbers and transaction details to an external site

How Referrer-Policy Solves the Problem

Referrer-Policy gives developers **fine-grained control** over what gets sent in the Referer header:

1. Send **no referrer** (maximum privacy).
2. Send **only the origin** (e.g., https://bank.com).
3. Send the **full URL** only when safe.
4. Prevent referrer leakage when downgrading from **HTTPS to HTTP**.

Available Policy Values

#	Value	What it Sends	Use case
1	no-referrer	Sends no referrer at all	Maximum privacy; prevents all leakage but can break analytics and certain integrations.
2	no-referrer-when-downgrade (old default)	Sends full URL only for same security level requests; nothing when HTTPS → HTTP	Basic protection against downgrade leaks.
3	origin	Sends only the origin (scheme, host, port)	Share general site info without revealing paths/queries.
4	origin-when-cross-origin	Sends full URL for same-origin requests, origin for cross-origin requests	Balances analytics with privacy.
5	same-origin	Sends full URL for same-origin requests; nothing for cross-origin	Strong privacy against external sites.
6	strict-origin	Sends only the origin if same security level; nothing if HTTPS → HTTP	Secure origin sharing only.
7	strict-origin-when-cross-origin (modern default)	Sends full URL for same-origin, only origin for cross-origin, and nothing on HTTPS → HTTP	Recommended for most sites today.
8	unsafe-url	Sends full URL always	Not recommended; can leak sensitive info.

Example in Action

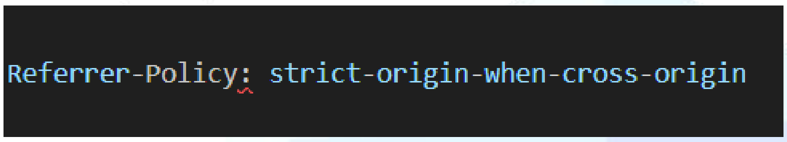
Policy: Referrer-Policy: strict-origin-when-cross-origin

Scenario:

- **Current page:** https://secure.com/dashboard?user=john
- **Click link to another site:** https://blog.com/article
- **Browser sends:** Referer: https://secure.com

Sensitive details (?user=john) are removed.

Best Practice Recommendation (2025)



```
Referrer-Policy: strict-origin-when-cross-origin
```

Reason:

- Protects sensitive query parameters from leaking to other domains.
- Maintains useful analytics for internal navigation.
- Prevents downgrade leaks to insecure HTTP.

Permissions-Policy (formerly Feature-Policy)

Browser has many built-in features (camera, mic, location, sensors, clipboard, etc.).

When you open a website, the browser is not just showing text and pictures — it can also connect to hardware, sensors, and sensitive data on your device.

These special abilities are called "powerful features" because they can affect privacy, security, or device performance.

Examples of Browser Powerful Features:

1. **Camera** – Lets a website take pictures or record video using your webcam.
2. **Microphone** – Lets a website record your voice.
3. **Geolocation** – Lets a website know your exact location (like GPS on your phone).
4. **Fullscreen** – Lets a website cover your entire screen (can be abused for fake login screens).
5. **USB / Bluetooth / NFC** – Lets a website connect to hardware devices like printers, sensors, or controllers.
6. **Payment Request API** – Lets a website request payment through stored credit card info.
7. **Motion Sensors** – Lets a website read your phone's accelerometer or gyroscope (can detect movement, walking patterns).
8. **Clipboard Access** – Lets a website read or write to your copy-paste memory.

The **Permissions-Policy** HTTP header lets you control **what powerful features of the browser a web page can use**, and **who** (your site or other embedded content) can use them. Think of it as **a set of rules you give to the browser** when it loads your page.

Why it exists

Before this header:

- Any third-party content you included (ads, analytics scripts, widgets, videos) could try to access **features like your camera, microphone, location, or sensors**.
- Even if the browser asked for permission, **users often didn't notice which part of the page was asking** and clicked "Allow" by mistake.
- Attackers could trick people into giving access, which meant **privacy leaks** and **security risks**.

Why it's important

Permissions-Policy:

- Prevents untrusted embedded content from requesting dangerous permissions.
- Gives site owners fine-grained control over browser features.
- Helps comply with privacy regulations by ensuring only the intended parts of your site can access sensitive data.
- Reduces the attack surface for malicious scripts.

How it works

When your server sends a web page, it can include this HTTP header:

Permissions-Policy: <feature>=<allow-list>, <feature>=<allow-list>

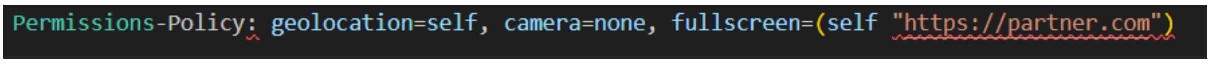
#	Feature	What it controls	Example Risk if Not Restricted
1	geolocation	Access to the device's location	A third-party ad could track user movements
2	camera	Access to the webcam	Malicious iframe could spy on users
3	microphone	Access to the microphone	Could record conversations
4	Fullscreen	Ability to make the page fullscreen	Phishing site could fake a browser UI
5	payment	Web Payment API	Fake checkout iframe could steal payment info
6	usb	USB device access	Malicious code could read from connected devices
7	accelerometer, gyroscope	Motion sensors	Could be used for tracking or detecting activity

Example

Your site needs geolocation only for your own pages.

You want to block camera access entirely.

You allow Fullscreen for your site and a trusted partner site.



```
Permissions-Policy: geolocation=self, camera=none, fullscreen=(self "https://partner.com")
```

Summary

- **Without it:** Any embedded content could request sensitive browser features.
- **With it:** You decide exactly which features are available, and to whom.
- **Benefit:** Stronger privacy, better security, more control over third-party content

Cross-Origin Resource Sharing (CORS) Security Headers

What is an Origin?

An Origin is just three things, and only these three:

1. **Scheme**: http, https etc
2. **Host (domain or IP):** example.com, sub.example.com, 127.0.0.1
3. **Port**: 443, 80, 8080 etc

Origin = scheme + host + port

Paths(/page), queries(?id=1), and fragments(#top) do not affect the origin.

Why ports matter

Browsers assume a **default port** when it's not written:

- http → **80**
- https → **443**

1. https://example.com and https://example.com:443 → **same origin** (443 is the default for https).
2. https://example.com and https://example.com:8443 → **different origins** (port differs).

Why Scheme matters

Even with the same host and port:

http://example.com vs **https://example.com** → **different origins** (scheme differs)

Why host matters

Subdomains are different hosts:

- **https://example.com** vs **https://sub.example.com** → **different origins** (host differs).
- **https://example.com** vs **https://www.example.com** → **different origins**.

Hostnames vs IPs are different too:

- **http://localhost:3000** vs **http://127.0.0.1:3000** → **different origins** (host differs).

Ask yourself: **Do these three all matches?**

- Same **scheme**?
- Same **host**?
- Same **port** (including implied defaults)?

If **anyone** is different → it's a **different origin**.

Same-origin vs. different-origin examples

Same origin (all three match):

- **https://example.com/one and https://example.com/two**
- **https://example.com and https://example.com:443 (443 is default for https)**

Different origin (one thing differs):

- **Scheme differs: http://example.com vs https://example.com**
- **Host differs: https://example.com vs https://sub.example.com**
- **Port differs: https://example.com vs https://example.com:8443**

What is SOP?

The **Same-Origin Policy** is a **browser security rule** that stops one website from freely reading data from another website **unless they have the same origin** (same **scheme + host + port**).

It exists to protect users — otherwise, malicious sites could load your online banking page and read your account balance without you knowing.

The Core Rule

A web page can request and interact with resources only from the same origin, unless explicitly allowed otherwise

"Interact" here means:

- Accessing data with JavaScript (AJAX / Fetch API)
- Reading the contents of a loaded iframe
- Inspecting responses from APIs

Cross-Origin Resource Sharing (CORS)

Now that you know SOP blocks cross-origin requests from being read, here's where CORS comes in.

Cross-Origin Resource Sharing is a mechanism that lets a server say:

"It's okay for this other origin to access my data."

How CORS works

1. Browser sends a request to another origin.
2. Server responds with special headers (like Access-Control-Allow-Origin) telling the browser who's allowed to read the response.
3. If the origin is allowed → browser gives the page the data.
   If not → browser blocks the response in JavaScript.

CORS Requests: Simple vs Preflight

Cross-Origin Resource Sharing (CORS) is a security feature implemented by browsers to control how web pages can request resources from a different domain (origin).
When a browser sends a cross-origin request, it determines how to handle it based on the request's complexity

There are **two main types of CORS requests**:

1. **Simple Requests**
2. **Preflight Requests**

1. Simple Requests

Definition

A **simple request** is a cross-origin request that **meets specific conditions** defined by the CORS specification.
For these requests:

- The browser sends the request **directly** to the server.
- There is **no preliminary OPTIONS request**.
- The browser checks the server's **CORS response headers** to decide whether to allow JavaScript access to the response.

Criteria for a Simple Request

A request qualifies as "simple" if **all** the following conditions are met:

Allowed HTTP Methods

- GET
- POST
- HEAD

Allowed Content-Types

If the request has a body, the Content-Type must be one of:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

Allowed Request Headers

Only **simple headers** are allowed:

- Accept
- Accept-Language
- Content-Language
- Content-Type (only with allowed values above)

No custom headers (e.g., Authorization, X-Custom-Header) are permitted in a simple request

2.Preflight Requests

A preflight request is an initial request made by the browser using the OPTIONS HTTP method before the actual request.

This step is required for requests that **do not meet** the criteria for a simple request.

The preflight request asks the server for permission, ensuring the server explicitly approves:

- The HTTP method
- Any custom request headers

When Preflight is Triggered

A preflight request occurs if any of the following are true:

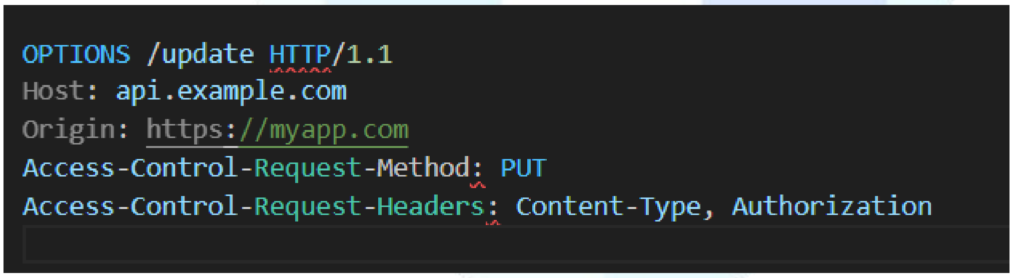
1. HTTP Method is not GET, POST, or HEAD (e.g., PUT, DELETE, PATCH).
2. Content-Type is not one of:

    - application/x-www-form-URL encoded
    - multipart/form-data
    - text/plain

3. The request includes custom headers (e.g., Authorization, X-Custom-Header).

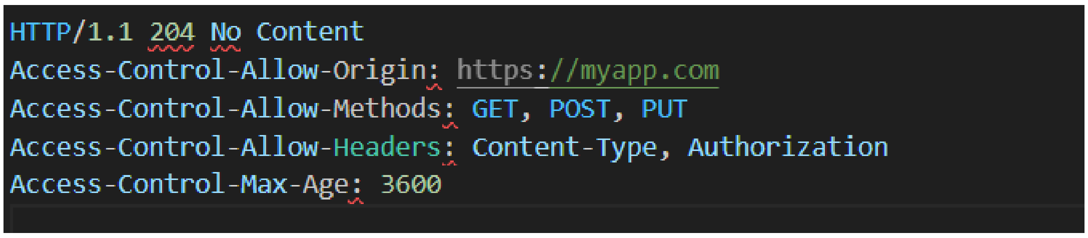
Preflight Request Flow

Step 1: Browser sends OPTIONS request



```
OPTIONS /update HTTP/1.1
Host: api.example.com
Origin: https://myapp.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Content-Type, Authorization
```

Step 2: Server responds



```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://myapp.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 3600
```

Step 3: Browser decision

1. If the method, headers, and origin match the server's policy → Browser sends the actual request.
2. Otherwise → Request is blocked.

Common CORS Headers

When implementing Cross-Origin Resource Sharing (CORS), the server uses specific HTTP headers to define which cross-origin requests are allowed.

1)Access-Control-Allow-Origin

Specifies which origins are allowed to access the resource.
It's the most important CORS header because it tells the browser whether the requesting domain is permitted.

Details

- **The value can be:**
    - **A specific origin (e.g., https://myapp.com)**
    - **A wildcard * (allowing all origins, but risky if sensitive data is involved)**
- **If credentials (cookies, authorization headers) are included in the request, you cannot use *. Instead, you must specify an exact origin.**

Example:

Access-Control-Allow-Origin: https://myapp.com

If multiple origins need access, the server must dynamically return the correct origin value depending on the request.

2) Access-Control-Allow-Methods

Specifies which HTTP methods are allowed when accessing the resource.
This is especially important in preflight requests where the browser asks permission before sending the actual request**.**

Details

- **Commonly allowed methods include GET, POST, PUT, DELETE, PATCH.**
- **If a request uses a method not listed here, the browser will block it.**

Example

Access-Control-Allow-Methods: GET, POST, PUT, DELETE

This means cross-origin requests can use GET, POST, PUT, and DELETE.

3) Access-Control-Allow-Headers

Specifies which custom headers can be used in the actual request.
This is important when the request includes headers other than the simple ones (Accept, Content-Type with limited values, Accept-Language, etc.).

Details

- Needed only when the request includes **custom headers** like:
    - Authorization
    - X-Custom-Header
    - Content-Type: application/json (since it’s not a "simple" value)

Example

Access-Control-Allow-Headers: Content-Type, Authorization, X-Custom-Header

If the browser sends a preflight request asking whether it can use these headers, the server must explicitly list them here.

4) Access-Control-Allow-Credentials

Indicates whether the response can be shared when the request includes credentials such as:

- Cookies
- HTTP Authentication
- Client-side SSL certificates

Details

- Value can only be true (never false).
- Must not be used with a wildcard Access-Control-Allow-Origin: *.
- Instead, the server must specify the exact trusted origin.

Example: Access-Control-Allow-Credentials: true

When set, it allows cross-origin requests with credentials (like a session cookie).

5) Access-Control-Max-Age

Specifies how long (in seconds) the results of a preflight request can be cached by the browser.
This helps reduce unnecessary preflight requests and improves performance.

Details

- Value is an integer (seconds).
- Common values:
    - 600 (10 minutes)
    - 3600 (1 hour)
    - 86400 (24 hours)

Example:

Access-Control-Max-Age: 3600

This tells the browser it can reuse the preflight response for the next **1 hour** without sending another OPTIONS request.

#	Header	Purpose	Example
1	Access-Control-Allow-Origin	Defines which origins can access the resource	Access-Control-Allow-Origin: https://myapp.com
2	Access-Control-Allow-Methods	Lists allowed HTTP methods	Access-Control-Allow-Methods: GET, POST, PUT
3	Access-Control-Allow-Headers	Lists allowed custom headers	Access-Control-Allow-Headers: Content-Type, Authorization
4	Access-Control-Allow-Credentials	Allows credentials (cookies, auth)	Access-Control-Allow-Credentials: true
5	Access-Control-Max-Age	Caches preflight response for a duration	Access-Control-Max-Age: 3600

Cache-Control (Security Implications for Sensitive Data)

What is Cache-Control?

The Cache-Control HTTP header is used to tell browsers and intermediate caches (like proxies or CDNs) how they should store and reuse responses.

- Caching improves **performance** (faster page loads, less server load).
- But if used incorrectly, it can cause **security risks**, especially for **sensitive data** (like personal info, financial transactions, or authentication tokens).

Why is Cache-Control Important for Security?

Without proper cache control:

- Sensitive data (like bank statements, health records, or session details) might get stored in shared caches (e.g., ISP proxy, corporate cache, or CDN).
- Later, another user could accidentally (or maliciously) see this data.

By setting secure Cache-Control rules, websites can **prevent sensitive responses from being cached improperly**.

Common Cache-Control Directives

Directive	Purpose	Example	Security Implication
no-store	Tells browsers and caches not to store the response at all.	Cache-Control: no-store	Best for sensitive data (e.g., banking pages, login responses). Prevents storage anywhere.
no-cache	Forces revalidation with the server before using a cached copy. (Content *can* be stored, but must be checked before reuse.)	Cache-Control: no-cache	Prevents serving outdated/stale sensitive data.
private	Allows caching only on the user's browser, not shared caches (like proxies/CDNs).	Cache-Control: private	Safe for personal dashboards — keeps data on user's device, not shared servers.
public	Allows caching by any cache (browser, proxy, CDN).	Cache-Control: public	Risky for sensitive data — never use for private info. Useful only for static assets (CSS, JS).
max-age=seconds	Defines how long a response can be cached before it must be revalidated.	Cache-Control: max-age=3600	Helps control freshness. For sensitive data, use short or zero values.
must-revalidate	Requires caches to check with the server once the response is stale.	Cache-Control: must-revalidate	Prevents stale data from being reused without confirmation.

Cross-Origin-Resource-Policy (CORP)

The **Cross-Origin-Resource-Policy (CORP)** HTTP response header is a **security header** that helps control who can load resources (like images, scripts, fonts, or iframes) from your server.

- Normally, resources on the web (like images or scripts) can be embedded in other sites.
- But sometimes you don't want other websites to embed your resources because it could lead to **data leaks or security risks** (e.g., leaking private user data through an image request).

CORP allows the server to **decide whether other origins (websites) can load its resources or not**.

Why is CORP Important?

Without CORP:

- Any site could load your resources silently.
- If your resources contain **user-specific or sensitive data** (e.g., private profile pictures, user-only content), another website might trick the browser into fetching them and exposing information.

With CORP:

- You **restrict resource sharing** and protect against **cross-origin data leaks**.

Available CORP Values

Value	Meaning	Use Case
same-origin	Only the same origin (same scheme, host, and port) can load this resource.	Best for sensitive data like private images, JSON APIs, or user content.
same-site	Allows loading by documents from the same site (same domain, including subdomains) but not from totally different origins.	Useful when you want subdomains (e.g., app.example.com and cdn.example.com) to share resources, but not outside sites.
cross-origin	Any site can load the resource.	Safe only for public assets like open-source fonts, logos, or files intended for sharing.

CORS = Who can read my data

CORP = Who can even load my resource

Cross-Origin-Embedder-Policy (COEP)

What is COEP?

The **Cross-Origin-Embedder-Policy (COEP)** header is a security header that tells the browser to **only load resources that explicitly grant permission to be embedded**.

This prevents your website from unknowingly embedding malicious or unauthorized resources (like scripts, images, or iframes) from other origins.

It is especially important for enabling powerful browser features like **SharedArrayBuffer** and ensuring your website runs in a **secure isolated context**

Why is COEP Important?

Before COEP, a malicious third-party resource (like an iframe or script from another origin) could be embedded into your site without explicit consent. This could lead to:

- **Data leaks** via side-channel attacks.
- **Resource hijacking**, where an attacker's resource runs inside your site.
- Inability to use advanced APIs like **SharedArrayBuffer**, which require strong isolation.

COEP ensures that **every cross-origin resource explicitly opts in** to being embedded.

How Does COEP Work?

- When your page sets the **Cross-Origin-Embedder-Policy** header, the browser will check each cross-origin resource (e.g., scripts, images, iframes).
- If the resource **does not explicitly allow itself** to be embedded (using headers like Cross-Origin-Resource-Policy (CORP) or Cross-Origin-Resource-Share (CORS)), the browser will **block it**.

This creates a **fully isolated environment** for your site.

Value	Meaning	Example
unsafe-none (default)	No restrictions. Any cross-origin resource can be embedded.	Cross-Origin-Embedder-Policy: unsafe-none
require-corp	Only cross-origin resources that grant permission via CORS or CORP can be embedded.	Cross-Origin-Embedder-Policy: require-corp

In practice, you should always use **require-corp** for security.

Summary:

- COEP = “Only load resources that explicitly say I can.”
- Works together with **CORS** and **CORP** to enforce strict isolation.
- Best practice: Always set Cross-Origin-Embedder-Policy: require-corp.

6. Conclusion

In today's digital landscape, where web applications are constantly exposed to a wide variety of security threats, **HTTP Security Headers** play a critical role in building a strong first line of defense. While they may seem like small configuration settings at first glance, these headers can significantly reduce the risk of attacks such as cross-site scripting (XSS), clickjacking, protocol downgrade, data leaks, and unauthorized access to powerful browser features. By explicitly instructing browsers on how to handle content, enforce restrictions, and manage cross-origin interactions, security headers serve as a protective shield that complements other security mechanisms.

A key takeaway from this documentation is that **security is never about a single control**—it is about layering multiple defenses. Firewalls, encryption, input validation, authentication systems, and intrusion detection are all important, but without properly configured headers, web browsers remain open to misuse. For example, Content-Security-Policy (CSP) helps mitigate XSS, but without Strict-Transport-Security (HSTS), users may still fall victim to protocol downgrade attacks. Similarly, X-Frame-Options and Referrer-Policy may seem like simple headers, yet they provide critical protection against clickjacking and unnecessary information exposure. When used together, these headers form a **defense-in-depth approach** that raises the cost of attack for adversaries.

Another crucial aspect is that security headers are not only about protecting the website but also about **protecting the end-users**. For instance, Referrer-Policy prevents sensitive URLs from leaking, Permissions-Policy restricts access to hardware features like camera or microphone, and Cache-Control ensures that sensitive data does not remain stored in shared or public caches. In other words, these headers safeguard both **application integrity** and **user privacy**.

From a best practices perspective, developers and security teams should:

- Always enforce **HTTPS** with HSTS.
- Define a strict and well-tested **Content-Security-Policy (CSP)**.
- Prevent MIME type sniffing with **X-Content-Type-Options**.
- Apply **X-Frame-Options** or equivalent CSP directives against clickjacking.
- Use **Referrer-Policy** to control information sharing.
- Apply **Cache-Control** to sensitive endpoints like login pages or financial data.
- Use **CORS headers** carefully, never allowing * for credentials.
- Adopt **CORP, COEP, and COOP** for modern isolation-based security.

To conclude, **security headers are small yet powerful tools** that bridge the gap between web servers and browsers. They empower developers to define strict security rules, they guide browsers in enforcing those rules, and they add an essential layer of resilience against ever-evolving threats. By understanding, implementing, and continuously updating these headers, organizations can ensure that their applications remain robust, trustworthy, and secure for all users.