Enabling Web2-Based User Authentication for Account Abstraction

Abstract—**In this demo, we describe the process of integrating typical Web2-based user authentication mechanisms into the ERC-4337 account abstraction (AA) and use the passkey-based authentication as an example to illustrate how to manage a smart contract wallet (SCW) using a passkey.**

I. Introduction

The emergence of Web3, i.e., the third iteration of the internet, represents a major paradigm shift that could fundamentally change the way we interact, transact and collaborate online. While Web3 has great potential to realize a decentralized internet that is open, trustless and permissionless, it is facing challenges of massive adoption due to a cumbersome user experience (UX). The account abstraction specified in ERC-4337 [1] is poised to fundamentally address the UX challenges in Web3 by allowing users to use smart contract wallets (SCWs) instead of Externally Owned Accounts (EOAs) as their primary accounts.

At its core, AA redefines how users interact with blockchain by abstracting the complexities of transactions, namely signature verification, nonce increase, gas payment and chain compatibility. The introduction of SCWs that allow arbitrary verification logic makes interaction with blockchain more intuitive for the average user. In particular, signature abstraction in AA enables a user to utilize a Web2-based authentication mechanism (e.g., password, OpenID Connect (OIDC) [2], passkey [3]) he/she is familiar with, which significantly lower the barrier to using Web3 applications. In this demo, we describe the integration architecture of Web2-based authentication in AA and demonstrate how a user can manage his/her SCW using a passkey.

II. An Overview of ERC-4337 Account Abstraction

A. Key Components

The ERC-4337 account abstraction consists of the following six key components:

- **UserOperation** (UserOp): The UserOp represents a user's desired transaction intent and contains all transaction-related information (e.g., sender, nonce, initCode, calldata, signature, etc.) submitted by the user.
- **Mempool**: The UserOperation objects are sent to a dedicated high-level mempool that could be public or private.
- **Bundler**: A bundler is a new type of node that listens to the UserOp mempool, bundles multiple UserOperations into a single transaction, and sends it to the EntryPoint contract for execution.
- **EntryPoint**: The EntryPoint contract is a singleton contract that validates and executes the bundled UserOperations sent to it. The EntryPoint contract is the global entry point for all ERC-4337 compliant smart contract wallets.
- **Paymaster (Optional)**: A Paymaster deals with the implementations of flexible gas payment policies (e.g., sponsored transactions, gas payment in other ERC-20 tokens or off-chain payment methods, etc.).
- **Aggregator (Optional)**: An aggregator utilizes a helper smart contract to combine multiple signatures from UserOperations into a single one, thereby improving transaction costs and efficiency.

B. Workflow

A high-level workflow of ERC-4337 account abstraction is shown in Fig. 1.

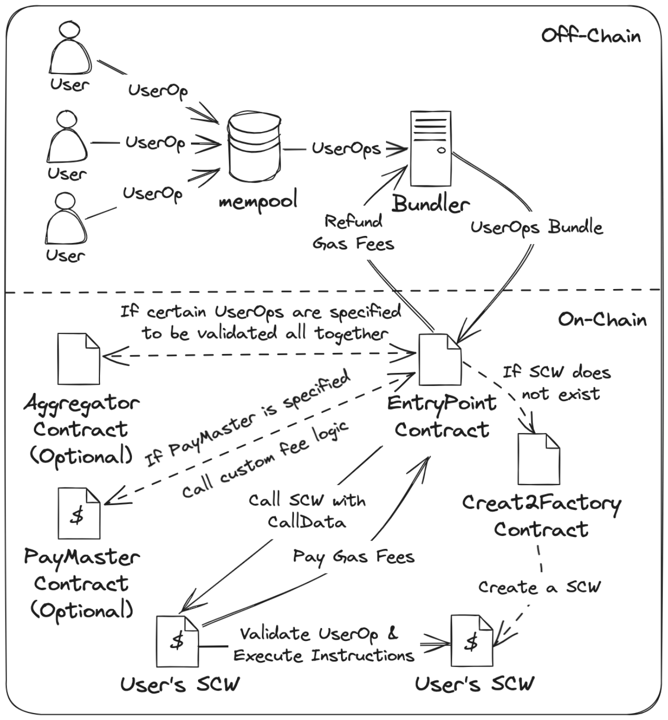


Fig. 1. A high-level workflow of ERC-4337 account abstraction

All the users first submit their UserOperation objects to a dedicated mempool. A bundler that monitors the mempool bundles multiple UserOperations together and sends it the EntryPoint contract. All the UserOperations in the bundle are processed by the EntryPoint contract in an iterative manner. If certain UserOperations are specified to be validated all together, the EntryPoint contact will aggregate multiple signatures with the aid of the Aggregator contract. In the case that a user's SCM does not exist, the system will create one automatically using the CREATE2 opcode in the Create2Factory contract. If a PayMaster is specified in a UserOperation, the EntryPoint contract will check whether the PayMaster has sufficient stake and agrees to pay for the UserOperation in question. For a UserOperation that does not specify Aggregator and Paymaster, the user's SCW validates the signature and nonce, pays gas fees, and executes the instructions (e.g., token transfer to another SCW) in the UserOperation. Finally, the EntryPoint contract refunds the gas fees to the Bundler. If a PayMaster is used, the EntryPoint contract will call the custom fee logic (e.g., pay gas fees in other ERC-20 tokens) in PayMaster to collect gas fees.

III. Integration with Web2-Based Authentication

One of the salient features of ERC-4337 AA is the signature abstraction. This feature allows a SCW to use any verification logic, including various signature schemes, social recovery, time locks, withdrawal limits, etc. In particular, signature abstraction opens the door for integration of conventional Web2-based user authentication mechanisms (e.g., password, OpenID Connect (OIDC), passkey, etc.) into AA. Fig. 2 illustrates the architecture for integrating typical Web2-based authentication methods in AA:

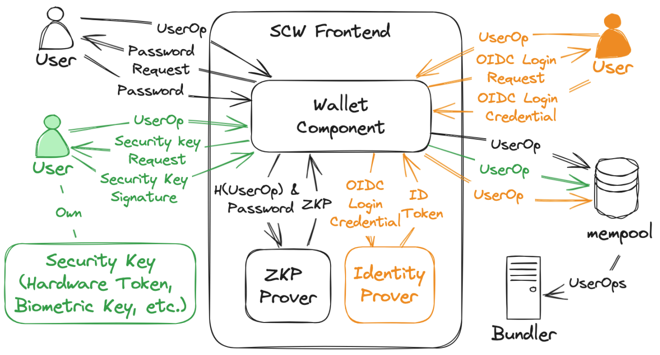


Fig. 2. The integration architecture of Web2-based authentication in AA

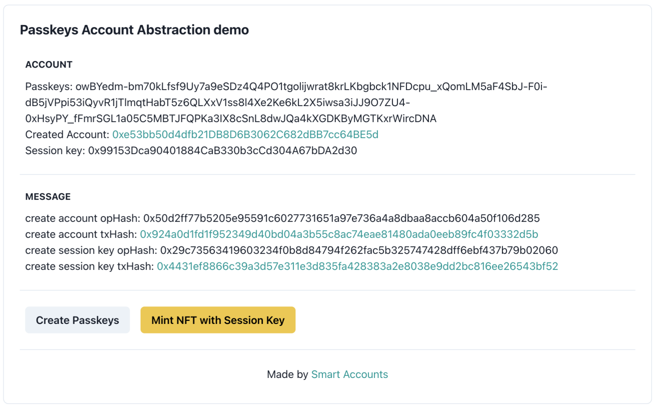
- *Password-based authentication* (marked in black): The frontend needs to generate a zero-knowledge proof (ZKP) that could be verified on-chain without leaking the password;
- *Passkey-based authentication* (marked in green): The frontend needs to request a security key to generate a signature that is then verifed inside a SCW;
- *OIDC-based authentication* (marked in orange): The frontend needs to request an ID token that is then verified inside a SCW;

The basic idea for such an integration is to intercept a UserOp with a SCW frontend (e.g., a web browser) and request a Web2-based authentication method to provide an identity proof that will be filled in the "signature" field of a UserOperation object. The rest of the workflow is the same as that of the ERC-4337 AA. Based on the chosen Web2-based authentication approach, a user's SCW should implement the corresponding verification logic.

IV. Passkey-Based Account Abstraction in Action

We implemented a passkey-based AA based on the integration architecture depicted in Fig. 2 and a demo website is available at https://passkeys-wallet.onrender.com/.

As shown in Fig. 3, a user can easily create a passkey-based SCW after a passkey is generated on a mobile device or USB token. Note that the passkey is only used locally to create a digital signature that can be verified inside a SCW. Due to the high gas cost of verifying an ECDSA signature over secp256r1, we further introduces session keys that are randomly generated private/public key pairs over secp256k1[1]. Each session key can be used for a pre-defined period of time (e.g., one hour) for a passkey-bounded SCW and a new session key needs to be generated once the user closes the browser or the current session key is expired.


Passkeys Account Abstraction demo

ACCOUNT

Passkeys: owBYedm-bm70kLfsf9Uy7a9eSDz4Q4PO1tgolijwrat8krLKbgbck1NFDcpu_xQomLM5aF4SbJ-F0i-dB5jVPpi53iQyvR1jTlmqtHabT5z6QLXxV1ss8l4Xe2Ke6kL2X5iwsa3iJJ9O7ZU4-0xHsyPY_fFmrSGL1a05C5MBTJFQPKa3IX8cSnL8dwJQa4kXGDKByMGTKxrWircDNA
Created Account: 0xe53bb50d4dfb21DB8D6B3062C682dBB7cc64BE5d
Session key: 0x99153Dca90401884CaB330b3cCd304A67bDA2d30

MESSAGE

create account opHash: 0x50d2ff77b5205e95591c6027731651a97e736a4a8dbaa8accb604a50f106d285
create account txHash: 0x924a0d1fd1f952349d40bd04a3b55c8ac74eae81480ada0eeb89fc4f03332d5b
create session key opHash: 0x29c73563419603234f0b8d84794f262fac5b325747428dff6ebf437b79b02060
create session key txHash: 0x4431ef8866c39a3d57e311e3d835fa428383a2e8038e9dd2bc816ee26543bf52

Made by Smart Accounts

Fig. 3. The creation of a passkey-based SCW and session key

During the pre-defined period of time, the user can use the session key to conduct consecutive operations (see https://testnet.iotexscan.io/tx/0x163aa30b8b276f9f77629c4c44c1fdd1441a54c449838cca55b98096742dd6c5 for the transaction of minting an NFT using the session key) on his/her SCW.

References

[1] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, and T. Hess, ERC-4337: Account Abstraction Using Alt Mempool, https://eips.ethereum.org/EIPS/eip-4337.
[2] OpenID Foundation, OpenID Connect Core 1.0. https://openid.net/specs/openid-connect-core-1_0.html.
[3] FIDO Alliance, Passkeys, https://fidoalliance.org/passkeys/.

[1]The gas cost of verifying an ECDSA signature over secp256k1 is significantly less than that over secp256r1.