EIP-4337 – Ethereum Account Abstraction Audit

OPENZEPPELIN SECURITY | APRIL 19, 2022

Security Audits

Introduction

EIP-4337 is a specification to add account abstraction functionality to the Ethereum mainnet without modifying the consensus rules. The Ethereum Foundation asked us to review the specification and a reference implementation.

The audited commit is 8832d6e04b9f4f706f612261c6e46b3f1745d61a and the scope included all files in the `contracts` directory excluding the `test` directory and the `SimpleWalletForTokens.sol` file. In addition, we reviewed and commented on the EIP and architectural design, which is documented in the appendix.

The `EntryPoint` and `StakeManager` contracts define the core singleton that will be deployed, while the `sample` directory contains examples that can be used as a baseline for wallet and paymaster developers. We've tagged each issue in this report with `[core]`, `[sample]` or `[core and sample]` to better clarify its impact.

***Update**: The final repository after applying the fixes noted in this report is available at commit a2f4b7be4d9996095e08d7102bacc9f13ea99ff6.*

Summary

executing arbitrary code within the same transaction. Moreover, some mitigations are implemented within the smart contracts, while others are implemented off chain. We believe that such a project would benefit from a strong commitment to defensive programming practices. Many of our specific recommendations involve failing early, specifying and enforcing assumptions, better separation of concerns, and more complete documentation. We believe this mindset applied to the code base generally would reduce the attack surface and make the code easier to reason about, change, and audit.

We also believe this audit would have benefited from a more detailed description of the off-chain processing steps that are not apparent from the code base itself, along with mitigations employed by paymasters and bundlers, including a walkthrough of the life cycle of user operations. We have included specific recommendations towards this goal within the report.

Additionally, due to the nature and complexity of the system, we recommend:

- Following best practices of software development through test-driven development and mandatory peer-reviews.
- Opening a public bug bounty program to engage independent security researchers from the community to uncover unexpected behavior as the code base evolves.
- Carefully monitoring and evaluating the system through beta testing phases until several projects have been onboarded and the dynamics have become battle-tested.

Security features and assumptions

The system is described in detail in the EIP, but we think it may be useful to highlight the dynamics of the system and the security assumptions used in this audit. This is partly based on conversations we had with the Ethereum Foundation.

User operations specify a sequence number (called a nonce in the code base and EIP) to order the transactions. Wallets can choose to ignore this value at their own risk, but this does not affect the safety of the rest of the system.

Before executing operations that do not involve a paymaster, the wallet must agree that the transaction is valid and that they're willing and able to pay for it. The `EntryPoint` contract

If the operation does involve a paymaster, the wallet must still agree that the transaction is valid, but now the paymaster must agree that they're willing and able to pay for it.
The `EntryPoint` contract guarantees the paymaster that if both validations succeed, then the user operation will be executed and the paymaster can perform after-operation functionality, whether or not the user operation reverted. In this way, paymasters can use their validation opportunity to ensure their requirements can be met (for example, by ensuring the user wallet can reimburse them with ERC20 tokens) and their after-operation functionality to actually implement their requirements (for example, by retrieving the tokens from the wallet).

However, it's possible for the wallet operation to undermine the paymaster's initial validation. For example, the operation might transfer the tokens that were required to reimburse the paymaster. When this happens, the `EntryPoint` contract will revert the whole user operation and give the paymaster a second chance to implement their requirements. If the paymaster's after-reversion functionality fails, then it must have been implemented maliciously or incorrectly.

The `EntryPoint` contract guarantees the bundler that if the wallet and paymaster validation succeeds then either:

- The user operation will be executed, the paymaster will be charged whether or not the operation reverts, and the bundler will be reimbursed, or
- The paymaster will fail in their after-reversion functionality and the bundler is justified in blaming the paymaster for this failure. The throttling and banning mechanism limits the effect of invalid paymasters.

All clients ensure the user operation passes validation (for both the wallet and optional paymaster), by simulating it locally, before accepting it into the mempool. In this way the mempool will only be filled with operations that were valid at the time they were received.

Since user operations can survive in the mempool for multiple blocks, bundlers must ensure each operation passes validation again before including it in a batch of operations. Importantly, the opcode restrictions specified in the EIP ensure that operations can only be invalidated by state changes in the wallet. Clients restrict the number of simultaneous operations per wallet in the mempool, so each state change has a limited effect.

Bundlers must only include one operation per wallet in each bundle to prevent possible interactions between the operations. Additionally, before submitting a batch to the `EntryPoint` contract, bundlers must simulate the effect of the whole batch to identify and remove any transactions that fail.

Lastly, bundlers must ensure their bundle cannot be invalidated by unrelated transactions. If they are a miner/proposer (or have an arrangement with a miner/proposer) they could ensure the bundle is the first transaction in the block, use access lists to ensure that preceding transactions do not affect the bundle, or at least make sure that bundles that would revert are not included. This ensures that the simulated bundle will be replicated on-chain.

Advice for developers

The system provides a large degree of flexibility for wallet and paymaster developers to innovate new functionality. However, there are some general principles that we believe should be taken into consideration:

- Wallets must implement their own replay protection. The `BaseWallet` ensures all transactions have different nonces, and the `SimpleWallet` demonstrates how the `requestId` (which includes the chain ID) can be used to enforce unique signatures. One goal of this system is to allow different authentication schemes but anti-replay functionality will almost always be required.
- Wallets could validate the maximum operation cost. The `validateUserOp` function is notified of any funds the wallet should send to the `EntryPoint` to complete its prepayment, but it is not informed of the total prepayment amount, so it may not know the maximum operation cost at the time of validation. If desired, it could query its current balance with the `EntryPoint` contract or simply defer the question to the user (who should not sign transactions with unacceptable costs).
- Users and wallets should consider paymaster failure cases. If the paymaster's post-operation function reverts the first time it is called, the operation will also revert. However, the post-operation function will be called again to execute its after-revert logic. Depending on the paymaster functionality, this could mean that wallets are charged for operations that are not

conditions are not met (for example, if they are not paid). Otherwise, they will be incorrectly charged for user operations.

- Paymasters should ensure that if `validatePaymasterUserOp` succeeds, then the after-revert `postOp` function must necessarily complete. Any discrepancies would allow users to create operations that will not be included in any batch. Bundlers will attribute this to paymaster misbehavior and throttle or ban the paymaster's operations from the mempool.

Critical severity

[C01] Deposit manipulation [core]

The `addStakeTo` function of the `StakeManager` contract allows an attacker to update the deposit record associated with another `account`, and manipulate it in two significant ways.

Firstly, the new funds are added to the caller's current balance instead of the current `account` balance. This effectively allows anyone to delete the deposit from any account. Secondly, the `account`'s new stake delay is directly chosen by the caller and could be unreasonably long.

Consider replacing the `addStakeTo` function with an `addStake` function that only allows the caller to update their own deposit record.

***Update**: Fixed in pull request #50. The `addStakeTo` function was renamed to `addStake` and updated such that the caller can only add value to their own stake.*

High severity

[H01] Incorrect prefund calculation [core]

In order to ensure a user operation can be financed, the maximum amount of gas it could consume is calculated. This depends on the individual gas limits specified in the transaction. Since the paymaster may use `verificationGas` to limit up to three function calls, operations that have a paymaster should multiply `verificationGas` by 3 when calculating the maximum gas. However, the calculation is inverted. This means that valid user operations could fail in two ways:

have any unused gas (including the overcharge) refunded after the operation is executed.

- Paymasters that are insufficently staked may nevertheless pass validation, which introduces the possibility that they will be unable to fund the operation. In practice, the excess funds will probably simply be deducted from their stake. Even so, this allows users to craft operations that would unexpectedly cause the paymaster to become unstaked.

Consider updating the maximum gas calculation to match the execution behavior.

***Update**: Fixed in pull request #51.*

[H02] Duplicate validation gas accounting [core]

The `handleOp` function of the `EntryPoint` contract tracks and records both the pre-operation gas and the gas consumed before the final `postOp` call. However, in contrast to the equivalent calculations in `handleOps`, both values use the same `preGas` value. This means that the gas used during payment validation is accounted for twice, and the wallet or paymaster will be overcharged for the operation.

***Update**: Fixed in pull request #61. The `handleOp` function has been removed.*

[H03] Paymasters can spend locked stake [core]

Wallets pay for their operations using the funds deposited with the `StakeManager`, whether or not those funds are locked. This is usually acceptable because wallets are not required to lock their funds. However, if a paymaster contract is also a wallet, it will be able to spend funds that are supposed to be locked as stake. This means that it can bypass the reputation system by consuming its locked funds after being throttled. Consider ensuring that wallets cannot spend locked funds.

***Update**: Fixed in pull request #53. If a paymaster has not been specified for a given user operation, the `_validateWalletPrepayment` function now checks if the `sender` has a staked deposit with the `EntryPoint` contract, and rejects the user operation if the wallet is staked. The particular requirement is more conservative than strictly necessary, but it correctly mitigates the issue and does not prevent valid use cases.*

[H04] Token transfers may fail silently [sample]

failure. For tokens that return `false`, such as the 0x Protocol Token, these transfers may fail silently, leading to incorrect internal accounting.

Consider checking the return value of all ERC20 transfers, or using OpenZeppelin's safe transfer functions.

***Update**: Fixed in pull request #54. The `DepositPaymaster` contract now uses OpenZeppelin's `SafeERC20` library functions for token transfers.*

[H05] Incorrect gas price [core]

Client reported: *The Ethereum Foundation identified this issue during the audit.*

The gas price to charge the user (potentially through the paymaster) for the operation is calculated as the minimum of the transaction gas price and the user-specified gas price (after accounting for any `basefee`). However, the user should always pay their specified price so the bundler can receive the excess, which provides the incentive to process the user operation in the first place. Consider allowing the user's gas price to exceed the transaction gas price.

***Update**: Fixed in pull request #55. The `tx.gasprice` value has been removed from the gas price calculation.*

Medium severity

[M01] Inconsistent behavior between ecrecover2 and ecrecover [sample]

The `ecrecover2` function of the `ECDSA` contract invokes the `ecrecover` precompile but ignores the return value. Since the input and output buffers overlap, whenever the address recovery operation fails, the least significant 20 bytes of the `hash` parameter are incorrectly returned as the signer.

Typically, we would recommend checking the returned status flag to identify failed address recovery operations. However, in our internal testing, the precompile unexpectedly returned `success` on all operations (whether the signer was recovered or not). An alternative option would be to use an empty output buffer, so failure to recover an address will return the zero

updating the code to correctly handle failed address recoveries.

***Update**: Fixed in pull request #56. A separate output buffer is now used to store the recovered address, and if recovery fails the zero address will be returned. The status value from the staticcall to the precompiled `ecrecover` function is also now checked.*

Following this pull request, the EIP was updated to allow the `GAS` opcode to be used when simulating validation, provided that it is followed by `CALL`, `DELEGATECALL`, `CALLCODE`, or `STATICCALL`. Additional statement from the Ethereum Foundation on this issue:

> This is no longer an issue, since we removed completely the need for "ecrecover2" (we allow the use of GAS opcode if immediately followed by "*CALL").

[M02] Separate stake and prepayment [core]

The `StakeManager` contract holds ETH on behalf of users for two different reasons:

- Paymasters lock a deposit as an anti-sybil mechanism
- Wallets and paymasters preemptively transfer funds to pay the gas costs associated with user operations

However, the `StakeManager` contract treats these funds equivalently, making the boundary harder to identify, enforce and reason about. For example, paymasters are unable to decrease the amount of funds available to users without first unstaking and waiting for the withdrawal period. Moreover, miners may choose different thresholds for the minimum paymaster stake, which means they can unilaterally partition a paymaster's deposit between the two amounts on a per-transaction basis. Consequently, unless they explicitly account for this in their validation function, paymasters cannot safely lock capital in the `StakeManager` without risking that it will be used to pay for user transactions.

Additionally, we believe the Paymasters can spend locked state issue is a consequence of this unclear boundary.

Consider separating the handling of transaction prepayments and paymaster stake to better encapsulate the two concepts.

Client reported: *The Ethereum Foundation notified us of this issue after it was reported by* `tjade273`.

The EIP states that value-bearing calls are not restricted during validation. This allows wallets to send funds to the `EntryPoint` contract to prepay for their operations. However, this allows wallets to invoke an external contract that sends funds to itself. Since this won't change the balance, it would be allowed by the validation restrictions. If the external contract spent its balance between the two simulations, the second simulation would revert. In this way, the success of a wallet validation can be made to depend on an external contract's balance.

If several operations in the mempool depend on the same external contract's balance, they can all be invalidated with a single state change, violating the intent of the EIP validation restrictions.

This is addressed in pull request #83 by modifying the validation restrictions to only allow a value-bearing call between the wallet and `EntryPoint` contract.

[M04] Gas discrepancy in bundle simulation

Client reported: *The Ethereum Foundation notified us of this issue after it was reported by* `tjade273`.

Accessing an account has a dynamic gas cost. In particular, the first access in a transaction costs more than subsequent accesses. This means that during a bundle execution simulation, when an operation's validation function references an account that was previously accessed in the same batch, it will cost less than it did during the individual simulation. The validation function can be designed to detect this difference, by running out of gas when the external account has not been accessed. It could then succeed during the individual simulation and fail in the batch.

This is addressed in pull request #83 by modifying the validation restrictions to ensure no call results in an out-of-gas revert.

Low severity

[L01] Use of transfer function is potentially unsafe [core and sample]

in `SimpleWallet` both uses Solidity's built-in `transfer` function (on line 129 and line 52, respectively) to send ether to a destination address. The use of `transfer` for this purpose is no longer recommended.

Consider using the `call` function or OpenZeppelin's sendValue function, and adhere to the checks-effects-interactions pattern when sending value to an external address. This pattern is already implemented in the `compensate` function of the `EntryPoint` contract.

***Update**: Partially fixed in pull request #57. The `SimpleWallet` contract's `transfer` function was left unchanged.*

[L02] Incorrect event parameter [core]

After processing a deposit, the `StakeManager` emits a `Deposited` event, but incorrectly uses the `msg.sender` instead of the updated account as the `account` parameter. This may mislead observers and offline processing systems. Consider updating the `account` parameter to match the deposit functionality.

***Update**: Fixed in pull request #50.*

[L03] Finite, fixed, and unrestricted token paymaster allowance [sample]

On deployment, the `TokenPaymaster` assigns the maximum token allowance to its owner. However, there is no mechanism to update the allowance.

In principle, this means the allowance can run out. More plausibly, if ownership is transferred, the old owner will still be able to spend the tokens, and the new owner will not. Consider allowing the owner to refresh their token allowance. Additionally, consider removing the existing allowance when ownership is transferred.

Moreover, the paymaster mints a single token unit to ensure the contract balance and total supply is non-zero for more predictable gas accounting. However, the owner can withdraw the contract's total balance, restoring it to zero. Consider preventing the withdrawal of the last token unit.

allowance is the maximum token allowance. This does implicitly allow refreshing a depleted allowance. The suggestion to prevent removal of the last token unit was not implemented. Ethereum Foundation's comment on this issue:

> While we `mint(1)` in the constructor, to make all postOp calls cheaper, we don't protect against the owner pulling this "last wei": This is an "optimization" the owner should bear in mind that if the paymaster is depleted, the next transaction will cost the OWNER (not the calling wallet) more to change the balance.

It is worth noting that this analysis assumes the `COST_OF_POST` parameter includes the higher balance-changing cost, so in some sense, the calling wallet always pays the higher cost. Otherwise, it would be possible for the first `_postOp` call (and hence the user operation) to fail when the `TokenPaymaster` contract has zero balance.

[L04] Inconsistent minimum stake delay [core]

The `StakeManager` contract specifies a minimum unstake delay for paymasters to withdraw their stake, but this minimum is not enforced when staking.

The Ethereum Foundation has indicated that they intend to remove the minimum entirely, and instead allow it to be a floating parameter, negotiated between miners and paymasters. To this end, they will introduce two unused parameters to the `handleOps` function call, set by the miner, specifying their minimum acceptable stake and delay values. These are merely signals that won't be enforced by the contract. Note that this solution would also involve removing the `EntryPoint` contract's `paymasterStake` parameter and modifying the `isPaymasterStaked` function to accept a delay variable.

While we acknowledge and endorse this solution, we would still recommend ensuring the stake delay is non-zero, since this parameter is semantically overloaded as a flag to indicate if the paymaster is staked.

***Update**: Fixed in pull request #59. Checks have been added to ensure that the `unstakeDelaySec` value in `StakeManager` is non-zero and that the individual `unstakeDelaySec` value for each user stake is greater than or equal to the*

[L05] Incorrect balance check comparison in TokenPaymaster [sample]

In the `TokenPaymaster` contract, the `validatePaymasterUserOp` function performs a check to verify that the sender's token balance is sufficient to pay the `tokenPrefund` cost for the user operation. In both cases where this check occurs in the function, `>` is used for the comparison when `>=` should also be valid. A user could have exactly the required prefund amount and still fail to pass validation.

Consider modifying the balance checks to support the case where the user's token balance exactly matches the `tokenPrefund` amount.

***Update**: Fixed in pull request #60.*

[L06] handleOp function is redundant [core]

In the `EntryPoint` contract, in addition to the `handleOps` function documented in the EIP-4337 specification, there is also a `handleOp` function that just implements the special case of `handleOps` where there is only a single `UserOperation` to be processed. This is inconsistent with the specification and increases the attack surface. It also introduces the possibility that the logic will differ between the two implementations, as demonstrated by Duplicate validation gas accounting.

Consider removing the `handleOp` function from the `EntryPoint` API, or using the `handleOps` function as its implementation.

***Update**: Fixed in pull request #61. The `handleOp` function has been removed.*

[L07] Event misordering possibility [core]

The `withdrawTo` function of the `StakeManager` contract emits an event after sending the funds. This violates the Checks-Effects-Interactions pattern and it introduces the possibility that some events could be emitted out of order if the recipient's fallback function executes another operation. Consider emitting the event before the funds transfer.

***Update**: Fixed in pull request #57.*

The `StakeManager` contract allows paymasters to lock funds for a period of time and they are intentionally prevented from reducing the delay. However, after unstaking their funds and waiting for the withdrawal period, they should be able to stake again with any delay. This is possible if they withdraw their funds first (which clears the saved delay), but this is should not be a necessary requirement.

Consider updating the staking guard condition to allow the delay time to be reduced if the `withdrawTime` has been reached.

***Update**: Fixed in pull request #76. Staking state is no longer affected by the `withdrawTo` function. Users can now unlock an existing stake using `unlockStake` without needing to withdraw the funds, and they can then immediately restake by calling the `addStakeTo` function.*

[L09] Wallet may not be deployed [core]

The `_validateWalletPrepayment` function of the `EntryPoint` will attempt to deploy the wallet if the operation specifies an `initCode`. This will fail if the wallet cannot be deployed, or it does not match the sender address. However, operations without an `initCode` do not guarantee that the wallet has already been deployed. If not, the `validateUserOp` call will unexpectedly revert *before* execution enters the `try` block (so the `FailedOp` event will not be triggered).

In practice, this should be identified by the bundler when constructing the batch. Nevertheless, in the interest of predictability, consider ensuring `_createSenderIfNeeded` always ends with a wallet deployed at the expected address.

***Update**: Acknowledged. The Ethereum Foundation decided not to address this because the bundler should identify this during simulation.*

[L10] Unchecked math blocks are not narrow in scope [core]

In the `EntryPoint` contract, there are several functions (`handleOp`, `handleOps`, `_validateWalletPrepayment`, `_validatePaymasterPrepayment`, `handlePostOp`) where all or nearly all of the function body is enclosed within

checked may be inadvertently added within the block.

For safety and clarity, consider restricting the scope of unchecked math blocks to only include the specific lines where it is needed.

***Update**: Not fixed. According to the Ethereum Foundation*

> for code clarity, it was added to wrap entire methods, as it makes the code far less readable if we try to wrap only the math expressions.

[L11] Missing docstrings [core and sample]

Many functions in the code base lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update**: Partially fixed in pull request #81. New docstrings were added and existing ones were expanded; most functions now have a docstring. However not all input and output parameters are explicitly documented. For example, several functions in the `StakeManager` contract do not have NatSpec comments.*

[L12] Downcasting without bounds checks [core]

The `StakeManager` contract downcasts the `amount` when incrementing or decrementing deposits, and when increasing an account's stake. Although these values are unlikely to overflow, as a matter of good practice, consider adding bounds checks.

if its size exceeds `type(uint112).max`.

[L13] Missing validations [core]

- The `compensate` function in `EntryPoint` takes a `beneficiary` address as input and sends the amount specified to that address. This happens as the final step of a `handleOps` or `handleOp` function call. The code does not check that `beneficiary` is not 0, which could lead to accidental loss of funds. Consider adding a check to verify that `beneficiary` is a non-zero value.
- In the `EntryPoint` constructor, there are no checks to ensure the immutable contract variables are set to non-zero values. If `_create2factory`, `_paymasterStake`, or `_unstakeDelaySec` were accidentally set to 0, the contract would need to be redeployed because there is no mechanism to update these values. Consider adding a non-zero check for each of the constructor parameters.

***Update**: Fixed in pull requests #59 and #63. Checks that reject zero values have been added for `beneficiary`, `_create2factory`, `_paymasterStake`, and `unstakeDelaySec`.*

[L14] DepositPaymaster warning [sample]

The `DepositPaymaster` contract is non-compliant with the current version of the EIP, because it accesses an external oracle during validation. Consider including a warning at the top of the contract to explain the associated risks and how bundlers should decide whether to support this paymaster.

***Update**: Fixed in pull request #64.*

[L15] Wallet storage in EntryPoint [core]

Client reported: *The Ethereum Foundation identified this issue after the audit.*

Bundlers ensure that wallets can only access their own mutable storage. However, they need to include an exception for the wallet's balance in the `EntryPoint` contract to allow the wallet to prepay for its operation. The `EntryPoint` contract should expose a mechanism for bundlers to query the necessary storage locations.

Notes & Additional Information

[N01] Suggested EIP changes

Client reported: *The Ethereum Foundation made or inspired many of these suggestions during the audit. We include them here for reference.*

Here are some suggestions to improve the precision and clarity of the EIP and associated documentation:

- To prevent cross-chain replay attacks, user operation signatures should depend on the `chainid`. This is handled in the current implementation but not yet included as a requirement in the EIP.
- After constructing a batch, bundlers should execute `eth_estimateGas` with the maximum gas limit. This would mitigate potential return bombing or other gas manipulation attacks that could cause batches to fail without specifically identifying the offending operation.
- State any restrictions on how bundlers should construct batches, including ensuring all user operations have different senders.
- State any restrictions on how batches should be arranged or ordered in blocks (e.g. using the access lists) to avoid interference from other transactions.
- Clarify any operations that paymasters can undertake that wallets cannot. Additionally, explain the conditions (e.g. whitelisting) under which a bundler may accept a non-compliant paymaster, such as the `DepositPaymaster`.
- Although implicit in the specification, we believe a complete walkthrough of the lifecycle of a user operation would be instructive.

***Update**: Fixed in pull request #83.*

[N02] Imprecise gas limits [core]

The `_validatePrepayment` function of the `EntryPoint` contract combines the various gas parameters into a single variable so they can be collectively compared against a hardcoded limit. Although this correctly ensures all values are individually less than the maximum `uint120` value, it is still possible for the combination of smaller values to equal the

must fit within a `uint120` variable".

***Update**: Fixed in pull request #86.*

[N03] Duplicated naming [core]

In the `EntryPoint` contract, the `paymasterStake` variable has the same name as one of the `PaymentMode` options.

Consider using different names to improve code readability and avoid confusion.

***Update**: Partially fixed. Pull request #76 relocates the `paymasterStake` variable from the `EntryPoint` contract to the `StakeManager` contract, but does not rename it.*

[N04] ECDSA signature length check allows invalid values [sample]

In the `VerifyingPaymaster` contract, the `validatePaymasterUserOp` function contains a check that the ECDSA signature being verified has a length >= 65 bytes. The `tryRecover` function in the OpenZeppelin `ECDSA` library does not support signatures longer than 65 bytes. It's possible for an invalid signature length to pass the check in `validatePaymasterUserOp` and later cause the transaction to revert.

Consider modifying the check in `validatePaymasterUserOp` to only allow ECDSA signature lengths equal to 65.

***Update**: Fixed in pull request #66. The signature length must now be equal to 64 or 65 to be considered valid. The `tryRecover` function supports both length values. Note that this expands the previous functionality to support the 64-byte encoding, so there are now multiple valid signatures for the same operation.*

[N05] Fixed Oracle [sample]

When the owner of the `DepositPaymaster` adds a new supported token-oracle pair, it ensures the token does not already have an oracle. There is no mechanism to change or remove the oracle. We are simply noting this in case it's an oversight.

After executing the user operation, the `internalHandleOp` function of the `EntryPoint` contract invokes `handlePostOp` with a zero `opIndex`, regardless of the actual position of the operation within the branch. In this particular invocation, the `opIndex` parameter is unused, so setting it to zero was chosen as a simplification and gas optimization. Nevertheless, in the interest of code clarity, robustness and to support local reasoning, consider either refactoring the code to avoid the unnecessary parameter, passing in the correct value, or clearly documenting any misleading parameter assignments.

***Update**: Fixed in pull request #78. A comment was added to explain the zero `opIndex` value.*

[N07] Inconsistent naming convention [core]

We identified the following examples of inconsistent naming:

- In `EntryPoint.sol`:
  - Most internal and private functions are prefixed with an underscore, while compensate and handlePostOp are not.
  - The `UserOpInfo` struct has a single parameter that starts with an underscore.
  - The `_salt` parameter of the `getSenderAddress` function starts with an underscore.

Additionally, the prefix “internal” in function names may cause confusion. It seems redundant for functions declared with the `internal` keyword, such as the deposit manipulation functions and misleading for the `external` `internalHandleOp` function. We believe the prefix is intended to be descriptive of the actual function behavior, but nevertheless would recommend a different prefix, perhaps “local”, to avoid overloading Solidity keywords.

For clarity and readability, consider using a consistent naming convention.

***Update**: Fixed in pull requests #63 and #67. The function and variable naming was made consistent by adding or removing underscores in the requested locations. The `internalHandleOp` function usage has been clarified by renaming it to `innerHandleOp` and adding an explanatory docstring. The deposit manipulation functions still retain the redundant “internal” prefix.*

as `constant`. This change will eliminate the use of a storage slot for this value.

For clarity, also consider assigning a named constant to the values 16000 and 35000 that appear within the `validatePaymasterUserOp` functions.

***Update**: Fixed in pull request #68. In the `TokenPaymaster` contract the `constant` keyword was added to the existing `COST_OF_POST` variable and the hard-coded value of 16000 was replaced with `COST_OF_POST`. In the `DepositPaymaster` contract a new `COST_OF_POST` constant was added and assigned the hard-coded value of 35000. It is also included in the amount charged to the wallet.*

[N09] Failing tests

On a fresh checkout of the project repository and `npm install` of the dependencies, the entire test suite fails to run. This has been identified as the result of a mismatch between specific package versions used during development vs. the current versions of those packages.

On a stable production branch, it is recommended to lock the package versions within the `package.json` file to prevent package updates from breaking code that has been previously verified to work correctly.

***Update**: Not an issue. Tests can be run successfully if the `yarn install` command is used instead of `npm install`. Providing the exact instructions for building and testing the project in a README file would benefit other developers.*

[N10] Unindexed event addresses [sample]

To support log filtering for the `EntryPointChanged` event in the `SimpleWallet` contract, consider adding the `indexed` keyword to the address parameters in the event.

***Update**: Fixed in pull request #69.*

[N11] IWallet doesn't strongly enforce required functionality [sample]

The `IWallet` interface specifies a single `validateUserOp` function that must be implemented by the wallet developer. This function's docstring lists requirements that *must* be

are not required to use this code and could make errors when attempting to refactor it into their own custom implementation.

To help developers create secure wallet implementations that follow the EIP specification, consider removing the `IWallet` interface and replacing it with an abstract "BaseWallet" contract that implements all of the mandated checks, but leaves the custom behavior such as `validateSignature` up to derived classes. The existing `SimpleWallet` contract would then be derived from BaseWallet.

***Update**: Fixed in pull request #82. A new `BaseWallet` contract has been added that requires wallet implementers to adhere to the `validateUserOp` function structure originally laid out in the `SimpleWallet` sample contract; `SimpleWallet` now inherits from `BaseWallet`.*

[N12] Not all state variables have explicit visibility [sample]

Several contracts in the samples directory don't explicitly declare the visibility of the state variables and constants:

In the `DepositPaymaster` contract: – `nullOracle` – `unlockBlock`

In the `TokenPaymaster` contract: – `knownWallet`

In the `SimpleWallet` contract: – `ownerNonce`

For clarity, consider always explicitly declaring the visibility of functions and variables, even when the default visibility type matches the intended type.

***Update**: Fixed in pull request #70.*

[N13] One oracle per token restriction [sample]

The `getTokenToEthOutputPrice` function of the `IOracle` interface does not specify the particular token to translate. This means that each `IOracle` contract can only support one token, which seems like an unnecessary restriction. Consider including a `tokenAddress` parameter to the function specification.

[N14] Unnecessary encapsulation [sample]

In the `SimpleWallet` contract, the `owner` and `nonce` state variables are grouped together into an `OwnerNonce` struct. Since these variables would still pack into a single storage slot without the struct, encapsulating them together in `OwnerNonce` does not appear to provide any benefit in the current design, but does require the addition of custom `owner` and `nonce` getter functions.

Consider removing the `OwnerNonce` struct and having standalone `nonce` and `owner` state variables.

***Update**: Fixed in pull request #70.*

[N15] Inconsistent solidity version [core and sample]

The majority of the code base supports solidity versions `^0.8.7`, but the `StakeManager` contract supports versions `^0.8` and the `ECDSA` contract supports versions `^0.8.0`.

Consider using consistent solidity versions throughout the code base. Additionally, in the interest of predictability, consider locking the contracts to a specific supported version.

***Update**: Fixed in pull request #71 and commit 4efa5fc296f0034dbf402dcd558a07be45bdd761. All contracts were updated to use Solidity version `^0.8.12`.*

All of the contract versions remain unpinned.

[N16] Undocumented assembly [core and sample]

There are several contracts where inline assembly code is undocumented. Using inline assembly is risky because it bypasses some of the compiler's safety checks, it's harder to read and audit, and more likely to contain errors. For these reasons, the recommended best practice is for every line of assembly to have a corresponding explanatory comment.

Consider adding documentation to the following lines of code:

***Update**: Partially fixed in pull request #72. High-level comments were added to assembly blocks but individual lines remain uncommented.*

[N17] Excessive code optimization [core]

In the `UserOperation` contract, the `getSender` function takes a `UserOperation` struct as input and returns the `sender` field of the struct using inline assembly code. This is the only struct member that has its own custom getter function, and even though there are two addresses in the struct (`sender` and `paymaster`), only this one has a getter function. It is recommended to avoid using inline assembly because in general it is less safe and error-prone. In this particular instance, the code works because `sender` is the first item in the `UserOperation` struct, but if the `sender` variable position in the struct were to change, this code would break.

For safety and clarity, consider removal of the `getSender` function.

***Update**: Not fixed. A comment was added in pull request #72 that indicates this function saves 800 gas by using inline assembly.*

[N18] Indirect import [sample]

The `TokenPaymaster` contract imports the obsolete `SimpleWalletForTokens` contract as an indirect mechanism to import the `SimpleWallet` contract. Consider replacing the indirect import with a direct one.

***Update**: Fixed in pull request #73.*

[N19] Unused imports [sample]

To improve readability and avoid confusion, consider removing the following unused imports:

- In the `TokenPaymaster` contract, the hardhat console library.
- In the `SimpleWallet` contract, the hardhat console library.
- In the `TestCounter` contract, the UserOperation contract and the IWallet interface.
- In the `TestOracle` contract, the OpenZeppelin ERC20 contract.
- In the `TestUtil` contract, the IWallet interface.

Consider making the following changes to eliminate redundant code:

- In `UserOperation.sol`, the `requiredGas` function calculates `mul` by evaluating the expression `userOp.paymaster != address(0)`, which is equivalent to `hasPaymaster(userOp)`.
- In `UserOperation.sol`, the `pack` function has an unnecessary return statement.
- In `DepositPaymaster.sol`, the statement to silence an unused variable warning for `mode` can be removed. The `mode` variable is used to determine the payment method.

***Update**: Fixed in pull request #73.*

[N21] Inconsistent clearing of memory [sample]

The samples directory contains a custom version of OpenZeppelin's `ECDSA` library, which replaces the existing `ecrecover` function with an alternate `ecrecover2` assembly implementation that does not use the GAS opcode. After `ecrecover2` has assigned the return value `signer`, it executes a few more instructions to zero out the first two words of the free memory space, overwriting the `signer` and `v` values. From the associated comment it can be inferred that memory space is being zeroed out for the benefit of future callers that might assume the memory has been cleared. However, the current implementation is not clearing all of the modified memory–it only zeroes out the `signer` and `v` values, and leaves `r` and `s` untouched.

Consider either clearing all of the memory used, or none of it. The Solidity documentation makes it clear that users should not expect the free memory to point to zeroed-out memory, so this final clearing operation is not necessary.

***Update**: Fixed in pull request #56. The memory clearing instructions were removed.*

Following this pull request, the EIP was updated to allow the `GAS` opcode to be used when simulating validation, provided that it is followed by `CALL`, `DELEGATECALL`, `CALLCODE`, or `STATICCALL`. Additional statement from the Ethereum Foundation on this issue:

[N22] Unclear use of mixed size values [core]

The `StakeManager` contract defines the `DepositInfo` struct with three values of size `uint112`, `uint32` and `uint64`. While the intention to pack the contents into a single word can be inferred, the reason for the particular sizes are not obvious. Consider documenting the reason for this design pattern and the corresponding (reasonable) assumptions about the maximum sizes of each type.

***Update**: Fixed in pull request #76.*

[N23] Unused parameter in validatePaymasterUserOp methods [sample]

The `validatePaymasterUserOp` function in the `IPaymaster` interface takes a `requestId` parameter which can be calculated by calling the `getRequestId` function of `EntryPoint`. None of the paymaster sample contracts (`TokenPaymaster`, `VerifyingPaymaster`, `DepositPaymaster`) use this variable in their implementations of `validatePaymasterUserOp`, making it unclear as to why it is being included in the interface.

Consider providing a sample paymaster contract that demonstrates the use of the `requestId` parameter for validation.

***Update**: Acknowledged.*

[N24] Naming suggestions [core and sample]

We believe some functions are variables could benefit from renaming. These are our suggestions:

- The `IOracle`, `DepositPaymaster` and `TokenPaymaster` contracts all have a `getTokenToEthOutputPrice` function, but in all cases, it's not a price (because it accounts for the amount bought) and it appears to be described backwards. Something like `getTokenValueOfEth` would be clearer.
- The last parameter of `validateUserOp` in the `IWallet` interface should be `missingWalletFunds` or `additionalFundsRequired` to match how it's used in the `EntryPoint` contract.

because neither uses the “stake” for gas payments.

- The `sender` parameter in the `_call` function of the `SimpleWallet` contract should be `target`.
- The `maxPriorityFeePerGas` component of the `UserOperation` struct should be `priorityFeePerGas`.
- In `BasePaymaster`, the `setEntrypoint` and `_requireFromEntrypoint` functions should capitalize the “p” for consistency with the rest of the code base.

***Update**: Partially fixed in pull request #80 and commit 074672b6ccfb596fe7ff44e13783881a2e1cfed2. The following naming suggestions were not implemented:*

- *The `maxPriorityFeePerGas` component of the `UserOperation` struct should be `priorityFeePerGas`. Ethereum Foundation comment on this issue:*

  > Note that maxPriorityFeePerGas was left unchanged, since it defines the maximum, not actual fee paid by the user.

[N25] Typographical errors [core and sample]

Consider addressing the following typographical errors:

- In `EntryPoint.sol`:
  - line 29 and line 30: “simulateOp” should be “simulateValidation”
  - line 178: “of” should be “if”
  - line 297: “done, by” should be “done and”
- In `IWallet.sol`:
  - line 10: “successfuly.” should be “successfully.”
- In `StakeManager.sol`:
  - line 12: “blocks to” should be “seconds to wait”
  - line 83: “time” should be “duration”
- In `DepositPaymaster.sol`:

- line 88: “(its” should be “(it’s”
- In `TokenPaymaster.sol`:
  - line 11: “os” should be “or”
  - line 18: “method-ids.” should be “method ids.”
  - line 77: “paymaster” should be “entrypoint”
- In `VerifyingPaymaster.sol`:
  - line 57: “signing” should be “to signing”
- In `eip-4337.md`:
  - line 46: “for to compensate the bundler for” should be “to compensate the bundler”
  - line 84: “worlflow” should be “workflow”
  - line 89: “simulateWalletValidation” should be “simulateValidation”
  - line 127: “paymaster” should be “paymaster)”
  - line 151: “ `op` ” should be “ `op` validation”
  - line 165: “valiation” should be “validation” and the line should end in a period.

***Update**: Fixed in pull request #79 and pull request #88.*

[N26] Abstract StakeManager contract [core]

The `StakeManager` contract is intended to provide staking functionality to other contracts, but should not be deployed directly. To better signal this intention, consider declaring the contract as `abstract`.

***Update**: Fixed in pull request #76.*

[N27] Declare uint as uint256 [core and sample]

To favor explicitness, consider declaring all instance of `uint` as `uint256`.

***Update**: Fixed in pull request #77.*

Conclusions

Appendix: Architectural Analysis

Overview

Many security features and mitigations to protect nodes and bundlers are executed partially or entirely off-chain; they also involve complex interactions between multiple parties with competing interests. Based on the EIP and our discussions with the Ethereum Foundation, we have captured our understanding of these features to correct any misunderstandings and identify any shortcomings. For clarity, we are referencing the latest design, which includes modifications that were made during or after the audit. This document attempts to describe high level concepts, and then systematically interrogate the details.

Basic Structure

Actors

Wallets

Smart contract *wallets* are primary accounts. Users construct arbitrary operations and publish them to a new mempool and each operation is associated with a specific smart contract wallet. In practice, each operation is an arbitrary call *to the wallet*. This allows the wallet to perform an action from its own context. A wallet can be deployed as part of its first operation.

Paymasters

Operations can specify optional *paymasters*. These are contracts that agree to pay the gas fees associated with the operation. Typically, they would be reimbursed somehow (perhaps with an ERC20 token), but the system does not specify or enforce any particular incentive. They must stake some funds beforehand as an anti-sybil mechanism. They must also prepay for the operations that they will fund.

Bundlers

could also be bundlers. They will pay the gas fees associated with the bundle at the transaction gas price. However, users specify their own gas price with each operation and bundlers will be reimbursed at this rate. The difference between the user-specified gas price and the transaction gas price provides the incentive for bundlers to participate.

Miners

Miners treat bundles like normal Ethereum transactions (with a small exception related to transaction ordering, explained below).

Nodes

Nodes are clients that participate in the mempool gossip network without necessarily mining or bundling.

Execution

Bundlers submit their bundle transactions to the global *EntryPoint* contract. For each operation in the bundle, the *EntryPoint* validates that the wallet contract accepts it as valid, and that the wallet or paymaster (if specified) is willing and able to pay for it. The paymaster can use this opportunity to validate that the wallet will reimburse them. If any of the validations fail, the whole bundle reverts.

For each (now validated) operation, the *EntryPoint* creates a new call frame and performs the following steps:

1. Executes the operation (i.e. invokes the wallet with the operation data) and traps any errors.
2. Hands control to the paymaster (if it exists) to perform any after-operation functionality, whether or not the operation succeeded. Typically, this would include retrieving reimbursement for the cost of the operation (that the paymaster will pay for).

This call frame won't revert in the first step because errors are caught. If the call frame reverts in the second (after-operation) step, its effects are rolled back, and the paymaster is given another chance to perform any after-revert functionality. If this after-revert step reverts, the whole bundle transaction reverts.

execution. When all transactions in the bundle have been processed, the *EntryPoint* sends the funds to a bundler-specified address.

Rationale

Valid operations should be charged

If an operation is validated, someone (either the wallet or the paymaster) has agreed to pay for execution, whether or not the operation itself succeeds. Ideally, the *EntryPoint* would guarantee that the operation is executed and the bundler will be reimbursed. In practice, the *EntryPoint* guarantees that either:

- All operations in the bundle will be executed and the bundler will be reimbursed, or
- The paymaster is misbehaving in an identifiable way, or
- The bundle is poorly constructed.

Bundlers are responsible for choosing valid operations

If any of the validations fail on chain, the bundler is at fault. Bundlers should:

- Only include valid operations within a bundle.
- Use the simulation mechanism (described below) to identify invalidated operations.

If the bundler-specified beneficiary address cannot be paid (for example, if its fallback function reverts), the bundler is at fault. In either case, the transaction reverts and the bundler has to pay the gas costs. Nobody else is affected.

Wallets are responsible for operation execution failures

If an operation fails, the wallet is at fault. Wallets should only accept valid operations. In the event that an operation fails, the wallet or paymaster is still charged for the execution. Paymasters are still provided with the same opportunity to retrieve reimbursement from the wallet.

Wallets are responsible for maintaining paymaster preconditions

those funds as reimbursement. However, paymasters cannot guarantee that this will succeed because the user operation could invalidate the precondition. For example, the user operation could spend the tokens or revoke the allowance.

Wallets should only choose paymasters with agreeable after-operation functionality, and only approve operations that they know will succeed. If the after-operation function reverts:

- The user operation will revert in the same call frame, which means its effects are rolled back.
- The paymaster will typically use its after-revert function to retrieve the funds anyway.
- The paymaster will pay the gas costs and the bundler will be reimbursed.

The overall effect is that the after-operation function is treated like part of the operation. If it fails, the wallet pays for the execution of a reverted operation.

Paymasters are responsible for guaranteeing payments

As noted above, wallets may cause the paymaster's after-operation function to revert. In this scenario, the *EntryPoint* will invoke the paymaster's after-revert function. When considering an individual operation, the paymaster validation step should guarantee the after-revert function will succeed, if it is invoked. Since the user operation has reverted (and its effects are rolled back), the operation cannot undermine this validation. The paymaster will typically use this function to retrieve reimbursement from the wallet; the validation would involve ensuring the wallet has sufficient funds with a corresponding allowance.

If the after-revert function reverts, the paymaster is typically at fault. However, the *EntryPoint* cannot penalize them directly (by trapping the error and charging them for the gas) because it's possible that a valid after-revert function reverts when included in a poorly constructed batch (which would be the bundler's fault). The batch could contain operations that affect the validity of subsequent operations (for example, if multiple operations spend the same funds).

Bundlers are responsible for bundle success

As noted above, when a paymaster's after-revert function reverts, the *EntryPoint* cannot identify if it's due to a poorly constructed paymaster or bundle. Therefore, the bundler is responsible for

consider the paymaster to be at fault. The off-chain simulation and paymaster reputation system (both explained below) limit the amount of griefing the paymaster can get away with.

Simulation

Overview

Bundlers are responsible for including operations that will pass validation (so the *EntryPoint* knows they are accepted by wallets and paymasters). They are not responsible for ensuring the operation itself succeeds or the paymaster is reimbursed. Bundlers use off-chain simulation to validate bundles before submitting them. It is worth noting that this process and the associated restrictions are intended as a safety feature for bundlers. It gives them confidence that bundle transactions will succeed and they will be compensated for the gas costs. Bundlers that understand the risks can choose to relax some of these restrictions.

First validation simulation

Clients should call *simulateValidation* (locally) on each user operation before accepting it into the mempool. This performs the validation for the wallet and paymaster (if it exists) against the current state of the blockchain. In this way, the mempool is only populated with operations that would have been valid at the time they were accepted.

Forbidden opcodes

The EIP lists some opcodes that cannot be used during validation because they contain environmental information about the execution context (not directly related to the state of the blockchain). This implies they can differ between off-chain simulation and on-chain execution. If these opcodes were used, a bundler could be misled into including transactions that would fail on-chain validation. The *EntryPoint* cannot distinguish this case from one in which the bundler simply included unauthorized transactions; the transaction would revert and the bundler would pay the gas costs. This is intended as a safety feature for bundlers, but it manifests as a restriction on wallets and paymasters.

Second validation simulation

possible that previously valid operations become invalid. The second simulation ensures

operations are still valid.

Forbidden state accesses

It is acceptable for some operations to become invalid between the first and second simulation. However, if too many operations were invalidated quickly, bundlers would waste time testing and rejecting them all. The EIP design principle is that each invalidated operation should cost an on-chain storage change. Stated more directly, a single on-chain storage change can only invalidate one operation in the mempool. This introduces a natural mitigation against intentionally invalidating multiple operations in the mempool as a denial-of-service attack (since changing on-chain storage costs significantly more than the nuisance caused by invalidating mempool operations).

Bundlers should enforce another set of validation restrictions: Wallets and paymasters are only able to access their own mutable storage. A minor exception: Wallets can also access their own balance with the *EntryPoint* contract. Similarly, value-bearing calls during validation are only allowed from the wallet to the *EntryPoint*. This is because operations without paymasters need to be prefunded. Wallets can send funds to the *EntryPoint* during their validation step. Since each wallet has a unique balance entry in *EntryPoint*, for the purposes of this restriction, it can be considered to be part of the wallet's own mutable storage. There is no restriction on accessing immutable storage with one exception: The bundler also saves the code hash of all accessed accounts and ensures they match across both simulations, to avoid discrepancies caused by contract code changes (i.e. as a result of `SELFDESTRUCT` or `CREATE` operations).

As with the opcode banning, this is a safety feature for bundlers that manifests as a new restriction on wallets and paymasters. It is more plausible that wallets or paymasters will want to base validation decisions on external storage. Still, most valid use cases are supported. Additionally, bundlers can choose to (internally) whitelist contracts or functions that they know are safe, even if they violate this restriction. It's worth emphasizing that there is no storage restriction during the execution of the operation (which would be very prohibitive).

Bundle execution simulation

The second validation simulation ensures that each operation in a batch individually passes validation. It does not guarantee that the bundle execution will succeed. A bundle of valid

- A paymaster's after-revert operation fails. This could occur if multiple transactions in the batch spend the same funds.
- The bundle runs out of gas, particularly as a result of return bombing
- A paymaster has insufficient funds to cover all operations in a batch. Bundlers should identify this scenario as they are constructing the batch.

Note that if an operation's execution undermines a later operation's execution, this is not an attack on the bundler. This is analogous to a normal Ethereum transaction in the mempool invalidating another transaction in the mempool. The second operation will fail, but the bundle will still succeed. To detect batch failures, bundlers should run *eth_estimateGas* on the entire bundle before submitting it. This step is used to obtain the gas required by the bundle. Moreover, if any operation fails, it should be removed from the batch. In this way (when combined with the opcode restrictions), bundlers know that the batch will succeed when executed on the current state of the blockchain.

Bundles have unique wallets

The EIP specifies that each operation in a bundle corresponds to a different wallet. When combined with the forbidden state access rules, this ensures wallet validations cannot interfere with each other. In practice, this is not much of a restriction since wallets will likely use nonces for transaction ordering. This means there should only be one valid operation per wallet at a time. Users can add additional operations with the same nonce to the mempool (e.g. to replace the transaction or increase its gas price), but only one should be included in the batch anyway. Note: There can be repeated paymasters within a bundle. Paymaster interference is handled by the reputation system (explained below). Additionally, our discussion with the Ethereum Foundation suggests that the mempool can only have a limited number operations corresponding to the same wallet.

Execution

Determinism

Bundlers must ensure that the on-chain bundle execution exactly matches the off-chain execution. Otherwise, an unrelated transaction could invalidate the bundle. To this end, bundlers must be a

- None of the preceding transactions affect the bundle. This could be detected and enforced with access control lists.

Paymaster Reputation System

Paymasters have two opportunities to undermine bundlers:

- Since the same paymaster can be used in multiple operations in the mempool, several operations can potentially be invalidated by a single on-chain state change.
- Paymasters can have after-revert functions that incorrectly revert.

In both scenarios, the bundler will identify the issue during the bundle execution simulation, and remove the offending operations. Over time, this will result in operations from the same paymaster being added to the mempool (and expiring after 10 blocks) without being included in bundles. As explained in the EIP, paymasters must stake some funds (as an anti-sybil mechanism), and will be throttled or banned if they have too many operations that are not included in bundles. Their stake is not slashed and the rate-limiting calculation is based on the amount of inclusions from the previous day, weighting the more recent hours more heavily. In this way, throttling is naturally reversed over time.

It is worth noting that even though bundlers can construct invalid batches from valid operations (explained above), this is not an attack against paymasters as long as there are some bundlers who include the operations in sensible batches. The possibility of invalid batches from valid operations is why the *EntryPoint* cannot penalize paymasters directly when their after-revert function fails, but invalid batches themselves do not directly affect the rest of the system because they are never published.

Related Posts

Beefy Zap Audit

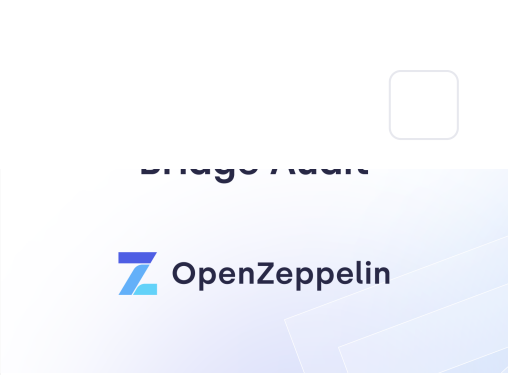
BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits



Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits