Blindflare: A Zero-Trust Encrypted Web System

Fang Judo

bling-glare-motion@1337.legal

www.blindflare.org

Abstract. Modern applications are built on an implicit trust that servers will behave. They are trusted to hash your password, store your secrets, protect your identity, and respect your privacy. This trust is often misplaced. Every compromise from database leaks to insider threats reinforces this. Blindflare introduces a system where the server is stripped of all cryptographic authority. Clients generate their own ECC keypairs. All data encryption and decryption is performed locally. Authentication is key-based. Storage is encrypted. Passwords are never transmitted. We define a new model called ztStorage (zero-trust storage). Encrypted blobs are stored on a remote server but the encryption is fully client-side, with the server having zero knowledge of the contents. This architecture ensures that even a fully compromised backend yields no plaintext, no password, and no metadata leakage. By encrypting all data above the transport layer, Blindflare also eliminates plaintext exposure at the reverse proxy layer especially from gateways like Cloudflare, which terminate TLS and can otherwise view, log, or alter traffic when the requests are decrypted by there gateways.

1. Introduction

The evolution of centralized web services has created a fragile and overly trusting security model. In today's applications, backends routinely receive plaintext credentials, session tokens, and user data, and are expected to handle them securely. Unfortunately, frequent breaches, misconfigurations, and reliance on third-party services have shown that this model is increasingly untenable. To address these challenges, systems must be redesigned around the principle of cryptographic ownership rather than relying on server-side policy enforcement. Blindflare embraces this principle by ensuring that servers only store data but never understand it. It achieves this by employing a secure authentication protocol where clients generate their own ECC keypairs and authenticate through key-based mechanisms, eliminating the need to transmit passwords or private keys. All hashing and key derivation are performed client-side during login and registration flows, ensuring that sensitive information is never exposed to the server. Furthermore, Blindflare introduces a novel zero-trust storage model, called ztStorage, which stores encrypted data blobs remotely but with encryption fully handled by clients. This approach guarantees that even if the server is fully compromised, no plaintext data, passwords,

or metadata can be leaked. By encrypting data above the transport layer, Blindflare also prevents exposure of plaintext data at reverse proxies like Cloudflare, which commonly terminate TLS and can otherwise access or log traffic. Thus, the system eliminates any server-side plaintext data handling, including password management, reinforcing a security model where the client controls authentication, encryption, and validation entirely.

2. Cryptographic Foundations

Blindflare relies on Elliptic Curve Cryptography (ECC), specifically the secp256k1 curve, which is widely used in Bitcoin, for public key derivation and digital signatures. Each client generates cryptographic keys using the BIP39 mnemonic standard, providing users with a human-readable backup mechanism while maintaining cryptographic security. The key generation process begins with the creation of a 12 or 24-word BIP39 mnemonic phrase using a cryptographically secure random number generator, providing 128 or 256 bits of entropy respectively. This mnemonic is then converted to a 512-bit seed using PBKDF2-SHA512 with 2048 iterations, using the mnemonic as input and "mnemonic" plus an optional passphrase as salt, following the BIP39 specification. The first 256 bits of the derived seed serve as the private key k, from which the public key K is derived as a point on the secp256k1 curve:

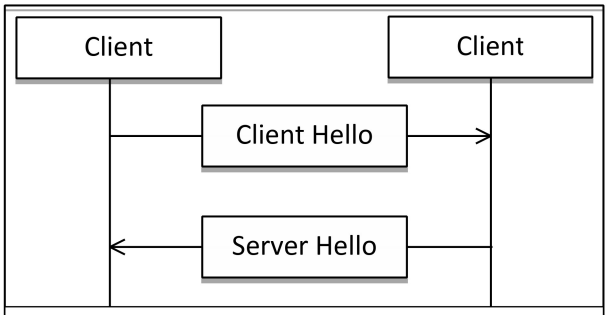
$$K = k \cdot G$$

Where G is the generator of the curve. This approach ensures compatibility with existing Bitcoin and cryptocurrency tooling while providing users with familiar backup mechanisms. The client retains the private key securely in encrypted local storage, but users can backup and restore their entire cryptographic identity using only the BIP39 mnemonic phrase. This human-readable format eliminates the complexity of managing raw cryptographic keys while maintaining the same security properties. The optional BIP39 passphrase, sometimes called the "25th word," provides an additional layer of security, creating a two-factor authentication mechanism where both the mnemonic and passphrase are required for key recovery. For users choosing to log in with a traditional password, the system maintains backwards compatibility by deterministically deriving a BIP39 mnemonic from the password and a salt value using a strong key derivation function such as Scrypt, Argon2, or PBKDF2. The derived mnemonic then follows the standard BIP39 to seed to private key flow described above. This process enables the regeneration of the same private key locally without ever transmitting the password, while still providing users with mnemonic backup capability. Crucially, the private key is never transmitted to the server under

any authentication method. During authentication, the user sends a hash of their private key to the server. Specifically, the private key is hashed using SHA-512 to produce a fixed-length digest, which the server uses to identify the user without ever receiving the raw private key, password, or mnemonic phrase.

3. Architecture and Handshake

The Blindflare protocol implements a formal handshake mechanism with capability negotiation that operates independently of user authentication. This architectural separation between cryptographic session establishment and user identity verification enables superior forward compatibility and security properties while allowing the protocol to evolve safely over time. The handshake process begins with a ClientHello message where the client declares its supported cryptographic capabilities, including encryption algorithms, elliptic curve preferences, and serialization formats. The client includes its public key, a cryptographically secure random nonce, and a timestamp to establish temporal bounds for the exchange. To prevent replay attacks, the client signs a canonicalized representation of the message that binds together the protocol version, timestamp, nonce, and public key using deterministic ECDSA. This signature proves possession of the private key while ensuring the message cannot be replayed by an attacker.The server responds with a ServerHello message that selects the specific cryptographic parameters from the client's offered capabilities and establishes the session key that will protect all subsequent communications. The server includes its own public key identified by a key identifier for rotation purposes, generates its own nonce for mutual authentication, and creates a random session key that it encrypts to the client's public key using ECIES. The server's response is itself signed to prove authenticity and prevent man-in-the-middle attacks. Upon receiving the ServerHello, the client verifies the server's signature against a pinned server public key, decrypts the session key using its private key, and stores both the session key and the negotiated cryptographic parameters for the duration of the session. All application traffic following the successful handshake is protected by a comprehensive message envelope system that provides authentication, integrity protection, and replay prevention. Each message contains versioned metadata including the server's key identifier, a fresh timestamp, a unique nonce, and a message type indicator, all of which are cryptographically bound to the encrypted payload through AES-GCM's Additional Authenticated Data mechanism. This binding ensures that attackers cannot tamper with message metadata without detection, as any



Hello proves key ownership and declares capabilities. The acknowledgment picks parameters, names the server key, and delivers the session secret.

modification would cause authentication failures during decryption. Timestamps are validated within a sliding window to account for clock skew while preventing stale message replay, and nonces are cached in a bounded least-recently-used structure to detect and reject duplicate messages. Session keys are derived using HKDF-SHA256 when the protocol employs ephemeral Elliptic Curve Diffie-Hellman key agreement for enhanced forward secrecy. The key derivation uses the ECDH shared secret as input key material, the concatenation of client and server nonces as salt, and a protocol-specific info string to produce a 32-byte AES key. All signatures throughout the protocol use deterministic ECDSA as specified in RFC 6979 to eliminate the risk of nonce reuse vulnerabilities that have historically compromised ECC implementations. When signatures are required over serialized data structures, the protocol mandates use of the JSON Canonicalization Scheme defined in RFC 8785 to ensure consistent byte-level representation across different implementations and platforms. The protocol's serialization strategy supports multiple wire formats through the capability negotiation process, allowing deployments to optimize for their specific requirements. JSON serves as the default format due to its ubiquity and ease of debugging, with canonical serialization applied to any signed content to ensure reproducible signatures. CBOR provides a compact binary alternative with deterministic encoding properties that reduce bandwidth requirements for resource-constrained environments. Protocol Buffers offer the most structured approach with strict schema validation, excellent multi-language tooling support, and natural forward compatibility through oneof field evolution, making them ideal for large-scale deployments with diverse client implementations.

3. Authentication and Login Protocol

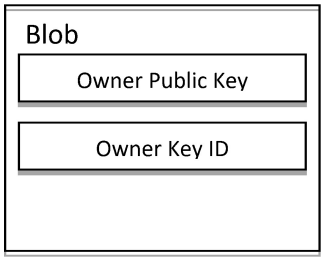
The authentication process begins with the client generating a new ECC keypair. Instead of sharing a password or private key with the server, the client hashes its private key using a cryptographic hash function, such as SHA-512, and sends only the resulting digest along with the associated public key. The server stores this hash and public key, and uses the hash to recognize and verify users during login. At no point does the server receive or store passwords or unencrypted keys all sensitive operations occur entirely on the client side. After successful identification, the server responds by issuing a session token and its own public key. A temporary shared secret is then established between the client and server using Elliptic Curve Diffie-Hellman (ECDH), calculated by combining the client’s private key with the server’s public key:

$$shared = k_{client} \cdot K_{server}$$

This shared secret acts as the symmetric key for all subsequent encrypted API communications between the client and server. Both request and response payloads are encrypted using this key, ensuring confidentiality even if transport encryption is intercepted or terminated early by intermediaries.

4. ztStorage: Zero-Trust Client-Side Encrypted Blobs

In Blindflare, clients construct data “blobs” entirely on their own devices before uploading to the server. Imagine preparing an object containing fields like name, email, and notes all serialized into a compact form and encrypted with a symmetric key using a modern algorithm like AES-GCM. This encrypted payload is what gets uploaded. To ensure that only the rightful owner can decrypt the data, that symmetric key itself is encrypted using the user's public key through an ECIES-like mechanism. Thus, the server stores both the encrypted payload and the encrypted key together. Before uploading, the client also signs the ciphertext with its private key using ECDSA over a SHA-256 hash. The server retains the resulting signature alongside the encrypted data, but cannot verify or modify it the client has the sole ability to check the signature upon later retrieval. A final blob object might conceptually include four inline fields:



Once uploaded, the server treats this blob as an opaque object it stores the four parts exactly as given, without any ability to decrypt or understand them. Since all core cryptographic operations encryption, decryption, signing, and verification happen entirely on the client side, the server remains absolutely blind to the actual contents, ensuring robust zero-trust storage.

5. Encrypted API Transactions

Once the client and server have established a secure communication channel using Elliptic Curve Diffie-Hellman (ECDH), all subsequent API requests and responses are encrypted symmetrically using the derived shared secret. This shared secret is computed independently on both sides by combining the client’s private key with the server’s public key. Each API request from the client includes two essential elements: a field containing the encrypted payload (for example, labeled simply as "data"), which allows the server to derive the same shared secret. The actual request content parameters, data, metadata is entirely encrypted using this shared key, ensuring that only the intended recipient can decrypt and process it. The server, having access to its own private key and the client’s public key (saved in the user’s session or in payload for state-less model), computes the same ECDH shared secret and uses it to decrypt the request payload. It

then processes the request, encrypts the response with the same shared key, and returns it to the client. This mechanism ensures full confidentiality even if the underlying TLS connection is terminated or observed by an intermediary such as a reverse proxy. The payload remains opaque, as only the client and server possess the necessary private keys to derive the shared encryption key. In effect, every API interaction between the client and server becomes a sealed message exchange, layered on top of any transport protocol, with payload visibility restricted strictly to the endpoints in possession of valid key material.

6. Defense Against Reverse Proxies

Transport Layer Security (TLS) is commonly used to protect web traffic in transit. However, in real-world deployments, TLS often terminates at reverse proxies such as Cloudflare or Nginx before reaching the backend application server. This architecture exposes plaintext content at the proxy layer, making it possible for these intermediaries to inspect, log, or even modify sensitive user data, despite the appearance of a secure HTTPS connection. Blindflare is specifically designed to render such inspection meaningless. All sensitive data is encrypted at the application layer by the client before it ever enters the network stack. This means that even when a reverse proxy decrypts TLS traffic, the actual content of requests and responses remains unintelligible. From the perspective of the proxy, all it can observe are the envelope metadata (version, timestamp, message type), encrypted data blobs, and cryptographic material none of which contain readable user content or enable impersonation attacks. The envelope system's AAD binding ensures that proxies cannot tamper with metadata without detection, as any modification would cause authentication failures during decryption. By applying end-to-end encryption independently of the transport layer, Blindflare ensures that no trusted intermediary no matter how privileged can access or interfere with user data. This design decisively closes the gap that TLS alone cannot protect, making reverse proxies effectively blind to the application's inner workings.

8. Key Management and Rotation

Blindflare implements comprehensive key management with support for key rotation and identifier-based key resolution. Both client and server keys are identified by unique key identifiers (kid) included in message envelopes, enabling seamless key rotation without breaking existing sessions. Server public keys are published through a signed JWKS-like endpoint that enables secure key pinning by clients. Server static keys are rotated on a scheduled basis, with the server maintaining the last n keys to support decryption of older session tickets and ongoing communications. For enhanced forward secrecy, the protocol supports ephemeral key exchange where session keys are derived from ephemeral ECDH rather than static key encryption. This ensures that compromise of long-term keys does not compromise past session confidentiality.

9. Implementation Strategy and Migration Path

The protocol is designed for incremental adoption and backward compatibility. The recommended implementation approach:

Phase 1: Implement JSON-based handshake and envelope system

Phase 2: Add timestamp and nonce validation with replay protection

Phase 3: Introduce capability negotiation for CBOR/Protobuf support

Phase 4: Optional migration to ephemeral key exchange

The handshake can be implemented as a new /blindflare/hello endpoint, with middleware updated to handle envelope-wrapped messages. Existing JWT-based session tickets can be maintained for compatibility while gradually migrating to session-key-based authentication tokens.

7. Security Properties

Blindflare is designed to withstand a wide range of modern security threats by minimizing trust in the server and elta, or passwords. All it stores are encrypted blobs and hashed identifiers, which are computaiminating any transmission of sensitive data in plaintext. In the event of a backend server compromise, attackers gain nothing of value, as the server never receives private keys, plaintext dationally infeasible to reverse. If the database is breached, the outcome is similarly benign. Since all stored data is encrypted client-side and signed digitally, there is no readable user information available to attackers. The stored key hashes also reveal nothing about the actual private keys or passwords due to the strength of the cryptographic hash functions used. Even threats that operate at the network layer, such as inspection by reverse proxies, are neutralized. Since all application-layer data is encrypted independently of TLS, terminating TLS at gateways like Cloudflare yields only cipher-text. These intermediaries can access only public keys, encrypted payloads, and hashed authentication tokens none of which can be leveraged to reveal user data or impersonate users. The system is robust against password theft through the server because the server never receives a password in any form. Key derivation occurs client-side, and only derived hashes are sent. As a result, even if the server is compromised, it cannot leak credentials. User impersonation attacks based on password guessing are also ineffective. Since authentication is entirely key-based and the private key is never transmitted or stored server-side, an attacker would need to guess or steal the user's private key, which is practically infeasible under standard cryptographic assumptions. Lastly, the system defends against blob tampering. Every encrypted blob is digitally signed using the client's private key. This signature is verified on the client before any blob is loaded or processed, ensuring that tampered data is rejected. Overall, Blindflare transforms the server into a passive storage component while shifting all cryptographic control to the client, effectively neutralizing the most common and dangerous classes of security threats.

8. Conclusion

Blindflare demonstrates that web services can operate without seeing user secrets. By combining elliptic curve cryptography, encrypted blob storage, client-side key management, and a robust handshake protocol, we eliminate the server from every critical cryptographic operation. The introduction of ztStorage provides a clean abstraction, the server holds encrypted blobs but never understands them. The protocol ensure that this architecture can evolve safely while maintaining strong security properties. Authentication, authorization, and verification all happen on the edge the user's device.

The user owns the keys. The client owns the logic. The proxy becomes a blind courier not a custom officer.

References

[1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.
[2] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records, 2006.
[3] Boneh, D., & Shoup, V. A Graduate Course in Applied Cryptography, 2020.
[4] NIST Special Publication 800-57, Recommendation for Key Management – Part 1, 2019.
[5] RFC 7748, Elliptic Curves for Security, IETF, 2016.
[6] W. Diffie, M. Hellman. New Directions in Cryptography, IEEE Transactions on Information Theory, 1976.
[7] RFC 8017. PKCS #1: RSA Cryptography Specifications Version 2.2, IETF, 2016.