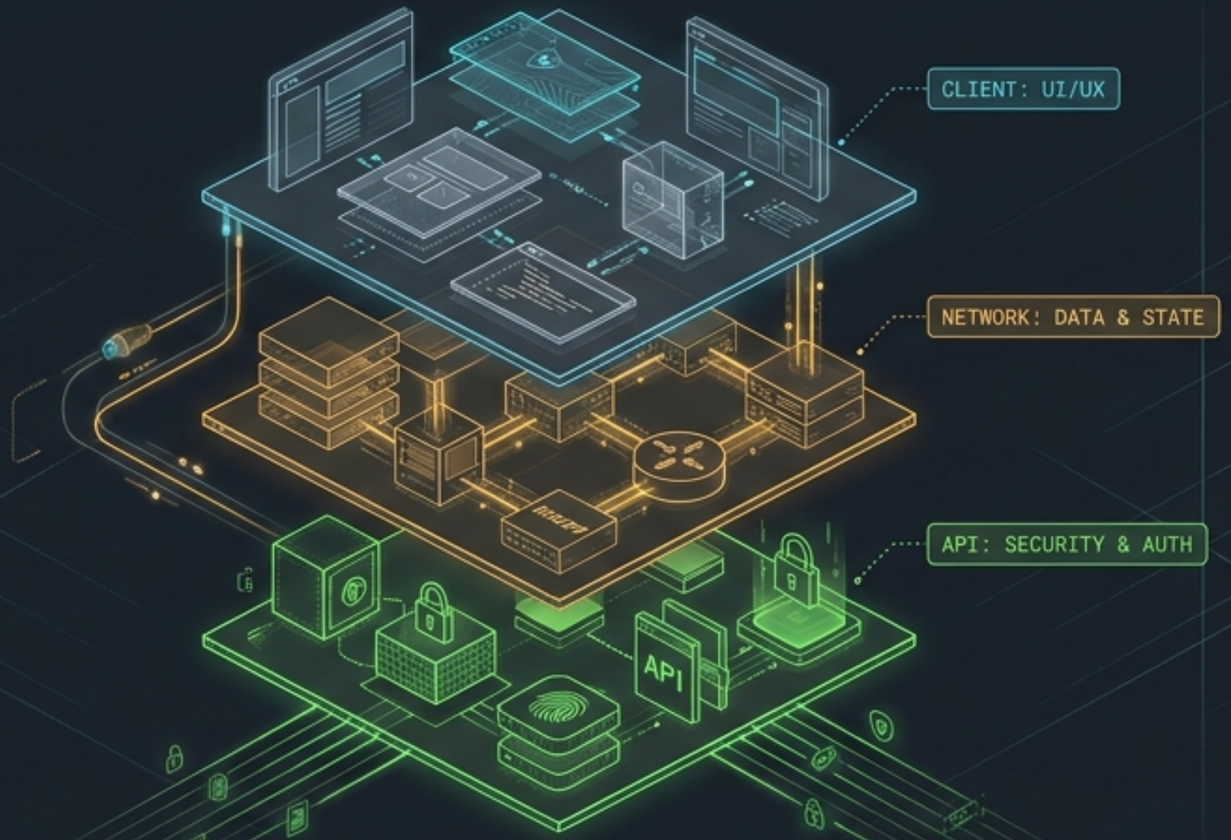
Architecting Modern Web Security

A Layered Defense Playbook for the Development Lifecycle

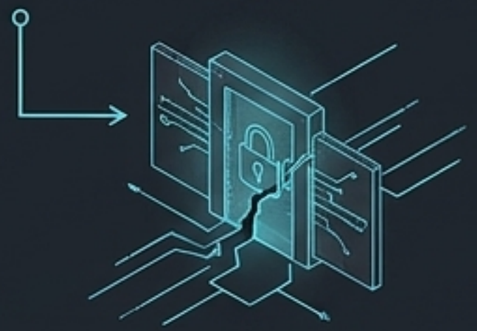


The Structural Fault Lines of the Web

OWASP TOP10

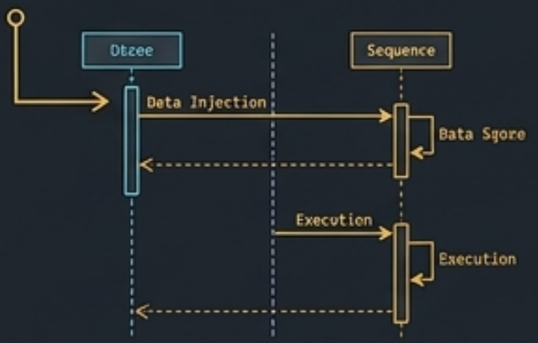
Broken Access & Auth

Identity systems failing to verify the user or their permissions (OWASP A01 & A07). The root of session hijacking.



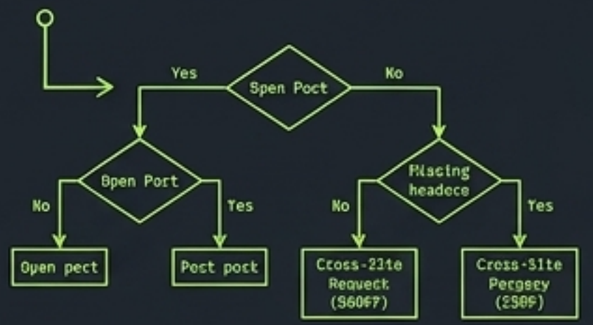
Injection

Hostile data executed as code by the browser or server (OWASP A03).
The core mechanism of Cross-Site Scripting (XSS).



Security Misconfiguration

Missing HTTP headers, default settings, and exposed state (OWASP A05).
The enabler of Cross-Site Request Forgery (CSRF).

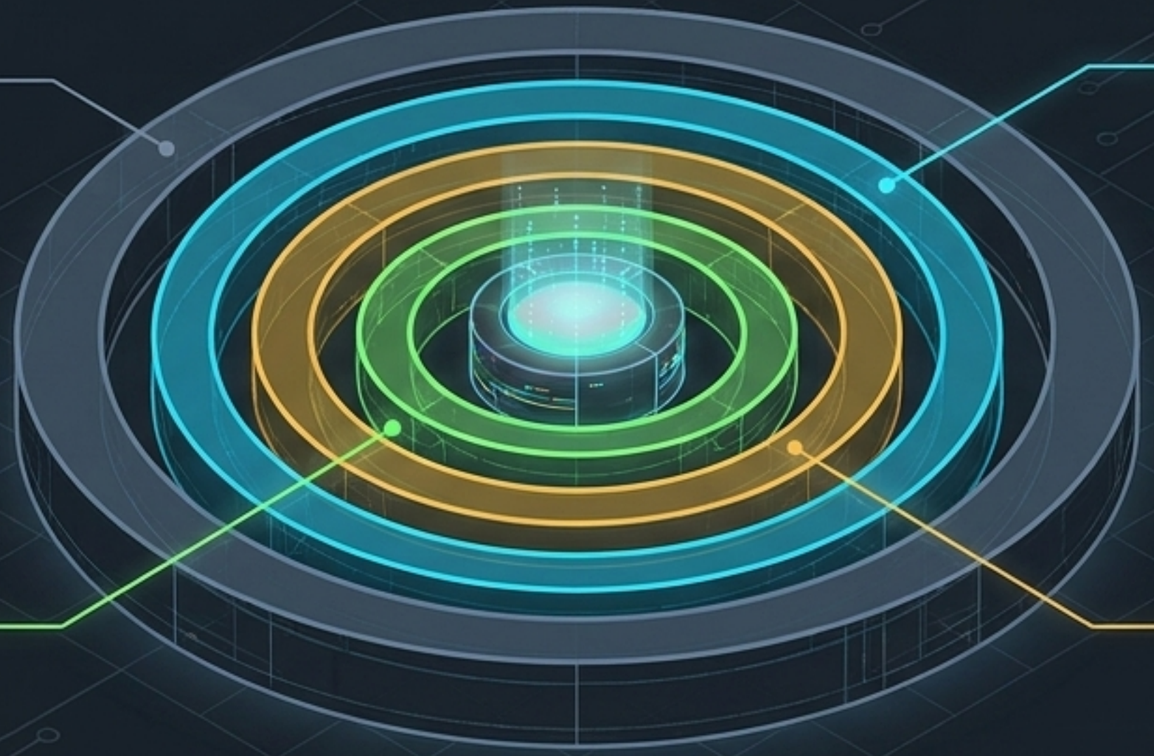


Takeaway: Security is no longer about patching holes; it requires designing a fortified architecture from the ground up.

The Defense in Depth Architecture

The Foundation: Software Development Lifecycle (SDLC) & Threat Modeling.

The Client: Securing the Frontend against XSS via Content Security Policy (CSP)



The Identity: Securing the API against Auth Bypass via JSON Web Tokens (JWT).

The State: Securing the Network against CSRF via Tokens, Fetch Metadata, and SameSite.

Eliminate single points of failure. If one layer is breached, the next layer intercepts the attack.

The Secure Development Lifecycle (SDLC)

Requirements: Defining ASVS baselines.

Design: Threat modeling the architecture.

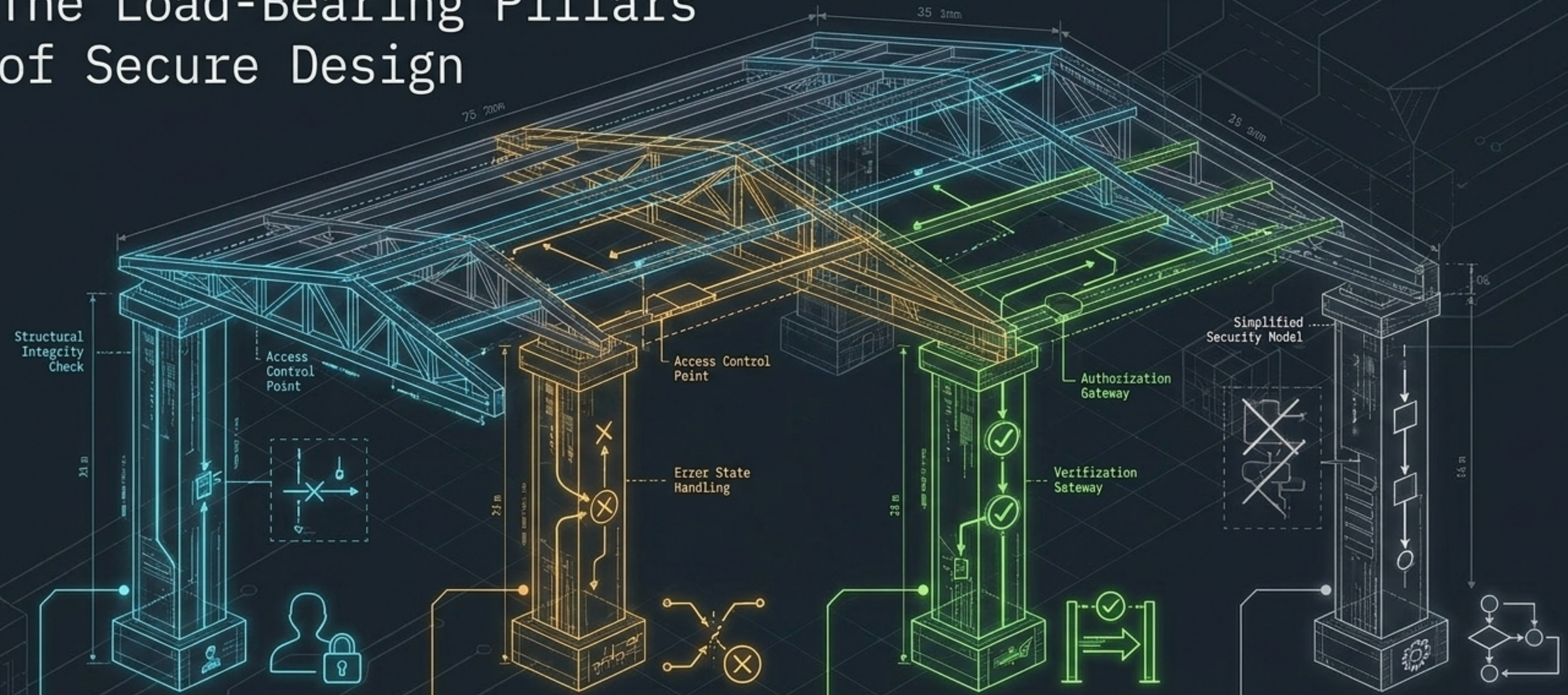
Implementation: Enforcing Proactive Controls and Secure Libraries.

Verification: Automated pipelines (SAST/DAST) and gap analysis.

Operations: DevSecOps, Continuous Monitoring, and WAF rules.

Security by Design: Treat security requirements exactly like functional requirements. Build it in, don't bolt it on.

The Load-Bearing Pillars of Secure Design



Least Privilege:
A process or user operates with the absolute minimum access rights necessary.

Fail Safe:
In the event of an error or exception, the system defaults to denying access.

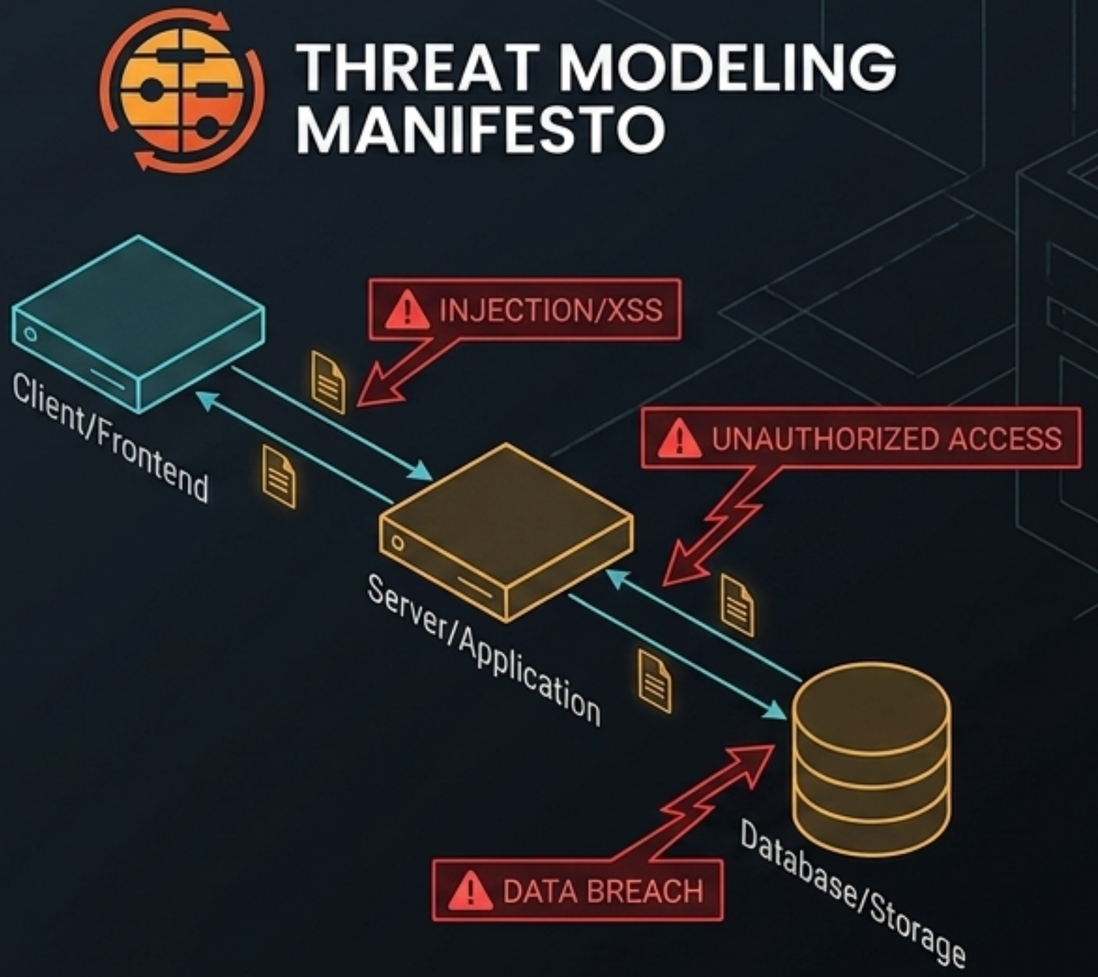
Complete Mediation:
Every single request for an object is checked for authorization--authority is never cached.

Economy of Mechanism:
Keep security controls simple. Complexity bzeeds hidden vulnerabilities.

Threat Modeling: Finding the Fault Lines

The Four Question Framework

1. What are we working on?
   (Define architecture & data flows)
2. What can go wrong?
   (Apply STRIDE or LINDDUN to find vulnerabilities)
3. What are we going to do about it?
   (Transfer, Accept, Mitigate, or Eliminate)
4. Did we do a good enough job?
   (Validate mitigations and metrics)

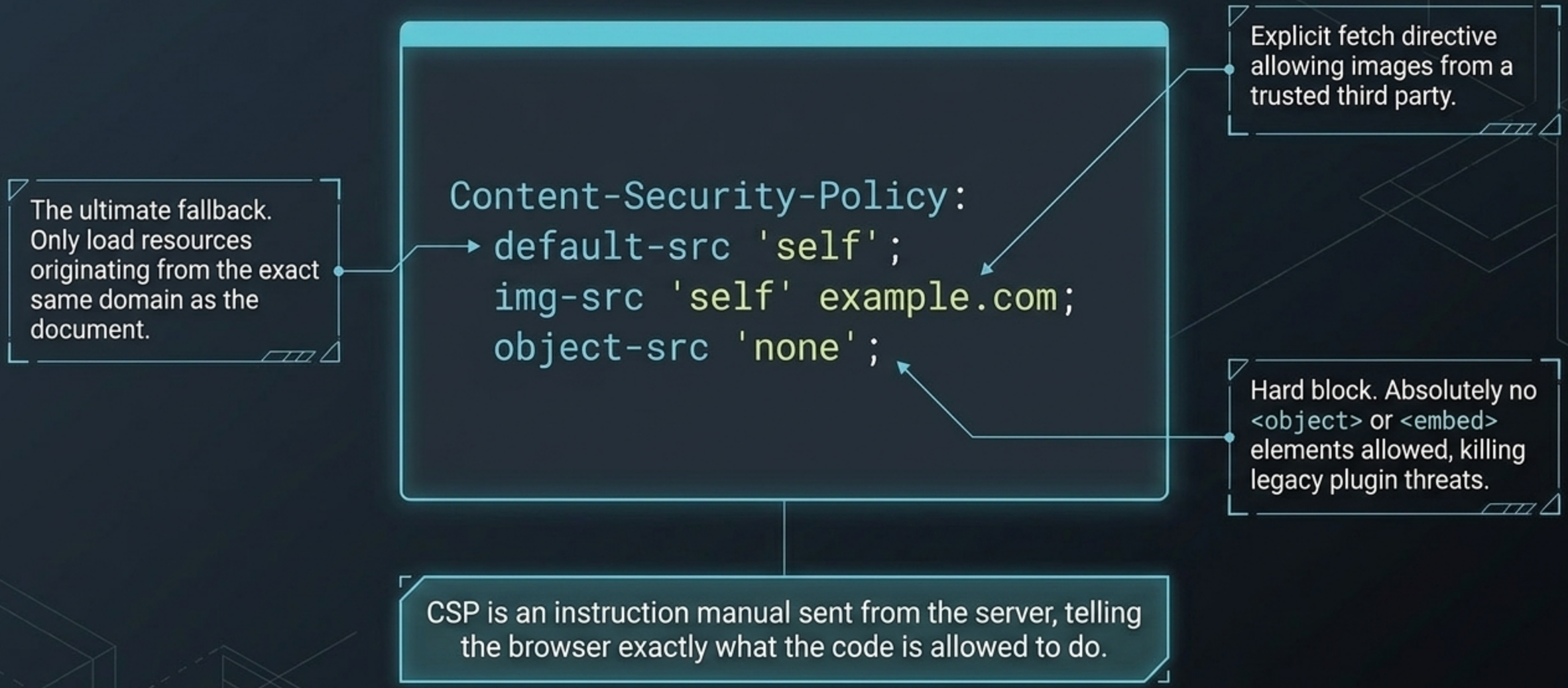


Layer 1: Securing the Client

Architectural Briefing

The browser is a hostile environment. If an attacker injects malicious scripts (XSS), they operate with the full permissions of the user. Sanitizing input is required, but it will eventually fail. We need a structural defense-in-depth policy to govern resource loading.

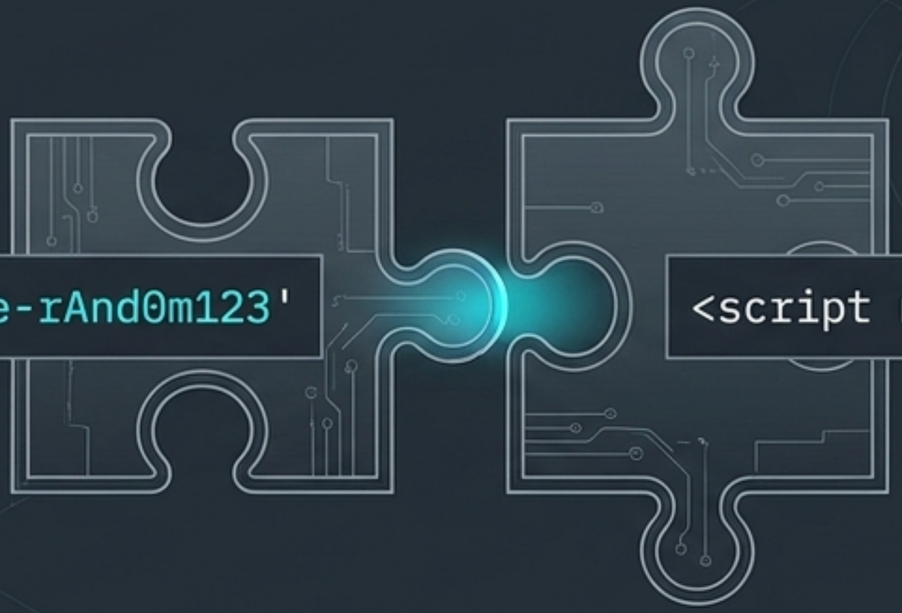
The Anatomy of a Content Security Policy



Defeating Inline Injection: The Nonce Match

Server HTTP Header

```
script-src 'nonce-rAnd0m123'
```



HTML DOM

```
<script nonce="rAnd0m123">
```

Step 1 (Server Generation)

The server generates a random, unpredictable Base64 string for every single HTTP response.

Step 2 (The Delivery)

Server sets the strict CSP header with the generated nonce.

Step 3 (The HTML)

Server injects the matching nonce attribute directly into trusted `<script>` tags.

The Mechanism: The browser compares the two. If an attacker injects `<script>alert(1)</script>`, it lacks the unguessable nonce and is instantly blocked.

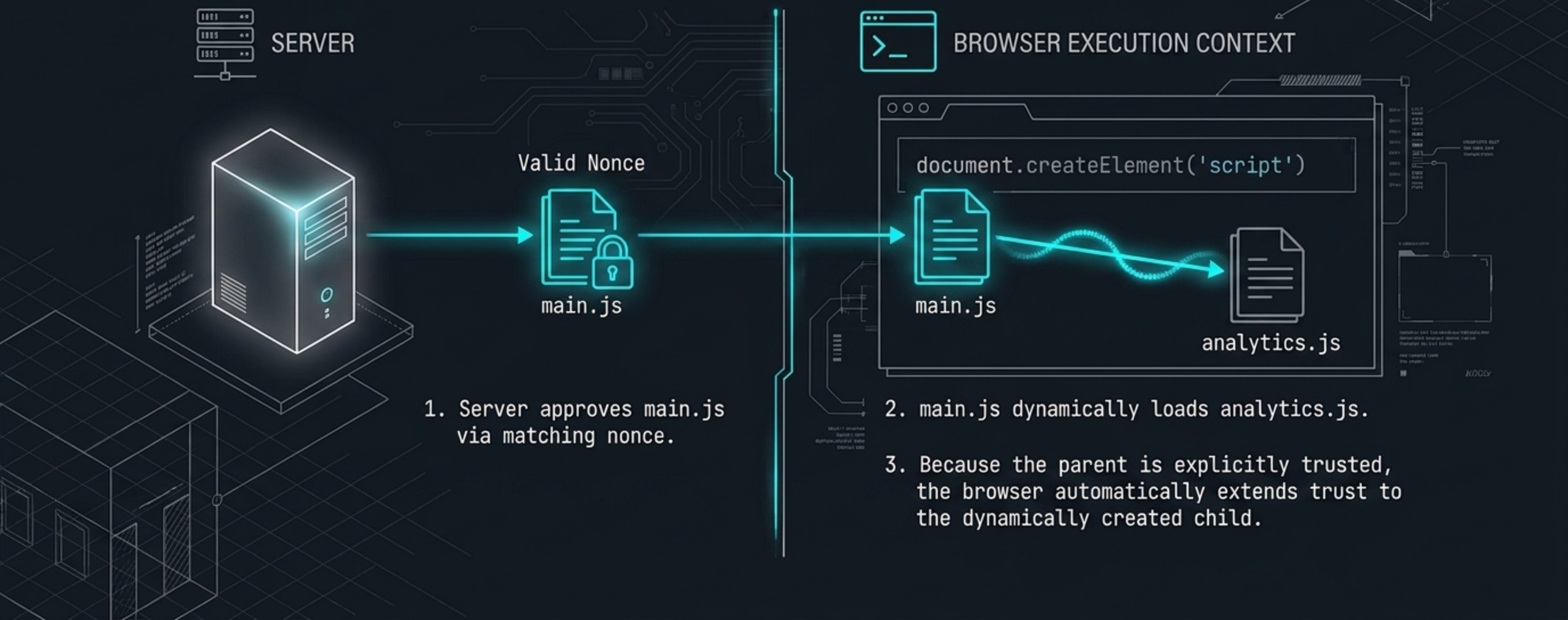
Diagnostic: Allowlist vs. Strict CSP

	Allowlist CSP	Strict CSP
Mechanism	Relies on trusting entire domains (e.g., `*.google.com`).	Relies on cryptographic nonces or content hashes.
Maintenance Bloat	Policies grow to hundreds of domains as 3rd-party scripts load 4th-party dependencies.	Policies remain static, concise, and decoupled from external changes.
Vulnerability	Easily bypassed. If an attacker finds an open JSONP endpoint on a trusted CDN, they win.	Mathematically secure. Trust is based on explicit server intent, not location.

Location-based trust is dead. Move to Strict CSP.

Propagating Trust with “strict-dynamic”

The Problem: Strict nonces break 3rd-party analytics scripts that dynamically inject additional tags, because child tags lack the nonce.
The Solution: Adding “strict-dynamic” allows trusted scripts to pass trust down the execution chain.



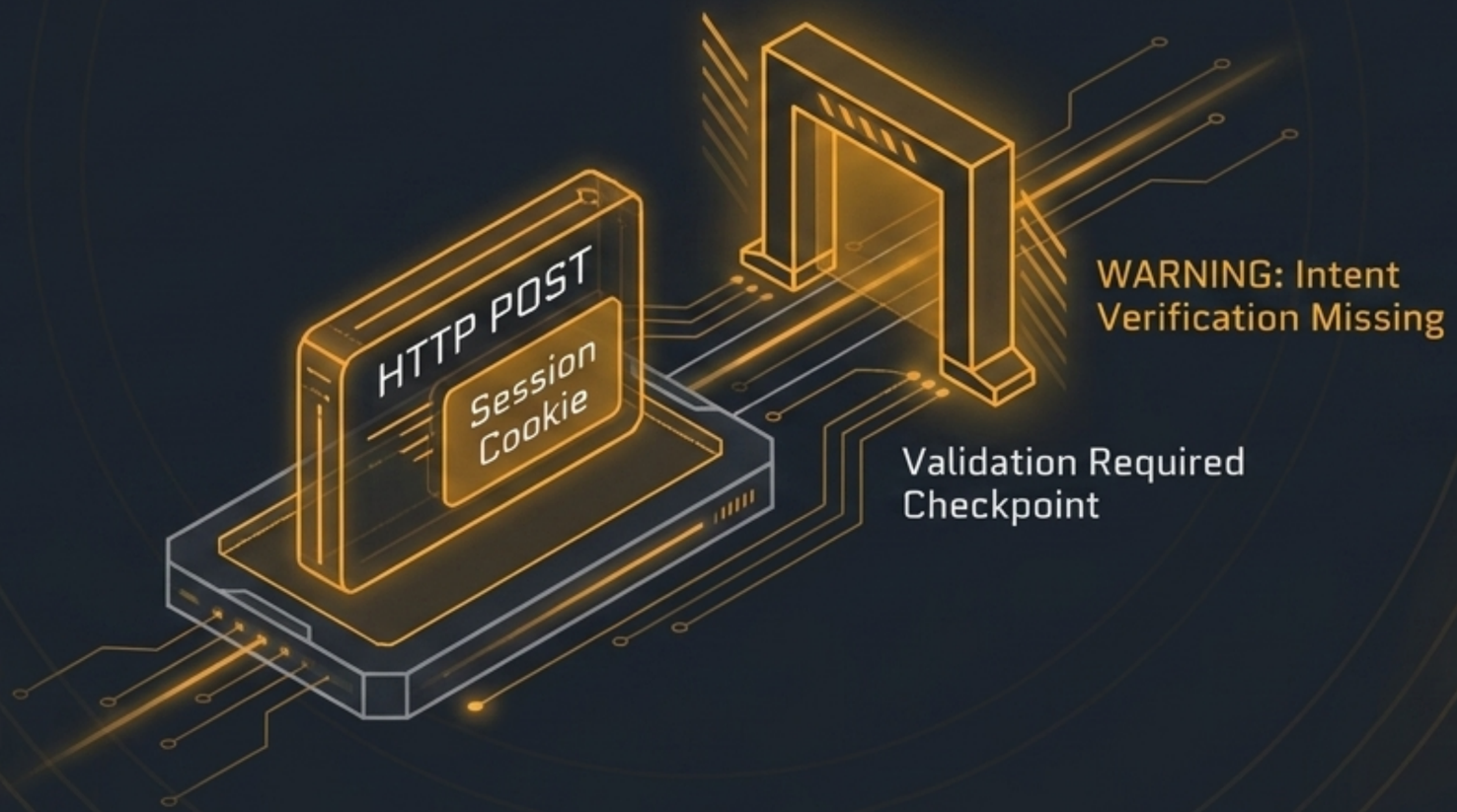
1. Server approves main.js via matching nonce.

2. main.js dynamically loads analytics.js.

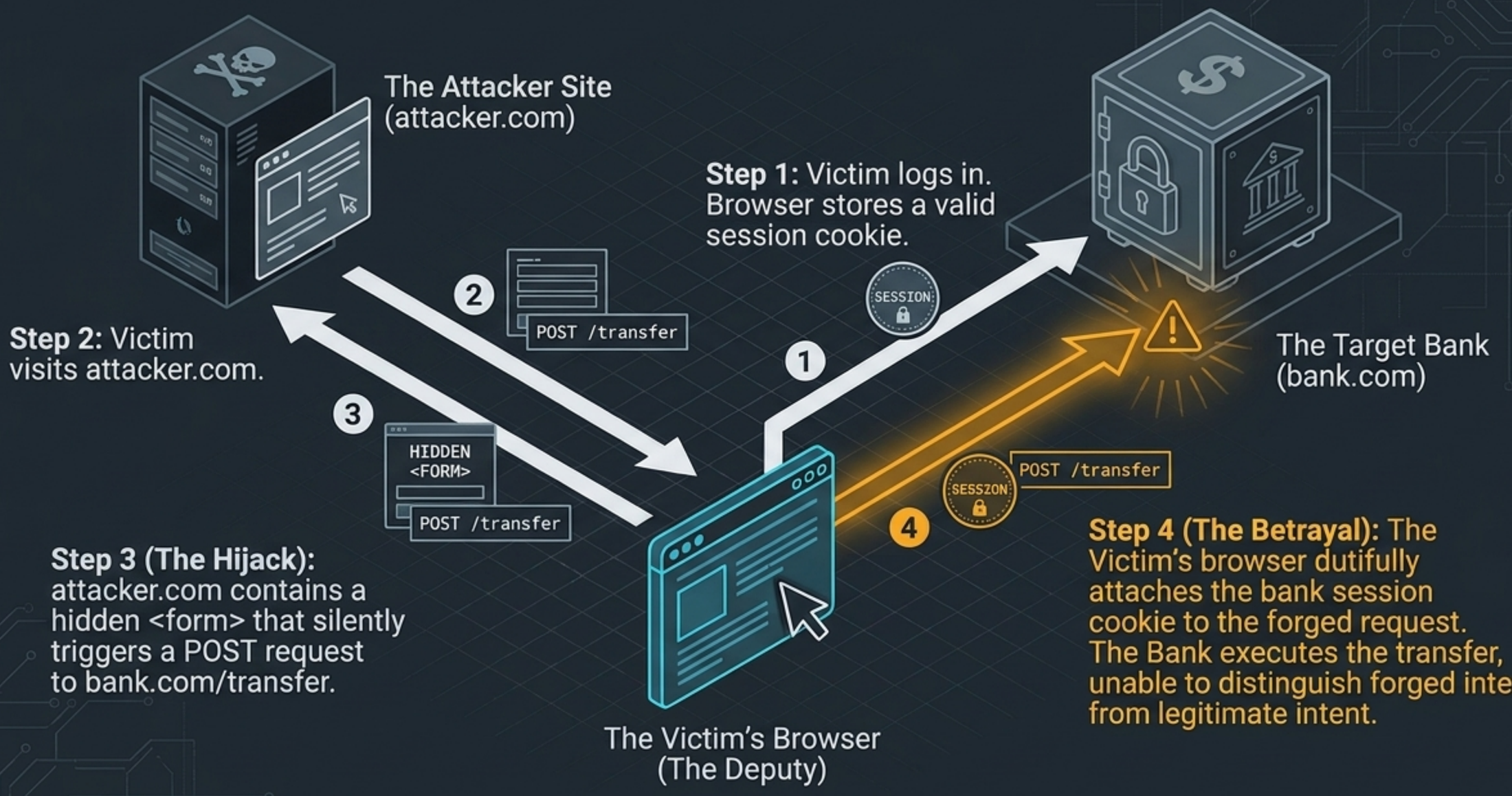
3. Because the parent is explicitly trusted, the browser automatically extends trust to the dynamically created child.

Layer 2: Securing State & Network

While CSP secures the frontend against malicious code execution, the network layer must secure the intent of HTTP requests. Browsers automatically attach cookies to cross-origin requests. Attackers exploit this behavior to forge state-changing actions.



The Confused Deputy (Understanding CSRF)



The SameSite Defense (And Its Blind Spots)

```
Set-Cookie: session_id=xyz; Secure; HttpOnly; SameSite=Lax
```

Mechanism: Prevents browser from sending cookies on cross-site requests, killing simple CSRF.

The Blind Spots (Why it's not enough)

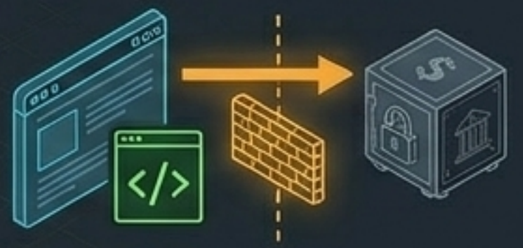
Safe-Method Mutations

Lax still sends cookies on top-level GET navigations. If a developer incorrectly mutates state on a GET endpoint, SameSite fails entirely.

Subdomain Takeovers

SameSite is scoped to the registrable domain. A compromised sibling subdomain (e.g., `blog.bank.com`) can launch a same-site attack on `app.bank.com`.

Client-Side CSRF



Malicious input causes same-origin JavaScript to issue a state-changing request, bypassing network boundaries and SameSite rules completely.

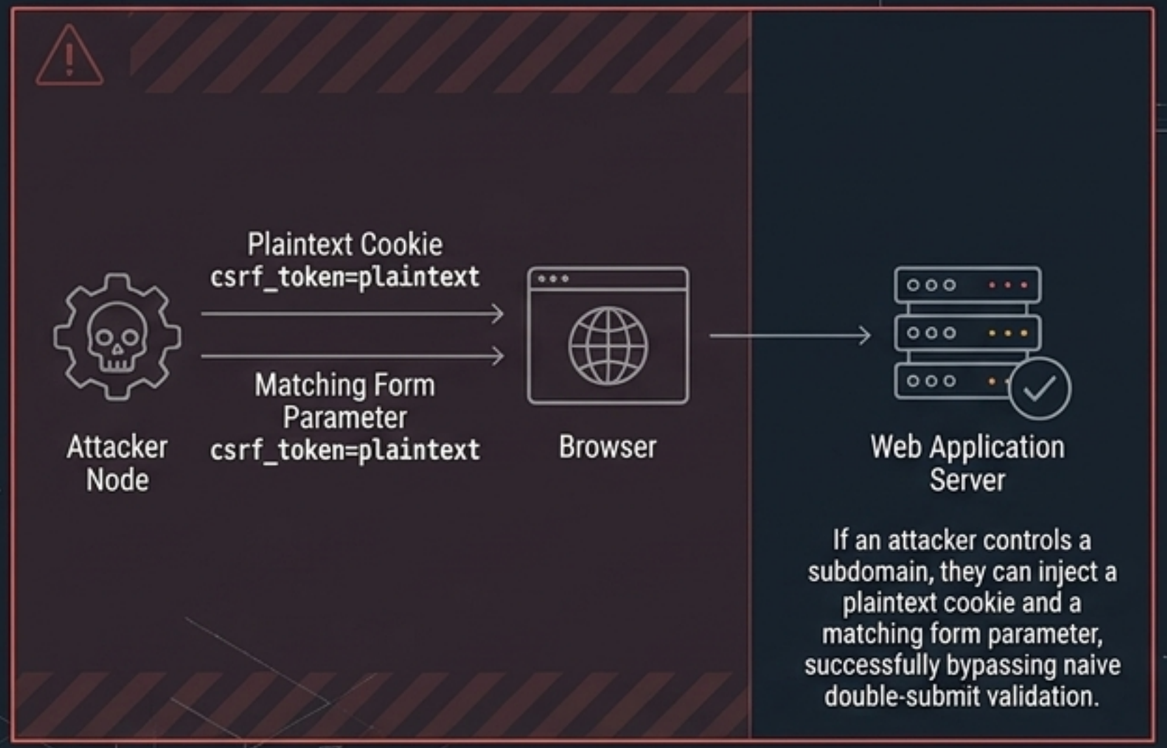
The CSRF Mitigation Matrix

Modern SPA Preferred Patterns

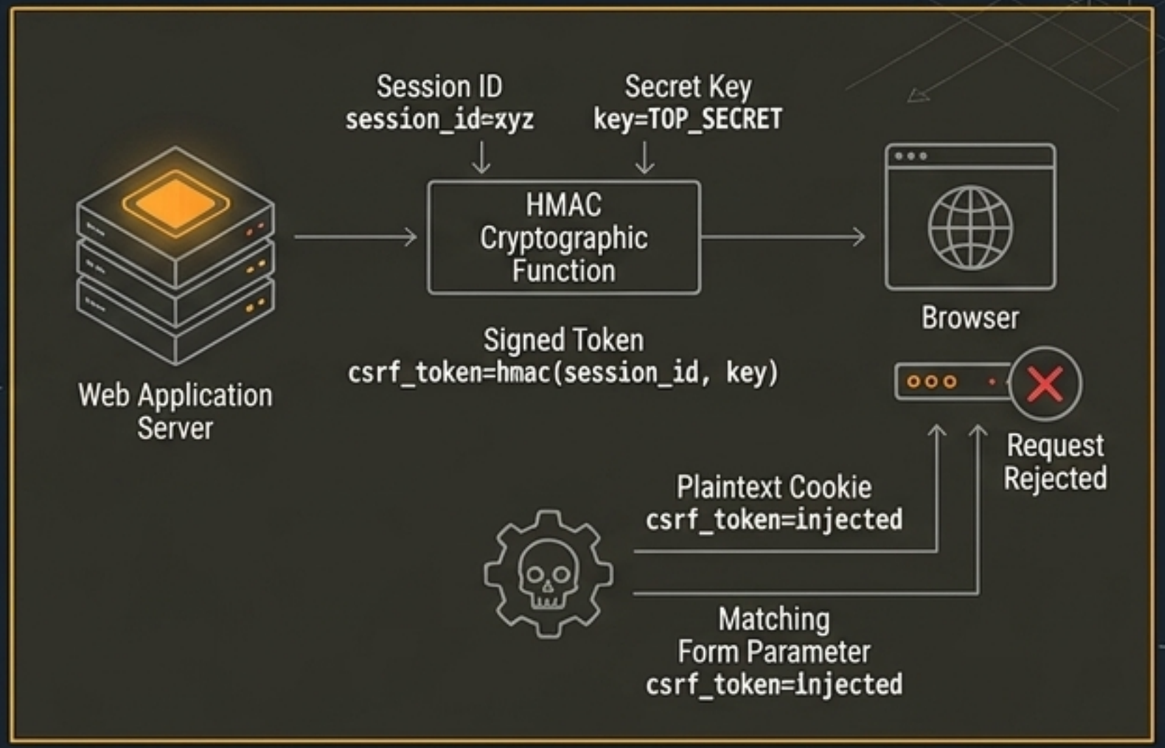
	Synchronizer Tokens	Signed Double-Submit	Custom API Headers	Fetch Metadata
Statelessness	Server state required	Stateless	Stateless	Stateless
Ease of Implementation	High friction	Medium friction	Zero-friction (APIs only)	Native browser feature
Threat Coverage	High	High (if HMAC'd)	High (relies on CORS)	High (with standard fallbacks)

Proving Intent: The Signed Double-Submit Cookie

The Naive Flaw



The Signed Solution



- The server binds the CSRF token to the user's Session ID using a cryptographic HMAC and a secret key.
- The token is delivered to the client.
- On submission, the server recalculates the HMAC using the active Session ID. If the injected token doesn't match the active session, the request is destroyed.

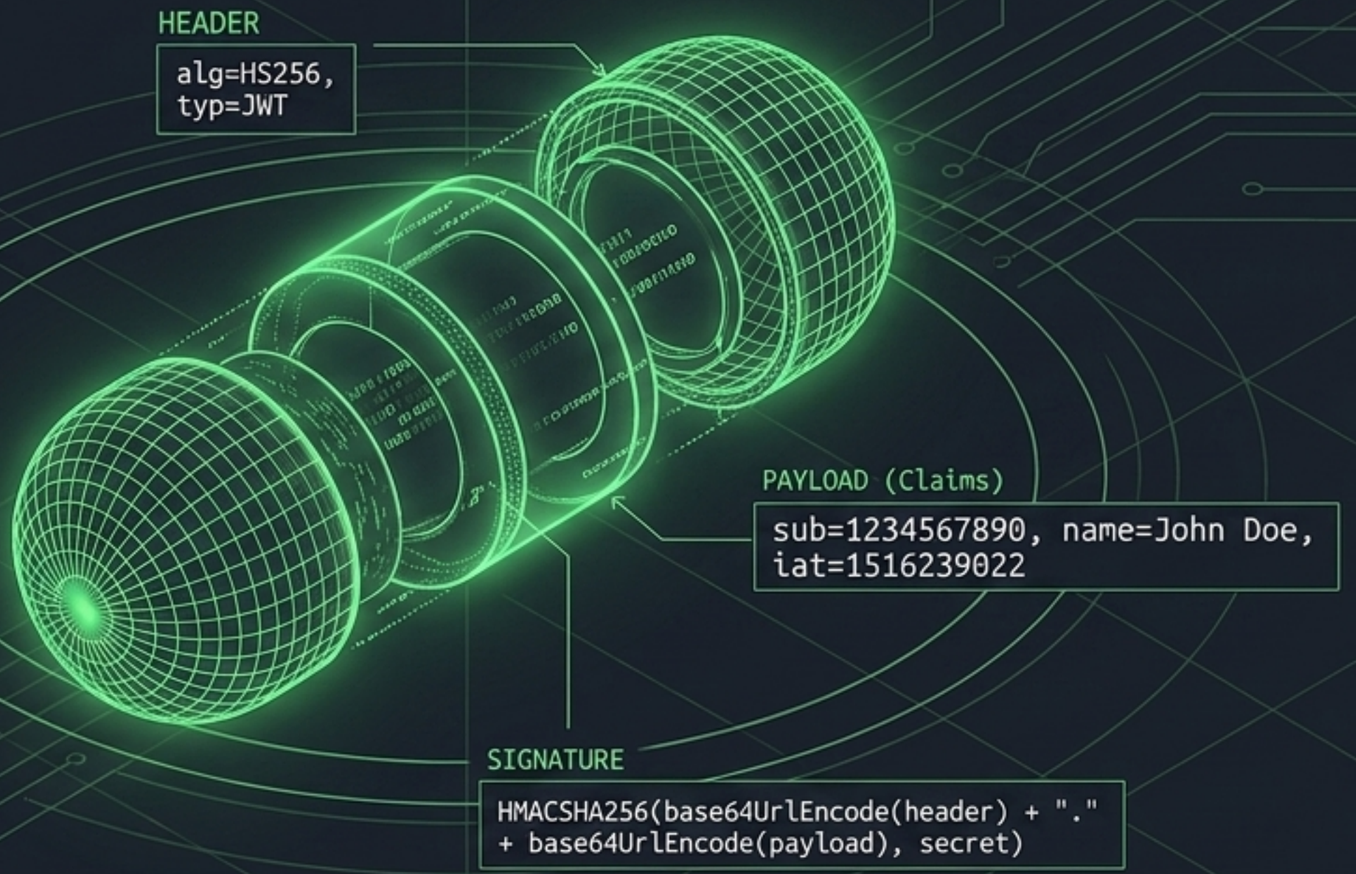
Modern Native Defense: Fetch Metadata

Modern browsers send `Sec-Fetch-*` headers detailing the exact context of a request, allowing the server to block obvious cross-site forgery at the load balancer or middleware level.

Takeaway: A lightweight, robust defense that requires virtually zero client-side coordination.

Layer 3: Securing Identity & APIs

Architectural Briefing Note: With the frontend secured by CSP, and network state secured by CSRF tokens, the final layer protects the identity of the user communicating with stateless APIs. JSON Web Tokens (JWTs) provide decentralized, cryptographically verifiable claims—but misconfiguration turns them into a catastrophic liability.



Anatomy of a JSON Web Token

Header

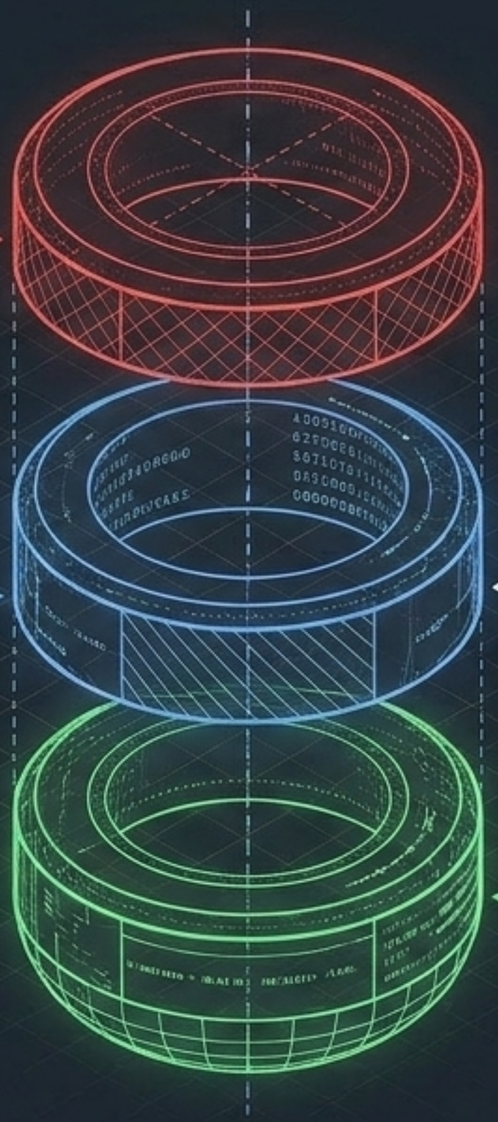
```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Defines the cryptography.

Payload

```
{
  "sub": "user_123",
  "exp": 1715000000
}
```

Contains the Claims.



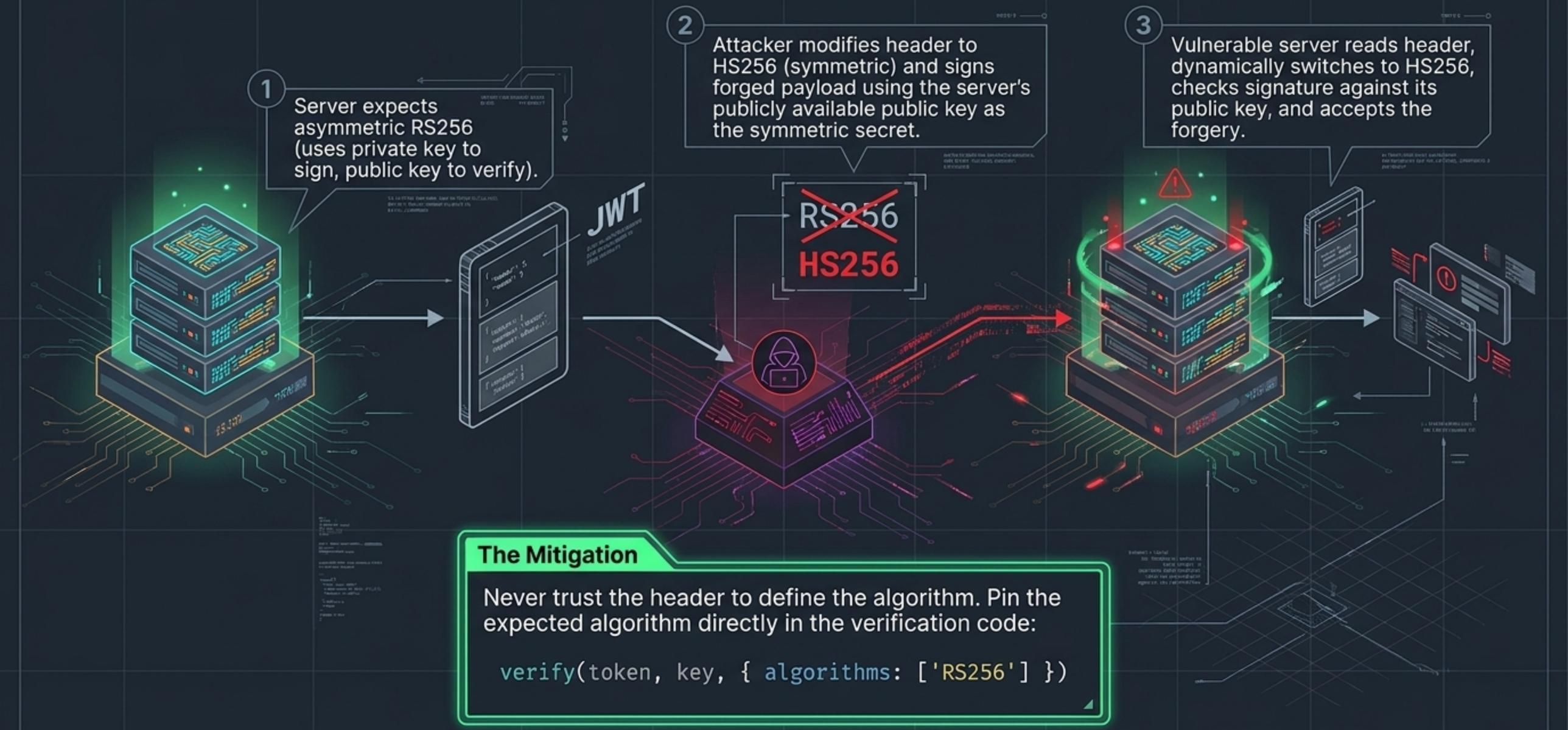
Encoded, NOT Encrypted

Never store passwords, PII, or internal secrets here. Anyone can read this.

Signature

The mathematical seal. Created by signing the encoded Header + Payload with the server's private key. Ensures the data has not been tampered with.

The Algorithm Confusion Attack



The Mitigation

Never trust the header to define the algorithm. Pin the expected algorithm directly in the verification code:

```
verify(token, key, { algorithms: ['RS256'] })
```

Rigorous Claim Validation

> IDE

> exp (Expiration):	Reject expired tokens. Factor in 30-60 seconds of clock skew.
> iss (Issuer):	Hardcode the expected issuer URL. Do not trust the token's claim blindly.
> aud (Audience):	Ensure the token was explicitly minted for your specific API service.
> typ (Type):	Prevent cross-JWT confusion. Explicitly type tokens (e.g., application/at+jwt) so an ID token cannot be submitted as an Access token.
> Hostile Headers:	Strip or strictly allowlist dynamic key fetching headers like 'jku' and 'x5u' to prevent Server-Side Request Forgery (SSRF).

Diagnostic: JWT Storage Locations

	`localStorage` / `sessionStorage`	In-Memory (JavaScript Variable)	`httpOnly` `Secure Cookie`
XSS Threat	FATAL. Any compromised JavaScript can instantly read and exfiltrate the token.	Highly Resistant (Token disappears on reload, hard to extract).	Safe (JavaScript cannot read the cookie).
CSRF Threat	Safe (tokens are manually attached to headers).	Safe.	Vulnerable (Requires CSRF mitigations from Layer 2).

Conclusion: Use **httpOnly cookies** for SPAs (backed by CSRF tokens) or In-Memory storage. Avoid `localStorage` entirely.

The Revocation Decision Tree

Do you need immediate session termination?

>_ EXPIRY: 5-15m

Short Expiration + Refresh (Simple & Standard)

Access tokens expire in 5-15 mins. Server maintains a revocable list of long-lived, opaque Refresh tokens.

>_ EXPIRY: 5-15m
>_ REFRESH: REVOCABLE_LIST

>_ KEY: JWT_JTI

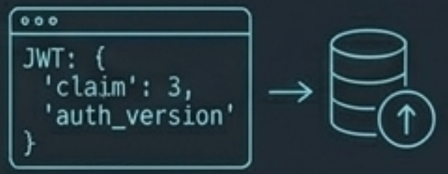
Token Denylist (High Security/Stateful)

Store the JWT's "jti" (Token ID) in a high-performance Redis cache upon revocation. Fast, but reintroduces state.

>_ STORGE: REDIS_CACHE
>_ KEY: JWT_JTI

>_ CLAIM: COUNTER

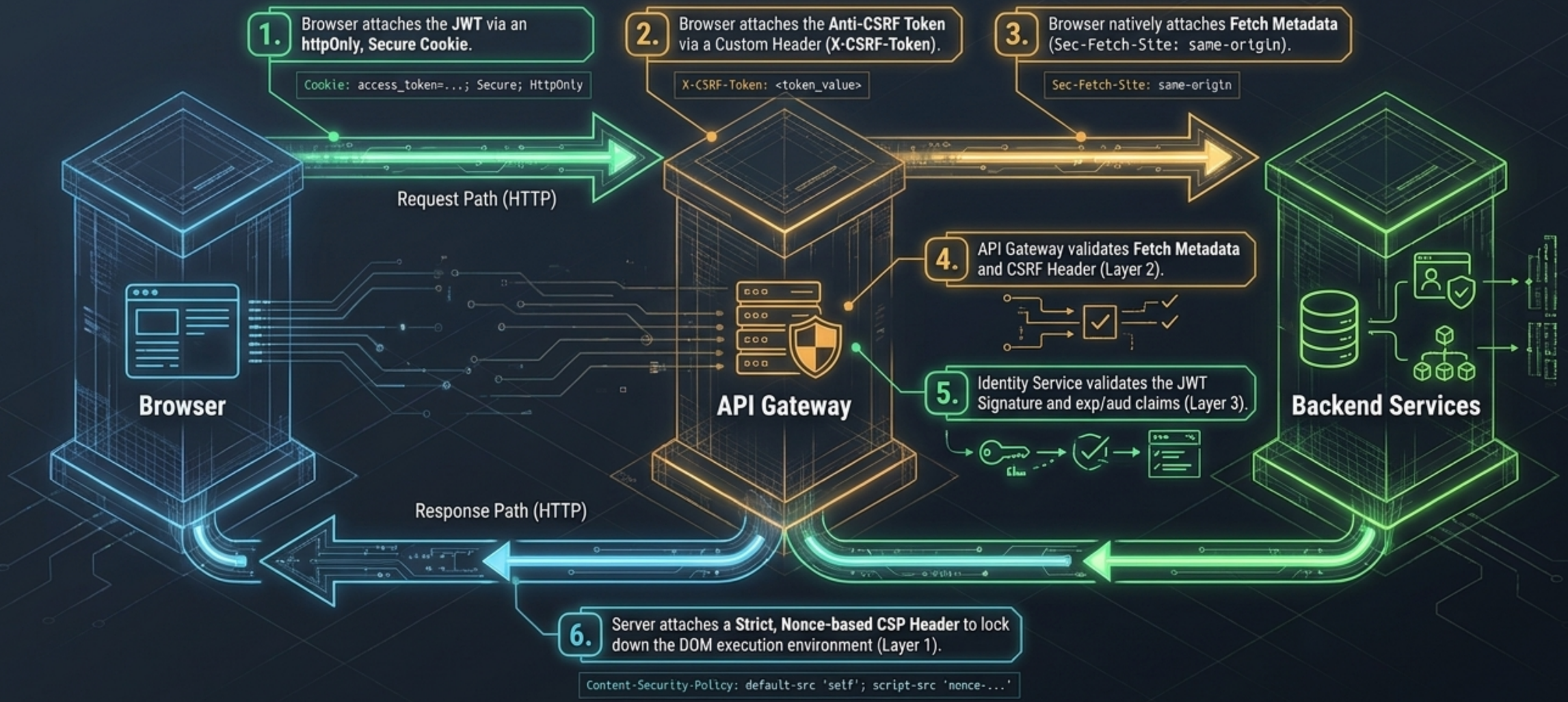
Versioned Claims (Versioned Architecture)



Add an "auth_version" integer to the JWT. If the user changes their password, increment the integer in the database. All existing JWTs instantly fail validation.

>_ CLAIM: auth_version (INT)
>_ VALIDATION: INSTANT_FAILURE

Synthesis: The Unified Secure Request



Security is an Architectural Discipline

Individual mitigations will fail. Tokens will be leaked. Scripts will be injected.
True application security is achieved not by eliminating all vulnerabilities,
but by building an architecture where the failure of one component
is safely absorbed by the resilience of the next.

Rethink the Blueprint.
Build Defense in Depth.