LEARN

Secure Vanilla Comms

Encrypting GET/POST payloads between a vanilla HTML/JS frontend and a PHP backend, with no frameworks and no Node.js.

Abstract: Securing data in transit usually means TLS certificates plus a heavy JS crypto library. This guide covers a leaner, complementary approach: encrypting the payload itself with a small, synchronized stream cipher shared between a vanilla JS frontend and a PHP backend — on top of HTTPS, not instead of it.

> **Important:** this is application-layer obfuscation for payloads you control end to end (your own frontend talking to your own backend) — it is not a replacement for TLS, and rolling your own cipher is never a substitute for vetted, audited cryptography in anything security-critical. Treat it as an extra layer for a same-origin app, not a way to skip HTTPS.

1. The core concept

Encrypt the payload before it leaves the browser, decrypt it only once it reaches the PHP backend, using the same algorithm and the same shared key material ported to both languages. See the Veil-10 Stream Cipher tool on this site (ruggi.site/tools/VeilStream/) for a live example — a 10-seed XOR+offset stream cipher with matching JS and PHP implementations you can study directly.

2. The vanilla JavaScript frontend

Skip heavy libraries like CryptoJS. Pure JS is enough for this.

- **Cipher:** a function that walks the input string byte by byte, XORing against a rotating key-string byte and mixing in a small per-key numeric offset — the same shape as a classic stream cipher, just self-rolled.
- **Encoding:** encode the resulting byte string to Base64 with the native btoa() so it survives transport as plain text.
- **Transmission:** plain fetch(), no library needed (see below).

```
fetch('server.php', {
  method: 'POST',
  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
  body: 'payload=' + encodeURIComponent(encryptedBase64Text)
});
```

3. The PHP backend

The server reverses the exact same steps — no Composer dependencies required.

- **Receive:** `$data = $_POST['payload'] ?? '';`
- **Decode:** `base64_decode($data)` back to the raw cipher bytes.

- **Decrypt:** run the PHP port of the same cipher, using the exact same key set as the frontend.
- **Respond:** if a response payload needs to go back encrypted too, encrypt it the same way and echo Base64; the frontend reverses it with atob().

4. Security considerations — read this part

- **No supply-chain surface:** writing the cipher in plain JS/PHP avoids pulling in an npm dependency tree that could itself be compromised.
- **Not a TLS replacement:** always serve over HTTPS regardless. This technique adds a layer, it doesn't remove the need for transport security.
- **Key distribution is the real weak point:** the JS-side key ships to every browser that loads the page — anyone can view-source it. This scheme is meant to obscure casual payload inspection and add a second layer against a compromised CDN/proxy, *not* to keep a determined attacker with browser access out. For anything where that distinction matters, use standard, audited crypto (WebCrypto API + a real TLS setup) instead.
- **Offline-capable:** a page using only this technique can run from a local file:// context talking to a LAN PHP server, which is genuinely useful for closed, trusted networks — just don't reach for it as your only defense on the open internet.