CASE STUDY #2

Embedding a Chatbot on Any Website With a Single Script Without Becoming a Security Liability for the Host Site

Askme Bot -- Embed Architecture, Script Safety and Host Page Isolation

Project	Askme Bot -- Customisable AI Chatbot Platform
Area	Embed Architecture, Third-Party Script Safety, Host Page Isolation
Difficulty	High -- Third-party script trust model, CSP compatibility, performance impact on host
Outcome	Single-script embed with full host page isolation, minimal footprint, and CSP-compatible delivery

Background

Askme Bot's embed feature was one of its most visible selling points. A user would create and configure their bot, copy a single script tag, and paste it into any website. The bot would appear as a chat widget on that page without any further development work. The simplicity was the feature.

What looked simple from the outside carried significant responsibility on the engineering side. A script tag injected into a third-party website runs with the same permissions as every other script on that page. Done naively, embedding Askme Bot onto a website would give Askme Bot's code full access to that website's DOM, its form inputs, its cookies, and its users' interactions. For website owners, this is exactly the kind of third-party script risk that has caused major data breaches. Building an embed that was genuinely safe for the host site, not just safe enough, was a non-negotiable engineering requirement.

The Problem

Problem 1 -- An Inline Script Injection Has Full Page Access by Default

The prototype embed worked by injecting a script that appended a div to the host page's body and rendered the chat widget directly into the host page's DOM. This meant the widget's JavaScript executed in the same origin as the host page. It could read any element on the page, listen to any event, access localStorage and sessionStorage, read cookies not marked HttpOnly, and make requests with the host page's session credentials.

From the host website owner's perspective, this was indistinguishable from a cross-site scripting attack. The Askme Bot script was trusted to behave well, but website owners had no technical

guarantee of that. Any vulnerability in Askme Bot's script delivery, or any compromise of the script file itself, would immediately expose every page it was embedded on.

The Third-Party Script Problem

Third-party scripts are one of the most common vectors for web supply chain attacks. A script loaded from an external domain and injected into the host page is fully trusted by the browser. If that script is compromised, everything on the page is compromised. Website owners who embed third-party chat widgets face this risk constantly, and any embed that does not actively constrain its own access makes that risk worse.

Problem 2 -- The Embed Script Blocked Host Page Rendering

The initial script tag was loaded synchronously in the page head, which is the most natural place website owners would paste it following common third-party script conventions. A synchronous script in the head blocks HTML parsing until the script has been downloaded, parsed, and executed. On slow connections or when Askme Bot's CDN was under load, this meant the host page's content was invisible to the user until the chat widget script had fully loaded. The widget was a supplementary feature on the host site. It was making the host site's own content load slower for every visitor, whether they used the chat widget or not.

Problem 3 -- The Embed Was Incompatible With Strict Content Security Policies

Many security-conscious websites implement a Content Security Policy that restricts which scripts, frames, and resources can load on the page. A CSP with a restrictive script-src directive would block the Askme Bot embed script from loading at all. An even stricter CSP would block the widget's subsequent API requests to Askme Bot's servers. Website owners who wanted to embed Askme Bot while maintaining their security posture had no clean way to do so. The only option was to add broad CSP exceptions that weakened their policy for all scripts, not just Askme Bot.

The Solution

1. Sandboxed Iframe as the Actual Widget Container

The embed architecture was rebuilt from the ground up with the widget living inside a sandboxed iframe rather than in the host page's DOM. The loader script, the only code that ran directly on the host page, was stripped down to a single responsibility: create the iframe, set its sandbox attributes, point it at Askme Bot's widget origin, and append it to the page. Every line of actual widget code, the chat UI, the API calls, the conversation state, ran inside the iframe on Askme Bot's own origin.

The iframe's sandbox attribute was configured to allow only what the widget genuinely needed: allow-scripts for the widget JavaScript, allow-forms for message submission, and allow-same-origin scoped to the iframe's own origin. The allow-same-origin permission applied within the iframe context, meaning the widget's scripts could access the iframe's own storage,

but had no access to the host page's storage, cookies, or DOM. The host page's JavaScript equally could not reach into the iframe.

2. Async Deferred Loader With Zero Render Blocking

The loader script was redesigned to be loaded with both the async and defer attributes, ensuring it never blocked HTML parsing or rendering on the host page. The script was also moved out of the page head guidance in the embed instructions, with a recommended placement just before the closing body tag. The iframe was created and appended to the page only after the DOMContentLoaded event fired, guaranteeing the host page's own content was fully parsed and rendering before the widget made any appearance.

The widget loaded progressively inside the iframe: the chat button appeared first using minimal CSS, and the full conversation interface loaded only when the user clicked to open the widget. A user who never opened the widget added almost zero load to the host page beyond the initial small loader script.

3. CSP-Compatible Resource Delivery With Explicit Domain Declarations

The embed documentation was updated with the exact CSP directives needed to allow Askme Bot's resources without opening broad exceptions. Website owners who ran strict CSPs could add Askme Bot's specific CDN domain to frame-src and the widget API domain to connect-src, permitting exactly what the widget needed and nothing more. The loader script was hosted on a stable, versioned URL so website owners could add a specific script hash to their script-src if they preferred hash-based allowlisting over domain-based allowlisting.

Askme Bot's servers also sent appropriate CORS headers scoped to the bot's configured allowed domains, so API requests from the widget iframe were only accepted from origins the bot owner had explicitly permitted during setup.

4. Controlled postMessage Interface for Host Page Integration

Some website owners needed the widget to integrate lightly with their page: pre-filling the user's name from a logged-in session, opening the widget automatically when a user visited a specific page, or receiving an event when the widget was closed. These integrations required some communication between the host page and the widget iframe. Rather than relaxing the iframe isolation, a narrow postMessage interface was defined with an explicit allowlist of message types the widget would accept and emit.

The host page could send a set-user-context message with a name and optional metadata. The widget would emit a widget-closed event when the user dismissed it. Any postMessage not matching the defined schema was silently discarded. The interface gave host pages the integration hooks they needed without giving their JavaScript any access to the widget's internal state or conversation content.

Problem	Solution
Inline script has full access to host page DOM and data	Sandboxed iframe: widget runs on Askme Bot origin with no host page access

Synchronous script blocks host page rendering	Async deferred loader, widget UI loads only on user interaction
Embed incompatible with strict host page CSP	Documented exact CSP directives; stable versioned URL for hash allowlisting
No integration between host page and widget possible	Narrow postMessage allowlist for specific approved message types only

Outcome

After the iframe-based embed shipped, website owners could add Askme Bot to their site with confidence that the embed would not become a security liability. The sandboxed iframe meant Askme Bot's code had no access to the host page regardless of what was running on it. The async deferred loader meant the widget added no measurable load time to the host site for users who never opened it. CSP-compatible delivery meant security-conscious sites could embed the widget without weakening their policies. The postMessage interface gave the integration hooks that developers needed without creating an open channel between the host page and the widget.

Key Takeaway

A third-party script that runs on someone else's website is a guest in that website's security model. A good guest does not help themselves to everything in the house. The right embed architecture actively constrains its own access rather than relying on the host site trusting the embed to behave. Sandboxed iframes, async loading, and explicit CSP guidance are how you build an embed that website owners can add without second-guessing whether they should. That trust is what makes the embed feature worth having in the first place.

-- Ehsan