docs

Bundlers

Introduction

**Introduction**

A bundler is the core infrastructure component that allows account abstraction to work on any EVM network without requiring any changes to the protocol. Its purpose is to work with a new mempool of UserOperations and get the transaction included on-chain.

**Security considerations**

When reading the ERC specs, you'll notice that there are many rules a bundler must follow. Each of these rules has been extensively debated and discussed by security researchers and builders within the Ethereum ecosystem.

One of the bundler's main jobs is to comply with these rules to prevent all possible DoS attack vectors. These include everything from basic sanity checks that make sure a UserOperation is structurally sound to more in-depth tracing for banned opcodes and storage access to make sure bundles cannot be censored once submitted to the network.

Similar to Ethereum clients, all bundler implementations are expected to pass a test suite to ensure their compliance and that they won't fragment the mempool. Yoav Weiss wrote [an article(opens in a new tab)](https://notes.ethereum.org/@yoav/unified-erc-4337-mempool) explaining the importance of this point.

**UserOperation mempool**

The canonical mempool for EIP-4337 is decentralized and is made up of a permissionless P2P network of independent bundlers.

docs

Bundlers

Build

**Building a bundler**

If you want to build your own bundler, it's crucial that it passes all the tests covered in our test suite. More on this in the [testing a bundler](https://www.erc4337.io/docs/bundlers/testing-a-bundler) section. A good reference point to start with is our basic implementations of a bundler. This bundler focuses on being compliant, not on being fast.

**Infinitism reference bundler**

The repo for this bundler can be found here: [https://github.com/eth-infinitism/bundler(opens in a new tab)](https://github.com/eth-infinitism/bundler)

This is a reference implementation for a bundler, implementing the full EIP-4337 RPC calls (both production and debug calls), required to pass the [bundler-spec-tests(opens in a new tab)](https://github.com/eth-infinitism/bundler-spec-tests) test suite.

To build this bundler on your machine, please follow the instruction of the [repository README](https://github.com/eth-infinitism/bundler#readme)

docs

Bundlers

Run

**Verifying UserOperation validity**

When a bundler receives a UserOperation, it must first run some basic sanity checks, namely that:

* Either the sender is an existing contract, or the initCode is not empty (**but not both**)
* If initCode is not empty, parse its first 20 bytes as a factory address. Record whether the factory is staked, in case the later simulation indicates that it needs to be. If the factory accesses global state, it must be staked - see [reputation, throttling and banning](https://www.erc4337.io/docs/faqs/reputation-throttling-banning.md) section for details.
* The verificationGasLimit is sufficiently low (<= MAX\_VERIFICATION\_GAS) and the preVerificationGas is sufficiently high (enough to pay for the calldata gas cost of serializing the UserOperation plus PRE\_VERIFICATION\_OVERHEAD\_GAS)
* The paymasterAndData is either empty, or start with the **paymaster** address, which is a contract that (i) currently has nonempty code on chain, (ii) has a sufficient deposit to pay for the UserOperation, and (iii) is not currently banned. During simulation, the paymaster's stake is also checked, depending on its storage usage - see [reputation, throttling and banning](https://www.erc4337.io/docs/faqs/reputation-throttling-banning.md) section for details.
* The callgas is at least the cost of a CALL with non-zero value. The maxFeePerGas and maxPriorityFeePerGas are above a configurable minimum value that the bundler is willing to accept. At the minimum, they are sufficiently high to be included with the current block.basefee. The sender doesn't have another UserOperation already present in the pool (or it replaces an existing entry with the same sender and nonce, with a higher maxPriorityFeePerGas and an equally increased maxFeePerGas). Only one UserOperation per sender may be included in a single batch. A sender is exempt from this rule and may have multiple UserOperations in the pool and in a batch if it is staked (see [reputation, throttling and banning](https://www.erc4337.io/docs/faqs/reputation-throttling-banning.md) section), but this exception is of limited use to normal accounts.

If the UserOperation object passes these sanity checks, the bundler must next run the first op simulation, and if the simulation succeeds, the bundler must add the op to the pool. A second simulation must also happen during bundling to make sure the UserOperation is still valid.

**Simulation**

In order to add a UserOperation into the UserOp mempool (and later to add it into a bundle) we need to "simulate" its validation to make sure it is valid, and that it is capable of paying for its own execution. In addition, we need to verify that the same will hold true when executed on-chain. For this purpose, a UserOperation is not allowed to access any information that might change between simulation and execution, such as current block time, number, hash etc.

In addition, a UserOperation is only allowed to access data related to this sender address: Multiple UserOperations should not access the same storage, so that it is impossible to invalidate a large number of UserOperations with a single state change.

There are 3 special contracts that interact with the account: the factory (initCode) that deploys the contract, the paymaster that can pay for the gas, and signature aggregator. Each of these contracts is also restricted in its storage access, to make sure UserOperation validations are isolated.

**Storage associated with an address**

We define storage slots as "associated with an address" as all the slots that uniquely related on this address, and cannot be related with any other address. In solidity, this includes all storage of the contract itself, and any storage of other contracts that use this contract address as a mapping key.

An address A is associated with:

1. Slots of contract A address itself.
2. Slot A on any other address.
3. Slots of type keccak256(A || X) + n on any other address. (to cover mapping(address => value), which is usually used for balance in ERC-20 tokens). n is an offset value up to 128, to allow accessing fields in the format mapping(address => struct)