Enhancing User Experience in Web3 Dapps with Social Authentications

Paweł Nowak

neotheprogramist
NeoTheProgramist
Paweł Nowak

1 Problem Description

In the world of web3 decentralized applications (dapps), a notable user experience concern arises. Each interaction with a smart contract necessitates that users sign the transaction with a digital wallet. This process results in frequent pop-ups, detracting from the overall user experience. Furthermore, the mandatory creation of a wallet and the responsibility of saving a recovery seed poses a significant barrier to the mass adoption of cryptocurrencies. This document proposes potential solutions that allow the integration of social authentications such as Gmail, GitHub, and Discord for creating smart contract sessions. The solutions presented here are based on current knowledge; additional technical research is essential.

2 Solution Proposals

2.1 OAuth2 Issuer-Based Authentication

System Environment Setup:

Parties in the System:

- **Client/Frontend**: Acts as a session user, permitted to execute methods on the treasury. (Language: TypeScript)
- **Blockchain**: Functions as a treasury, wallet for clients, and verifier. (Language: Solidity/Cairo/Rust)
- **Issuer**: Verifies valid login events and signs messages, enabling clients to initiate sessions on the smart contract. (Language: Rust)
- **Authority**: Platforms such as Gmail, GitHub, etc.

System Workflow:

1. The Issuer registers with a provider (e.g., Gmail, GitHub) to obtain `client_id` and `client_secret`. It also sets up a webhook for the authority, facilitating user authentication tags. Additionally, the Issuer holds a private key recognized by the smart contract for session initiation.

2. Clients wishing to create a session interact with the Authority using OAuth2. During this process, they generate a session private key and provide:

   - Authority tag
   - Session public key (enabling the smart contract to verify requests)
   - Deadline

3. The client's creation request is forwarded to the Issuer, which verifies the tag's authenticity through the Authority. Once verified, the Issuer signs the request using its private key, returning a signed transaction, inclusive of a `vault_id`, to the client. The Issuer maintains a database of associations between users and their respective `vault_id`.

4. Subsequently, clients can send requests to the smart contract to establish a session. The smart contract retains the session's public key, verifying subsequent session requests and associating `vault_id` with the public key and deadline.

This approach possesses a significant drawback: the Issuer emerges as a trusted entity with the potential to manipulate users' vaults, granting it unbridled control over user balances.

2.2 Verification of Authority Authenticity using zk Proofs

One challenge to address is verifying that a trusted party, the Authority, sends a valid response to the client. Since centralized architectures underpin authorities, and they use HTTPS for secure communication, the introduction of zk proofs poses complications. An ideal solution would enable the zk proofs to confirm the authenticity of data supplied by the client, ensuring no tampering occurred. This section explores a potential solution using recursive zk proofs, which would mimic the TLS handshake, ensuring the validity of the server's response.

Challenges:

1. The client can craft server responses since both parties are privy to the shared secret from the TLS handshake.

2. To ensure authenticity, a modified TLS version might be needed. This version would let clients decrypt server messages but prevent them from crafting messages on the server's behalf.

2.2.1 Leveraging the TLS Notary

TLS Notary offers a potential solution to distinguish genuine server messages from those manipulated by clients. The proposed mechanism entails:

1. Partial decryption of requests by the notary, followed by signing the section of the request indicating the authorization status.
2. Hardcoding notary public keys into the zk proofs.
3. Implementing a hashing mechanism that thwarts users from repetitively using the zk proof chain with recorded data. This forces users to continually communicate with authorities for fresh sessions.

While promising, this method is not without vulnerabilities. The notary's ability to censor a user is reminiscent of a blockchain node's potential actions. A countermeasure might involve a system of publicly accepted notaries, complemented by zk proofs with hardcoded public keys.

Key Observations:

- The notary cannot request on the user's behalf due to the absence of the shared secret essential for message decryption.
- Clients can supply all information required for the zk proof to reconstruct the shared secret.
- Clients cannot alter the server's response since it is signed by the notary.

It is paramount to ensure that the zk program remains impervious to client manipulations. Thorough scrutiny is necessary to determine whether the entire process can be recreated within the zk program while extracting relevant HTTP traffic data.