OpenZeppelin | security

Account Abstraction Audit

ethereum foundation

February 20, 2024

Table of Contents

Summary

Type	Account Abstraction
Timeline	From 2024-01-15 To 2024-01-19
Languages	Solidity

Total Issues	24 (24 resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	5 (5 resolved)
Low Severity Issues	8 (8 resolved)
Notes & Additional Information	8 (8 resolved)
Client Reported	3 (3 resolved)

Scope

We audited the eth-infinitism/account-abstraction repository at commit 9879c93.

In scope were all non-test Solidity files that were changed since our last review commit, excluding the `LegacyTokenPaymaster` contract. As with our previous audits, we included the ERC specifications. We also reviewed pull request #396 and pull request #447.

***Update**: As part of the fix review process and our review of these changes, we reviewed the pull requests that affect in-scope contracts up to commit `8086e7b`.*

System Overview

The system architecture is described in our previous audit reports (1, 2). Here we only describe the relevant changes.

New Validation Rules

The off-chain validation requirements have been moved to ERC-7562 to better encapsulate, classify and explain the rules and their rationale.

The ruleset has also been refined to limit the attack surface (for example, by banning unassigned opcodes) and to attribute account validation failures to a staked factory, where applicable. The most notable change is that staked entities can now read the storage of external contracts, but the reputation system will still restrict the scope of mass invalidations.

New Gas Accounting Structure

Previously, the verification gas limit covered the validation step for both the account and paymaster. The same value was also independently used to limit the paymaster's `postOp` call. Now, there is a new limit specifically for paymaster validation and another one for the paymaster `postOp` call. This provides users with more fine-grained control over the operation parameters and simplifies the `EntryPoint` code.

Remove Second Paymaster `postOp` Call

The possibility of a second paymaster `postOp` call was removed to ensure validated operations cannot revert. The paymaster's `postOp` will only be executed once in the same call frame as the operation's execution. As before, if this `postOp` call reverts, the user operation will revert as well. Either way, the paymaster will be required to pay for the gas costs. However, depending on the paymaster logic, the user might be charged during validation.

Simulation Code Is Moved Off-Chain

Previously, the `EntryPoint` contained several functions that were never intended to be called on-chain (and, in fact, always reverted). These were used to help bundlers simulate user operation validations and executions, profile gas usage, and detect forbidden behavior. These functions have been moved to an `EntryPointSimulations` contract which cannot be deployed on-chain but can be used during simulations. This simplifies the `EntryPoint` code and allows for easy upgrades of the simulation logic.

`EntryPoint` Supports ERC-165

The `EntryPoint` contract advertises its functionality using the ERC-165 protocol. In this way, other participants can validate that they are interacting with a compatible version.

Accounts Can Receive the Whole User Operation During Execution

User operations can optionally indicate that the account should receive the whole user operation during execution (including gas limits, nonce, paymaster data, etc). Standard operations will continue to invoke the account with the specified `callData`.

Unused Execution Gas Penalty

Whenever a user operation consumes less than its maximum allowed execution gas (including the paymaster's `postOp` call), it will only be refunded 90% of the difference. The remaining 10% is given to the bundler as a penalty. This discourages user operations from specifying very large gas limits that consume space in the bundle, preventing other operations from being included.

The Paymaster `postOp` Receives the Operation's Gas Price

The paymaster now receives the gas price that the operation is charged in its `postOp` function. This is simply a convenience to provide the paymaster with more context.

New `TokenPaymaster`

There is a new `TokenPaymaster`. It allows user operations to pay their gas costs with an ERC-20 token at a slightly inflated rate. During validation, the paymaster retrieves excess ERC-20 tokens from the account and then refunds the unused amount in its `postOp` call, where the oracle is queried. When its deposit with the `EntryPoint` falls low enough, it sells the ERC-20 tokens on Uniswap to top up its balance.

Other Changes

There were several other minor changes to the codebase that did not change core functionality.

Security Model and Trust Assumptions

Architecture

The main architectural change is to remove the paymaster's second `postOp` call. This was originally intended to provide the paymaster with a second chance to perform necessary cleanup (including retrieving funds from the account) if the account's operation behaves unexpectedly or maliciously. Now, the paymaster should either guarantee that it is compensated during the validation step or otherwise ensure that its first and only `postOp` call does not fail, regardless of how the user operation behaves. Similarly, users should ensure that they only specify paymasters that will not fail in the `postOp` call.

Integrations

The `TokenPaymaster` integrates with Chainlink oracles to determine the token price, and with Uniswap to exchange the tokens for ETH. As such, it assumes that both systems are functioning correctly. In particular, if the oracle stops updating for any reason (including due to being paused by the Chainlink multisig), the `TokenPaymaster` will become unusable until it is reconfigured with a live price feed. In the meantime, operations that use this paymaster will still pay fees in the specified token, but the operation will be reverted.

Privileged Roles

The `TokenPaymaster` has an owner address that can manage its stake with the `EntryPoint` contract. It can also change the paymaster's configuration at will, including safety parameters and the markup percentage it charges accounts. If these are changed unexpectedly, possibly by front-running user operations, an account may be overcharged or have its operation fail. Therefore, users should trust the owner address to choose these parameters safely and fairly.

Medium Severity

M-01 Operations Can Throttle Paymasters [core]

Note: this error was present in the previous audit commit but was not identified by the auditors at the time.

The `simulateValidation` function of `EntryPointSimulations` combines the validity conditions of the account and the paymaster to determine when the operation is considered valid by both parties. However, the `aggregator` parameter is semantically overloaded, to represent either a signature success flag or an aggregator address, and this is not completely handled in the combination.

Specifically, the combined `aggregator` is either the value chosen by the account, or if that is zero, the value chosen by the paymaster. This leads to two possible mistakes:

- an account's "signature success" flag (`0`) would be overwritten by a paymaster's non-zero aggregator parameter.
- a paymaster's "signature failed" flag (`1`) would be ignored in the presence of an account's non-zero aggregator.

The first condition would be identified by the rest of the security architecture and likely has minimal consequences. However, the second condition would cause bundlers to include unauthorized operations in a bundle and then blame the paymaster for the failure. This would cause paymasters to be unfairly throttled.

Consider updating the simulation to require paymasters to only return `0` or `1` as the `aggregator` parameter, and to ensure that the "signature failed" flag (`1`) always takes precedence.

Update: *Resolved in pull request #406. The Ethereum Foundation team stated:*

> *Remove the simulateValidation code to intersect the paymaster and account time-ranges (and signature validation). This calculation should be done off-chain by the bundler. The simulation functions now return the "raw" validationData, as returned by the account and paymaster (separately).*

M-02 Ineffective Unused Gas Penalty [core]

Operations specify several gas limits for each part of the lifecycle. The `callGasLimit` and `paymasterPostOpGasLimit` can be collectively denoted the "execution gas limit". This is because these values are related to the operation's execution and the amount of gas consumed in this phase cannot be reliably estimated during simulation.

Once an operation is executed, any unused execution gas is partially confiscated, and given to the bundler. This discourages operations that specify a much larger execution gas limit than they need, which consumes excess space in the bundle and prevents other valid operations from being included.

However, the `paymasterPostOpGasLimit` is not penalized when the `context` is empty. Although this corresponds to a situation where the `postOp` function is not called, the gas limit is still reserved, so it still consumes space in the bundle.

Consider always including the `paymasterPostOpGasLimit` in the penalty calculation.

Update: *Resolved in pull request #407.*

M-03 Unattributable Paymaster Fault [core]

Note: this error was present in the previous audit commit but was not identified by the auditors at the time.

The paymaster's `validatePaymasterUserOp` function is executed inside a `try-catch` block to trap any errors. However, this does not catch errors that occur when decoding the return values. This means that a malicious paymaster could provide an incompatible return buffer (e.g., by making it too short) in order to trigger a revert in the main call frame.

Importantly, this bypasses the `FailedOpWithRevert` error. This means that if a paymaster passes the operation simulations and triggers this error inside a bundle simulation, the bundler cannot use the revert message to immediately identify the malicious paymaster, and will spend excessive computation to construct a valid bundle.

Fortunately, as the Ethereum Foundation pointed out to us, bundlers are now expected to use `debug_traceCall`, and could identify the failing operation from the trace.

Consider using a low-level call and explicitly checking the size of the return data. Moreover, consider using this pattern generally with the other `try-catch` blocks that require return values to be decoded.

Update: *Resolved in pull request #429. The Ethereum Foundation team stated:*

> *The paymaster (and account) can use assembly code to create a response that is un-parseable by solidity, and thus cause a revert that can't be caught by solidity's try/catch and thus can't be mapped to a FailedOp revert reason. Instead of performing low-level call using `abi.encodeCall`, and later decoding the response manually using assembly code, we require the bundler to use the traceCall of the bundle (which is already mandatory), and find the root cause of the revert, as the last called entity just prior to this revert.*

M-04 Inconsistent Price Precision [samples]

The `TokenPaymaster` appears to be in the middle of transitioning between two choices for precision.

- The `PRICE_DENOMINATOR` constant defines a scaling factor of 26 decimals. The `priceMarkup` should have the same precision but is described as having 6 decimals.
- The same scaling factor is used in the `OracleHelper` contract. As such, the `priceUpdateThreshold` should have the same precision, but it is instead forced to have 6 decimals.

In the second case, the threshold effectively rounds to zero and an update will be triggered on every change. Consider using 26 decimals of precision throughout the contract.

Update: *Resolved in pull request #428.*

M-05 ERC Recommendations [core]

Procedural Update

The `validateUserOpSignature` function allows the aggregator to replace the operation signature. If this happens, the bundler should re-run `validateUserOp` with this new signature to ensure that it succeeds and returns the same aggregator. Otherwise, the operation might fail unexpectedly in the bundle.

Update Specification

There are places where the specification references outdated features of the system and thus should be updated:

- References to `UserOperation` should be replaced with `PackedUserOperation`. This includes updating the field descriptions and all the affected interfaces.
- The references to `ValidationResultWithAggregator` (1, 2) should be removed.
- The specification should mention the new `IAccountExecute` interface and how it can be used.

Technical Corrections

- The specification requires the `EntryPoint` to fail if the account does not exist and the `initCode` is empty. It actually just skips validation when the `initCode` is empty (although it would revert later when attempting to interact with the empty address). While failing explicitly would typically be recommended, this check has been moved to the simulation. For completeness, this should be explained in the ERC.
- The specification incorrectly claims that the `postOpReverted` mode implies that the user operation succeeded.
- The specification claims that the paymaster's `addStake` function must be called by the paymaster. However, it is called by the paymaster's owner address.
- The specification requires the `EntryPoint` to validate the aggregate signature after performing the individual account validations. It is actually performed beforehand.

Additional Context

The specification mentions some examples of how to attribute `AAx` errors. It would benefit from a complete table explaining all the error code prefixes.

Update: *Resolved in pull request #412.*

Low Severity

L-01 Temporarily Unusable ETH [samples]

The `TokenPaymaster` includes a mechanism to receive ETH donations. However, they cannot be used or withdrawn until they are deposited to the `EntryPoint`. Therefore, the ETH remains unusable until the deposit balance falls low enough and user operation triggers the refill mechanism.

Since the ETH will eventually become a deposit with the `EntryPoint`, consider removing this function and instead requiring donations to use the existing `deposit` mechanism.

Update: *Resolved in pull request #420, pull request #433. The Ethereum Foundation team stated:*

> *TokenPaymaster receives eth, but that's not for "donations" but part of its business logic: when converting tokens, it converts them to WETH and then to ETH, which is sent through this "receive" function. Then the paymaster uses these funds to replenish its deposit in the EntryPoint. We did add a method so that it would be able to withdraw any Eth that got accumulated there.*

L-02 Imprecise Refresh Requirement [samples]

The `OracleHelper` contract is configured for a price feed that updates every day but still accepts prices that are two days old. On a well-functioning feed, the price should never be more than one day old.

Consider using this more restrictive requirement (with a possible small buffer).

Update: *Resolved in pull request #424.*

L-03 Inconsistent Oracle Configuration [samples]

In the `OracleHelper` contract, when the `tokenOracle` price is already based in the native asset, the `nativeOracle` is unused. However, it must still be configured to a valid contract with a `decimals` function.

Consider requiring it to be the zero address in this case.

Update: *Resolved in pull request #423.*

L-04 Incomplete Generalization [samples]

The `TokenPaymaster` refers to the native asset and a bridging asset, but also explicitly mentions Ether and dollars. This is not purely descriptive. It also assumes that the Chainlink price is updated every 24 hours, even though different Chainlink oracles can have wildly different heartbeats, ranging from 1 hour to 48 hours.

Consider choosing a specific configuration, or making all parameters and comments generic.

Update: *Resolved in pull request #425.*

L-05 Misleading Comments [core and samples]

The following misleading comments were identified:

- This comment is incorrect now that deposits occupy 256 bits.
- This comment still refers to a second `postOp` call.
- The simulation functions both claim (1, 2) to always revert, but that is no longer accurate.
- The paymaster validation comment incorrectly implies that it can return a non-zero authorizer address.
- This comment still references the obsolete `ValidationResultWithAggregation`.
- The `BasePaymaster` `_postOp` comment still references the obsolete second call.
- The `paymasterAndData` parameter is incorrectly described as a paymaster address followed by a token address.
- The `requiredPreFund` parameter in the `TokenPaymaster` contract is described as the amount of tokens, but it is the amount of ETH.

Consider updating them accordingly.

Update: *Resolved in pull request #413, pull request #440.*

L-06 Incomplete Docstrings [core and samples]

Throughout the codebase, there are several parts that have incomplete docstrings:

- In the updateCachedPrice function in `OracleHelper.sol`:
  - The `force` parameter is not documented.
  - The return value is not documented.
- In the `_postOp` function in `BasePaymaster.sol`, the `actualGasCost` parameter is not documented.
- In the getUserOpPublicKey function in `BLSSignatureAggregator.sol`, the `userOp` parameter is not documented.
- In the addStake function in `BLSSignatureAggregator.sol`, the `delay` parameter is not documented.
- In the validateUserOp function in `BaseAccount.sol`, the return value is not documented.
- In the innerHandleOp function in `EntryPoint.sol`, the return value is not documented.
- In the _getValidationData function in `EntryPoint.sol`, the return values are not documented.
- In the getUserOpHash function in `IEntryPoint.sol`, the return value is not documented.
- In the delegateAndRevert function in `IEntryPoint.sol`, the `target` and `data` parameters are not documented.
- In the simulateValidation function in `IEntryPointSimulations.sol`, the return value is not documented.
- In the simulateHandleOp function in `IEntryPointSimulations.sol`, the return value is not documented.
- In the executeBatch function in `SimpleAccount.sol`, the `dest`, `value`, and `func` parameters are not documented.
- In the initialize function in `SimpleAccount.sol`, the `anOwner` parameter is not documented.

- In the balanceOf function in `StakeManager.sol`, the return value is not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of any contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #414.*

L-07 Different Pragma Directives Are Used [core and samples]

Pragma directives should be fixed and the same across file imports in order to clearly identify the Solidity version in which the contracts will be compiled.

Throughout the codebase, there are multiple different pragma directives:

- The `BLSAccount.sol` file has the pragma directive `pragma solidity ^0.8.12;` and imports the following files with different pragma directives:
  - `SimpleAccount.sol`
  - `IBLSAccount.sol`
- The `BLSSignatureAggregator.sol` file has the pragma directive `pragma solidity >=0.8.4 <0.9.0;` and imports the following files with different pragma directives:
  - `IBLSAccount.sol`
  - `BLSHelper.sol`
- The `EntryPoint.sol` file has the pragma directive `pragma solidity ^0.8.23;` and imports the following files with different pragma directives:
  - `StakeManager.sol`
  - `SenderCreator.sol`
  - `Helpers.sol`
  - `NonceManager.sol`
  - `UserOperationLib.sol`
- The `EntryPointSimulations.sol` file has the pragma directive `pragma solidity ^0.8.12;` and imports the file `EntryPoint.sol` which has a different pragma directive.

- The `OracleHelper.sol` file has the pragma directive `pragma solidity ^0.8.12;` and imports the `IOracle.sol` file which has a different pragma directive.
- The `SimpleAccountFactory.sol` file has the pragma directive `pragma solidity ^0.8.12;` and imports the `SimpleAccount.sol` file which has a different pragma directive.

Consider using the same fixed pragma version in all files.

Update: *Resolved in pull request #415.*

L-08 Incomplete Event [samples]

The `UserOperationSponsored` event includes the market price but does not include the markup price which is what the user actually paid.

Consider including the markup price in the event as well for completeness.

Update: *Resolved in pull request #431.*

Notes & Additional Information

N-01 Code simplifications [samples]

The following code simplifications were identified:

- In the `OracleHelper` contract, the `previousPrice` variable is redundant because the `_cachedPrice` and `price` already represent the old and new values. There is no need to update the `_cachedPrice` value because `price` can directly be assigned to storage. Consider removing the redundant value.
- The `UniswapHelper` contract accepts `_tokenDecimalPower` instead of calculating it from the `token.decimals()` value. Presumably, this is intended to support ERC-20 contracts that do not implement the metadata extension. However, it is initialized in the `TokenPaymaster` using the `decimals` function, which suggests that logic could be

moved to `UniswapHelper`. Although, in this case, `tokenDecimalPower` is unused and could instead be removed entirely.

Update: *Resolved in pull request #422.*

N-02 Unused Functions With `internal` or `private` Visibility [core and samples]

Throughout the codebase, there are unused functions:

- The `getGasPrice` function in `TokenPaymaster.sol`
- The `swapToWeth` function in `UniswapHelper.sol`

To improve the overall clarity, intentionality, and readability of the codebase, consider using or removing any currently unused functions.

Update: *Resolved in pull request #426.*

N-03 Multiple Contracts With the Same Name [samples]

There are two incompatible instances (1, 2) of the `IOracle` interface.

Consider renaming the contracts to avoid unexpected behavior and improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #427.*

N-04 Lack of Security Contact [core and samples]

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for the

maintainers of those libraries to establish contact with the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are contracts that do not have a security contact.

Consider adding a NatSpec comment containing a security contact above the contract definitions. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Resolved in pull request #432.*

N-05 Using `uint` Instead of `uint256` [core and samples]

The following instances of using `uint` were identified:

- The `INNER_GAS_OVERHEAD` constant in `EntryPoint.sol`
- The `g` and `x` variables in `EntryPointSimulations.sol`
- The `actualUserOpFeePerGas` variable in `TokenPaymaster.sol`

In favor of explicitness, consider replacing all instances of `uint` with `uint256`.

Update: *Resolved in pull request #417.*

N-06 Naming Suggestions [core and samples]

To favor explicitness and readability, listed below are suggestions for better naming:

- `postOp` should be `"postOpGasLimit"`.
- `paymasterAndDataLength` should just be `"dataLength"`.

Update: *Resolved in pull request #418.*

N-07 Typographical Errors [core]

Consider addressing the following typographical errors:

- "with" should be "which"

Update: *Resolved in pull request #419.*

N-08 Unused or Indirect Imports [core and samples]

Throughout the codebase, there are multiple imports that are unused or only indirectly refer to the value imported:

- Import `import "./Helpers.sol";` in `BaseAccount.sol`
- Import `import "./Helpers.sol";` in `BasePaymaster.sol`
- Import `import "../interfaces/IEntryPoint.sol";` in `NonceManager.sol`
- Import `import "../core/UserOperationLib.sol";` in `TokenPaymaster.sol`
- Import `import "@uniswap/v3-periphery/contracts/interfaces/ISwapRouter.sol";` in `OracleHelper.sol`

Consider removing unused and indirect imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #419.*

Client Reported

CR-01 `simulateHandleOp` does not set `_senderCreator` address

The `simulateValidation` function computes and sets the `_senderCreator` variable, which is required when the account is deployed in a user operation. However, the `simulateHandleOp` function does not perform the same initialization.

Update: *Resolved in pull request #411, by moving the initialization to the `simulationOnlyValidations` function.*

CR-02 Unverified `TokenPaymaster` gas limit

The `TokenPaymaster` estimates the gas cost of its `postOp` operation but does not validate that the user operation provides enough gas. If insufficient `postOp` gas is provided, the user would still pay the gas costs, but the operation would revert.

Update: *Resolved in pull request #434 at commit 900a6a8.*

CR-03 Insufficient Prefund

Note: This error was present in the previous audit commit but was not identified by the auditors at the time. It was reported to the Ethereum Foundation by OKX.

In addition to the enforceable gas limits, user operations are charged `preVerificationGas` to compensate bundlers for off-chain work and transaction overhead. Since `preVerificationGas` cannot be measured by the `EntryPoint` contract, it is directly added to the measured amount that will be charged from the user or paymaster.

However, this means that when confirming that the pre-funded charge covers the measured usage, the cost of `preVerificationGas` is implicitly included on both sides of the inequality, and is therefore irrelevant. The guard condition actually checks whether the measured gas, including transaction overhead, exceeds the enforceable limits (which do not cover overhead). If all of the enforceable limits are completely consumed, the overhead might be sufficient to trigger the revert, which would cause the entire transaction to revert at the bundler's expense.

To mitigate this risk, bundlers could ensure that at least one of the enforceable gas limits is not completely consumed during simulation so that there is enough buffer to cover the overhead. In practice, they should choose the user's verification gas limit to guarantee that the simulated buffer amount is reproduced on-chain.

Update: *Resolved in pull request #441, pull request #449.*

Conclusion

The EIP-4337 aims to enhance both the user experience and the security of Ethereum accounts without altering the consensus rules. This audit marks our third review of this EIP for the Ethereum Foundation. In this audit, our focus was directed at all non-test Solidity files that have been modified since our previous evaluation. The code was well-written and very well-documented which contributed positively to the audit process.

We identified several issues of medium severity, including a bug that could lead bundlers to include unauthorized operations during bundle simulation, resulting in unfair throttling of paymasters. We also noted various instances of outdated specifications related to the system's features. For these and all other issues outlined in this report, we have provided specific recommendations to either rectify or mitigate the associated risks.

Finally, we have highlighted other issues of lower severity and proposed general recommendations aimed at minimizing the overall attack surface.