Passkey Signer Package

TLDR— We represent a new signer for account abstraction wallets or SDK for having a passkey-based login mechanism. The passkey allows users to use their device's primary authentication mechanisms like face id, touch id or passwords to create a wallet and sign a transaction.

Thanks to @nlok5923 and @0xjjpa for their work, reviews and discussions

Definition

According to the ethers documentation, a signer is:

"…an abstraction of an Ethereum Account, which can be used to sign messages and transactions and send signed transactions to the Ethereum Network to execute state-changing operations."

The PasskeySigner

package will extend the abstract signer provided by ethers and offer the functionality to sign transactions, messages, and typed messages for blockchains using passkeys. A passkey

is a digital credential tied to a user account and a website or application. Passkeys allow users to authenticate without entering a username or password or providing any additional authentication factor. This technology aims to replace legacy authentication mechanisms such as passwords

. Passkeys serve as a replacement for private key management, offering faster sign-ins, ease of use, and improved security.

Bundlers and EntryPoint

in this context refer to the same concepts mentioned in ERC 4337.

Advantages

There are several advantages:

- User Experience:

Onboarding new users to the blockchain is challenging, with seed phrases and private key management being less than ideal. We aim to address this by ensuring even users unfamiliar with security concerns can safely manage their funds.

- Security

: Passkeys provide inherent security by eliminating issues like weak and reused credentials, leaked credentials, and phishing.

- Plug and Play:

Simplify the integration of smart contract wallets by replacing the reliance on Externally Owned Accounts (EOA) for transaction and message signing. With the passkey module, wallet infrastructure and wallets can seamlessly integrate passkey functionality, streamlining the user experience.

- Cross-platform support

: Extend the solution to devices without biometric scanning but with Trusted Execution Environment (TEE) support. Utilizing QR code scanning, devices perform a secure local key agreement, establish proximity, and enable end-to-end encrypted communication. This ensures robust security against phishing attempts.

[

2718×1544 245 KB

](https://ethresear.ch/uploads/default/original/2X/6/64ba5c827a3dacec883ed4f6109d96032f1e2c27.jpeg)

Disadvantages

Gas cost

: On-chain signature verification for passkey-based transactions incurs significant gas costs. Efforts have been made to reduce the gas cost for verification. Opclave utilizes a custom rollup with the "secp256r1 verifier" precompile contract following Optimism's Bedrock Release standards. Ledger is also working on further optimizing gas costs.

Device dependency

: Though passkeys are device dependant there are workarounds. Apple device users can securely back up their passkeys in iCloud Keychain, overcoming this restriction. For other devices, the module will provide multi-device support, allowing users to add multiple owners (devices) to their smart contract wallet.

Usage of PasskeySigner

The initialization would be straight forward if the wallet sdk has itself has exposed necessary interfaces for accepting an external signers. For example let’s suppose if there’s a wallet sdk known as abc sdk. And let’s assume it accepts the signer while initializing it’s instance so in that case the integration would look like this

// importing BananaPasskeyEoaSigner package import { BananaPasskeyEoaSigner } from '@rize-labs/banana-passkey-manager'; import { ABCWallet } from 'ABCWallet-sdk';

// initializing jsonRpcProvider const provider = ethers.getDefaultProvider();

// creating an instance out of it const bananaEoaSignerInstance = new BananaPasskeyEoaSigner(provider);

// initializing the EOA with a specific username (it should be unique) corresponding to which the passkey // would be created and later on accessing await bananaEoaSignerInstance.init('');

// initializing signer for smart contract wallet. const abcSmartWalletInstance = new ABCWallet(bananaEoaSignerInstance);

Flow of making a transaction will look like this

[

1600×1468 132 KB

](https://ethresear.ch/uploads/default/original/2X/8/8c0fa887c74da3310af964e95ef03ced64043acb.jpeg)

Background

One of the notable features of smart contract wallets is the provision of custom signatures for transactions. Among the prominent signature schemes, secp256R1-based signatures stand out. It’s important to note that secp256k1 is a Koblitz curve, whereas secp256r1 is not. Koblitz curves are generally considered slightly weaker than other curves. Both secp256k1 and secp256r1 are elliptic curves defined over a field z_p

, where p represents a 256-bit prime (although each curve uses a different prime). Both curves adhere to the form $y^2 = x^3 + ax + b$

. In the case of the Koblitz curve, we have:a=0

and b=7

For R1 we have a = FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFC

and b = 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B

The R1 curve is considered more secure than K1 and supports major hardware enclaves. Also, most security enclaves cannot generate K1-based signatures, which are commonly used by EVM blockchains for signing transactions and messages.

Creating a new passkey

A typical ethers signer signs transactions by either having the private key itself or by being connected to a JsonRpcProvider to fetch the signer. However, the Passkey Signer operates differently as it does not possess the private key. Instead, it can sign transactions and messages by sending them to the hardware, and the signed string is provided as the output.

const publicKeyCredentialCreationOptions = { //The challenge is a buffer of cryptographically random bytes generated on the server, and is needed to prevent "replay attacks". challenge: Uint8Array.from(randomStringFromServer, c => c.charCodeAt(0)), rp: { name: "Your Name", id: "yourname.com", }, user: { id: Uint8Array.from("UZSL85T9AFC", c => c.charCodeAt(0)), name: "your@name.guide", displayName: "ABCD", }, //describe the cryptographic public key. -7 is for secp256R1 elliptical curve pubKeyCredParams: [{alg: -7, type: "public-key"}], authenticatorSelection: { authenticatorAttachment: "cross-platform", }, timeout: 60000, attestation: "direct" };

const credential = await navigator.credentials.create({ publicKey: publicKeyCredentialCreationOptions });

//credential object

PublicKeyCredential { id: 'ADSUllKQmbqdGtpu4sjseh4cg2TxSvrbcHDTBsv4NSSX9...', rawId: ArrayBuffer(59), response: AuthenticatorAttestationResponse { clientDataJSON: ArrayBuffer(121), attestationObject: ArrayBuffer(306), }, type: 'public-key' }

The public-key is extracted and that is passed to the smart contract wallet.

Signing using passkeys

During authentication an assertion

is created, which is proof that the user has possession of the private key.

const assertion = await navigator.credentials.get({ publicKey: publicKeyCredentialRequestOptions });

The publicKeyCredentialCreationOptions

object contains a number of required and optional fields that a server specifies to create a new credential for a user.

const publicKeyCredentialRequestOptions = { challenge: Uint8Array.from(randomStringFromServer, c => c.charCodeAt(0)), allowCredentials: [{ id: Uint8Array.from(credentialId, c => c.charCodeAt(0)), type: 'public-key', transports: ['usb', 'ble', 'nfc'], }], timeout: 60000, }

The interaction with the hardware will be done using the webauthn library which allows us to generate new cryptographic keys within the hardware. The public key which is an (x,y) co-ordinate, corresponding to this private key is fetched. These co-ordinates should be stored inside the smart contract wallets and they will act like a owner to the smart contract wallet.