invozone

InvoDocs

Ethereum BlockChain

InvoBlox Account Abstraction

Prepared By: Muhammad Talha

InvoDocs

May 3, 2023

1._ Account Abstraction

1.1_Introduction:

Account abstraction in a blockchain system refers to separating the control of an account from its associated private key. This allows an account to be controlled by a smart contract rather than simply a private key.

For example, Ethereum account abstraction may work in a way so as to help you recover your wallet account when you lose your password. For experienced users, the new feature could offer more control and flexibility to their accounts.

To understand account abstraction we first need to understand how accounts work on Ethereum today.

There are two types of accounts on Ethereum:

- **Externally Owned Accounts (EOA)**
- **Contract Accounts (CA).**

1.2_Externally Owned Accounts (EOA)

As the name indicates, EOAs are accounts owned by something external to the blockchain - namely, users.

EOAs have three properties:

- A balance to represent the amount of ETH available to the account.
- A nonce to ensure that every transaction is unique.
- An address to uniquely identify the account on the network.

The state of the blockchain, hence the state of an account, can only be modified through transactions. This trigger must come from something external to the blockchain, hence on Ethereum, every transaction must be initiated from an EOA. That means that when a transaction is executed by the Ethereum Virtual Machine (EVM) the first account being touched must be an EOA and the corresponding account must pay a fee to the miner for the execution of the entire transaction.

An account on Ethereum is made of three components:

- A state containing a balance and a nonce.
- Hardcoded logic in the EVM to validate and execute a transaction from the account.
- An address.

The last point highlights a very important design choice of the Ethereum blockchain: the concept of Account (the object holding your tokens) and the concept of Signer (the object authorized to move these tokens) are basically the same thing! If you have a private key you automatically have an account at the associated address, and to own an account at a given address you must be in possession of the corresponding private key. That logic is hardcoded at the heart of the EVM.

“Because your private key is your account, losing your key means losing your account.”

Even worse, if someone else has your private key they also have your account... and all the tokens it contains. And there is nothing you can do about it!

1.3_Contract Accounts (CA)

Account abstraction enables smart contracts to initiate transactions themselves, so that any logic that the user wishes to implement can be coded into the smart contract wallet itself and executed on Ethereum.Let’s decouple the object holding your tokens (the account) from the object authorized to move these tokens (the signer).

In simpler terms, it means separating the thing that holds your digital assets (like money or cryptocurrency), which is called your account, from the thing that has permission to use or move those assets, which is called the signer. By doing this, you can give permission to other signers to move your assets without giving them access to your entire account. It's a way to make your assets more secure and easier to manage.

Turn accounts into smart contracts with their own logic to define what a valid transaction is. The only requirement is that they comply with a specific interface with methods to validate and execute transactions.

In computer science lingo we say that the account has been abstracted, hence the term: **account abstraction**.

And we can immediately see why it's so powerful. It’s no longer one-account-fits-all-use-cases. Instead, each user can have an account that is adapted to their needs.

- Do you want to use a different signing scheme than ECDSA? No problem, you can write an account for that.
- Do you want to use multiple keys to authorize transactions? No problem, you can write an account for that.
- Do you want to change the signer of your account every week? No problem, you can write an account for that.

1.4_Externally Owned Accounts VS Contract Accounts.

Use Case	EOAs	Contract Accounts
Permission controls	Private key grants full access to everything.	Define a list of tiered permission levels. e.g. Require 3 out of 5 signers to approve a transaction.
Batch transactions	Each individual action requires a separate signature.	Capable of batching transactions together; e.g. approving a token transfer and transferring a token in the same operation.
Account Recovery	Loss or exposure of a private key means full loss of control over the wallet.	There is no private key. You can write any arbitrary logic in code that allows you to recover the funds in the wallet.
Transaction limits	Any transaction your wallet signs is what occurs. You cannot restrict anything.	Write any logic to control how funds can be transferred. E.g. a function to halt transactions to other addresses while you recover your account.

Account abstraction is a way to separate an Ethereum address from an Ethereum contract. Up until now, when a smart contract was created, it was associated with an Ethereum address. This means that any interaction with the contract must go through that address. Account abstraction, on the other hand, allows for the creation of contracts that are not tied to a specific address, making it easier to develop more complex contracts and reducing gas costs.

2._ Examples of security logic that can be built into a smart contract wallet:

2.1_ Multicall - for 1 tap crypto

If you're using a Dapp on Ethereum today, you have to make a new transaction for every on-chain interaction. This is frustrating, time consuming and expensive when gas fees are high.

With Account Abstraction, you can instead bundle multiple transactions into one, and execute the sequence of operations in one atomic transaction. This feature is called multicall.

As an example, providing liquidity to Uniswap usually requires three transactions: approving each of the two tokens, then depositing them. With multicall, you can do it in just one atomic transaction. Quicker. Easier. And more secure (as you remove the need for an infinite approval).

2.2_ Session keys - for simplicity and security

Session keys are a breakthrough for UX, particularly for blockchain games. They allow you to pre-approve the rules for interacting with a Dapp so you can use it as much as you want within those rules without having to sign every single transaction.

2.3_ Social recovery - for security & the end of seed phrases

The goal of social recovery is to protect people if they lose their account or it's somehow compromised. Social recovery does this while avoiding seed phrases, the typical recovery method for wallets such as MetaMask. Seed phrases need to be eliminated as they're hard to use, insecure, and a major barrier to mass adoption. With social recovery, if you lose your private key you can just authorize a new key as the legitimate wallet owner. The mechanisms for this can vary. You could choose a recovery method that relies on your trusted contacts, your hardware wallet(s), or even a third party service. Or a combination of them all.

Importantly, social recovery does not sacrifice self-custody. You remain in control of your assets. And for further protection you can use time delays so that you have an opportunity to cancel the recovery if you wish.

2.4_ Multi-factor authentication & enhanced security

Account Abstraction lets you have accounts that require signatures from multiple keys, with a transaction only going ahead if certain conditions are met. Account Abstraction allows you to tailor your account's security levels to meet your needs and use a variety of different devices to approve transactions.

- Have two (or more) factor authentication for crypto. Imagine that one of the keys for your account is managed by a service that will only co-sign if you've confirmed with a second factor like email or SMS. If you confirm the second factor, the transaction succeeds. If you don't, it's automatically blocked.

- Keep a list of scam addresses and automatically block transactions to them. You could also block transactions to an incorrect contract.

- Set a daily transfer limit and automatically block anything above it (unless you explicitly approve it to go through).

- Integrate off chain services for additional protection. For example, use a security service to check an NFT collection is verified on OpenSea before approving a transaction, asking for two-factor authentication if not.

2.5_ Plug-ins - for greater flexibility

Plug-ins make an account more flexible and modular. Third party developers can build plug-ins with new functionalities they want to enable when creating their account. You can also make an account extendable by letting users add or remove functionalities after the account has been created.

You can almost think of it as an app store for your account - choosing a plug-in for gaming, social recovery, session keys or more.

3._ Additional benefits of Account Abstraction

3.1_ Pay fees in any token

The status quo in Ethereum is that you need to have a certain token to pay a gas fee. This can be a burden for users as you need to make a transfer or trade to get the right one. With Account Abstraction though, you can pay gas fees in any token.

3.2_ Projects can pay fees on behalf of people

Projects can act as ‘paymasters’ paying the gas fees on behalf of users. This significantly reduces the friction for users.

3.3_ Different signing schemes

Ethereum today relies on a single signing scheme - ECDSA - and one elliptic curve. Importantly, the only way to change this is with Account Abstraction.

3.4_ Make every phone a hardware wallet

A different signature scheme could let you use the secure enclave of iOS and Android devices to turn smartphones into hardware wallets. With over 6 billion smartphone users globally, this change would make today's hardware wallet market look vanishingly small.

3.5_ Quantum resistance

We know that quantum computers are coming and can break ECDSA. As Vitalik has written, with Account Abstraction you could explore "post-quantum safe signature algorithms (e.g. Lamport, Winternitz)".

3.6_ More efficient signature algorithms

This could lead to lower gas fees by executing fewer computation steps when verifying a transaction signature in the smart contract wallet.

3.7_ Upgradeability

Because accounts are contracts, they can use the well known proxy pattern and delegate the execution to an implementation. If the proxy is programmed to be upgradable, users can upgrade the code of their account as new features become available.

4._ What Does ERC-4337 Mean For Users?

ERC-4337 could spell the end for the complicated crypto wallet user experience and, in doing so, could also increase adoption. Here are some highlights of what ERC-4337 could enable:

4.1_ Wallet setup:

No need to write down seed phrases. Setup can be quick and easy with just a few clicks.

4.2_ Worry-free account recovery:

Users no longer need to sweat over losing their seed phrases, as multi-factor authentication and account recovery are now possible.

4.3_ User-friendly wallet functions:

Users can enjoy a wide range of customized services including auto-pay, pre-approve transactions, and bundled transactions. The sky's the limit.

4.4_ Better security:

Wallets could potentially be more secure as the possibility of human error is reduced — no more hiding seed phrases under your mattress! ERC-4337 should, in theory, lead to a smoother and friendlier user experience for users, therefore removing one major hurdle for mass adoption.

4.5_ Gas flexibility:

Wallets powered by ERC-4337 can now pay gas fees with any ERC-20 tokens and beyond. Developers can build wallets that make paying gas fees in any tokens and even fiat possible.

5._How does this proposal work?

In the study of the proposal's architecture, its components include; EntryPoint contract, paymaster contract, UserOperation, Bundler, Miner, and client library.Account abstraction," enables the use of cryptographic keys for cryptocurrency to be stored on standard smartphone security modules and also enables two-factor authentication, monthly spending limits, and social recovery of accounts.

Users are expected to send off-chain messages called user operations.A UserOperation is not a transaction; it is a structure that describes a transaction that a user wants to be sent on their behalf. Users send UserOperation objects into a dedicated user operations mempool.There exists a special category of actors called a Bundler, also termed a node or a block builder, who listens in on the user operation mempool, and bundles multiple UserOperations into a transaction.A bundle transaction packs up multiple UserOperation objects into a single handleOps, and creates an EntryPoint call to the contract, before the transaction is included in a block.The proposer or builder is responsible for filtering the operations to ensure that they only accept operations that pay fees. There is a separate mempool for user operations, and nodes connected to this mempool do ERC-4337-specific validations to ensure that a user operation is guaranteed to pay fees before forwarding it.

By using a signature scheme that includes the chainId and EntryPoint address, we can prevent replay attacks and ensure that transactions are only valid on the intended chain. This helps to keep transactions secure and prevent fraud.

5.1_Specification:

Field	Type	Description
Sender	Address	The account making the operation.
Nonce	uint256	Anti-replay parameter; also used as the salt for first-time account creation.
initCode	bytes	The initCode of the account (needed if and only if the account is not yet on-chain and needs to be created).
callData	bytes	The data to pass to the sender during the main execution call.
callGasLimit	uint256	The amount of gas to allocate the main execution call.
verificationGasLimit	uint256	The amount of gas to allocate for the verification step.
preVerificationGas	uint256	The amount of gas to pay for to compensate the bundler for pre-verification execution and calldata.
maxFeePerGas	uint256	Maximum fee per gas (similar to EIP-1559 max_fee_per_gas).
maxPriorityFeePerGas	uint256	Maximum priority fee per gas (similar to EIP-1559 max_priority_fee_per_gas).
paymasterAndData	bytes	Address of paymaster sponsoring the transaction, followed by extra data to send to the paymaster (empty for self-sponsored transaction).
signature	bytes	Data passed into the account along with the nonce during the verification step.

5.2_ UserOperation

A UserOperation looks like a transaction; it's an ABI-encoded struct that includes fields such as:

- UserOperation - a structure that describes a transaction to be sent on behalf of a user. To avoid confusion, it is not named "transaction".
    - Like a transaction, it contains "sender", "to", "calldata", "maxFeePerGas", "maxPriorityFee", "signature", "nonce".
    - unlike a transaction, it contains several other fields, described below.
    - also, the "nonce" and "signature" fields usage is not defined by the protocol, but by each account implementation.

- EntryPoint - a singleton contract to execute bundles of UserOperations. Bundlers/Clients whitelist the supported entrypoint.
- Bundler - a node (block builder) that bundles multiple UserOperations and create an EntryPoint.handleOps() transaction. Note that not all block-builders on the network are required to be bundlers.
- Aggregator - a helper contract trusted by accounts to validate an aggregated signature. Bundlers/Clients whitelist the supported aggregators.

6._ Contracts Info(Working Flow)

A wallet is a smart contract, and is required to have these functions:

- validateUserOp, which takes a UserOperation as input. This function is supposed to verify the signature and nonce on the UserOperation, pay the fee and increment the nonce if verification succeeds, and throw an exception if verification fails.

- An op execution function,that allows a wallet to understand what actions to take based on the instructions it receives. These instructions are in the form of "calldata". The function can interpret these instructions in many different ways, but usually we expect most common behavior would be to parse the calldata as an instruction for the wallet to make one or more calls.

- In order to keep a wallet safe, some complicated code is needed. But instead of putting that code directly in the wallet, it's put in a separate "entry point" contract. The entry point checks to make sure that only authorized actions can be taken on the wallet.When someone wants to do something with the wallet, like sending money or changing settings, they have to go through the entry point. The validateUserOp and execution functions are expected to be gated with require(msg.sender == ENTRY_POINT), so only the trusted entry point can cause a wallet to perform any actions or pay fees.The entry point is also responsible for creating new wallets when they're needed. It uses a special code to create a new wallet, and makes sure that everything is set up correctly so that the wallet is safe to use.

6.1_Sponsorship with paymasters

Sponsored transactions have a number of key use cases. The most commonly cited desired use cases are:

- Allowing application developers to pay fees on behalf of their users.
- Allowing users to pay fees in ERC20 tokens, with a contract serving as an intermediary to collect the ERC20s and pay in ETH.

This proposal can support this functionality through a built-in paymaster mechanism. A UserOperation can set another address as its paymaster. If the paymaster is set (ie. nonzero), during the verification step the entry point also calls the paymaster to verify that the paymaster is willing to pay for the UserOperation. If it is, then fees are taken out of the paymaster's ETH staked inside the entry point (with a withdrawal delay for security) instead of the wallet. During the execution step, the wallet is called with the calldata in the UserOperation as normal, but after that the paymaster is called with postOp.

Example workflows for the above two use cases are:

- The paymaster verifies that the paymasterData contains a signature from the sponsor, verifying that the sponsor is willing to pay for the UserOperation. If the signature is valid, the paymaster accepts and the fees for the UserOperation get paid out of the sponsor's stake.
- The paymaster verifies that the sender wallet has enough ERC20 balance available to pay for the UserOperation. If it does, the paymaster accepts and pays the ETH fees, and then claims the ERC20 tokens as compensation in the postOp (if the postOp fails because the UserOperation drained the ERC20 balance, the execution will revert and postOp will get called again, so the paymaster always gets paid). Note that for now, this can only be done if the ERC20 is a wrapper token managed by the paymaster itself.