OpenZeppelin | security

Uniswap Calibur Audit

Uniswap

May 20, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-04-21 To 2025-05-16
Languages	Solidity

Total Issues	29 (20 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	4 (3 resolved)
Low Severity Issues	14 (8 resolved, 1 partially resolved)
Notes & Additional Information	10 (8 resolved)
Client Reported Issues	1 (1 resolved)

Scope

OpenZeppelin audited the Uniswap/calibur repository at commit 732a20a.

In scope were the following files:

```
src
├── BaseAuthorization.sol
├── EIP712.sol
├── ERC1271.sol
├── ERC4337Account.sol
├── ERC7201.sol
├── ERC7739.sol
├── ERC7821.sol
├── ERC7914.sol
├── KeyManagement.sol
├── MinimalDelegation.sol
├── MinimalDelegationEntry.sol
├── Multicall.sol
├── NonceManager.sol
├── interfaces
│   ├── IEIP712.sol
│   ├── IERC1271.sol
│   ├── IERC4337Account.sol
│   ├── IERC7201.sol
│   ├── IERC7821.sol
│   ├── IERC7914.sol
│   ├── IExecutionHook.sol
│   ├── IHook.sol
│   ├── IKeyManagement.sol
│   ├── IMinimalDelegation.sol
│   ├── IMulticall.sol
│   ├── INonceManager.sol
│   └── IValidationHook.sol
└── libraries
    ├── BatchedCallLib.sol
    ├── CallLib.sol
    ├── CalldataDecoder.sol
    ├── ERC7739Utils.sol
    ├── EntrypointLib.sol
    ├── HooksLib.sol
    ├── KeyLib.sol
    ├── ModeDecoder.sol
    ├── PersonalSignLib.sol
    ├── SettingsLib.sol
    ├── SignedBatchedCallLib.sol
    ├── Static.sol
    ├── TransientAllowance.sol
    └── TypedDataSignLib.sol
```

Update: *All resolutions and the final state of the audited codebase mentioned in this report are contained in the* `v1.0.0` *release. This release has two extra files in addition to the above scope —* `libraries/PrefixedSaltLib.sol` *and* `libraries/WrappedSignatureLib.sol`*. Moreover,* `MinimalDelegation.sol` *was renamed to* `Calibur.sol`*.*

System Overview

Calibur is a new smart contract wallet made by the Uniswap team. It is a non-upgradeable EIP-7702 compliant wallet that provides a comprehensive set of features for secure transaction execution and key management. It enables the following functionalities:

- **Direct Execution**: The EOA and specified admin keys can directly execute transactions.
- **Delegated Execution**: Transactions can be executed by anyone with a valid signature from an authorized key.
- **Batched Transactions**: Multiple calls can be executed atomically with configurable failure modes.
- **Account Abstraction**: Supports ERC-4337 `UserOperations` for gasless transactions.
- **Signature Validation**: Supports ERC-7739 `isValidSignature` for smart contract signature validation.
- **Native Token Approval and `transferFrom`**: Supports ERC-7914 native token approval and `transferFrom` standard.

Any EOA can delegate itself to this contract and become a smart wallet. After the delegation is complete, the EOA can register additional keys that act on behalf of the wallet. Some of these keys can be made `admin` which gives them power to carry out sensitive functionalities. Each key is either a Secp256k1, a P-256, or a WebAuthn key. The EOA is known as the root key.

This wallet uses hooks at various places, before and after the execution or signature validation flow. A hook can be added to each registered key's setting by an `admin` to provide additional finer-grained validation logic. For instance, a hook can revert if the associated key is not permissioned to transfer above a certain amount out of a particular token. Each key can have a different hook and expiry timestamp.

Security Model and Trust Assumptions

Several trust assumptions were made during the review of this contract:

- It is assumed that the `admin` keys only register trusted keys and update each key's settings with the appropriate expiry date and hook address. The security of the wallet depends on the hook setting for each key. The hooks are expected to add extra layers of fine-grained validation against unauthorized actions. Without hooks, this wallet behaves like 1/n multisig, where n is the number of registered accounts.
- It is assumed that the `admin` keys do not approve native tokens to untrusted spenders.
- It is assumed that all registered keys would not sign malicious transactions.
- All inherited libraries and out-of-scope contracts are assumed to work as intended.
- The assumptions made by the `WebAuthn` library are trusted to be appropriate for this use case.
- The *WebAuthn* registration ceremony is assumed to have happened correctly.
- The webAuth relying party is assumed to require `userVerification` for registration and assertion.
- Callers of `isValidSignature` are assumed to ensure that the validated signature is not replayable (i.e., the original struct contains some form of nonce) as the `isValidSignature` is a `view` function and the same signature can be valid repeatedly.
- Only public keys associated with trusted relying parties are added to the wallet.
- WebAuthn, traditionally used in web2 for passwordless authentication with random challenges to prevent replay attacks, has been adapted to sign structured, transaction-related digests. However, the wallet's integration with webAuthn deviates from its typical use case as browser session logins. The special trust assumption here is on the relying party to clearly display the message on their user interface to allow for proper signing. In particular, it is trusted that the relying party is not compromised and that the hashed message can be manually checked by the user before signing.

Privileged Roles

There are three types of keys — the root key, admin keys, and registered non-admin keys. Each key can be one of the following types: Ethereum address, standalone P-256 key, or a webAuthnP256 key. It is worth noting that different keys may have different default security properties depending on where they are used and stored.

- The `root` key is the address of the wallet itself. It is always an Ethereum address.
- The `admin` keys are set by the `root` key or other `admins`. It can be of any type.
- The `root` and `admin` keys can:
  - call the `execute(BatchedCall memory batchedCall)` and `execute(bytes32 mode, bytes memory executionData)` functions
  - call the contract itself. This allows them to call the `setERC1271CallerIsSafe`, `updateEntryPoint`, `approveNative`, `approveNativeTransient`, `register`, `update`, `revoke`, and `invalidateNonce` functions
  - Behave like a registered key
- A registered key or a non-admin key can be set by the `root` key or an `admin` key. They can be of any key type and are able to:
  - sign a transaction for the `execute(SignedBatchedCall memory signedBatchedCall, bytes memory wrappedSignature)`, `validateUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 missingAccountFunds)`, and `isValidSignature` functions

Medium Severity

M-01 The `from` Address in ETH_CALL Can Be Non-Zero

The `isValidSignature` function, defined in EIP-1271, allows smart contracts to issue signatures in the same way as EOAs. In the `MinimalDelegation` contract, signature verification is performed differently depending on the type of caller:

- Direct verification of the signature, if the caller is whitelisted.
- Verification using the `PersonalSign` workflow defined in ERC-7739, if the call is of off-chain origin.
- Verification using the `TypedDataSign` workflow, also defined in ERC-7739 for all other cases.

However, the function assumes that an off-chain call (done using `ETH_CALL` RPC method) will always have the sender as `address(0)`. However, the `ETH_CALL` RPC method allows users to specify any optional `from` address. This would result in the execution flow passing to either of the two branches: the `erc1271CallerIsSafe` branch or the `_isValidTypedDataSig` branch. If the execution enters the first branch, a proper rehashing does not happen and the signatures of the same owner can be reused for other wallets. If the code execution enters the second branch, it would likely revert due to a decoding failure.

Currently, there is no reliable way to tell whether a call has been made off-chain. The incorrect assumption made in the `isValidSignature` function causes the contract to behave in undesirable ways. As such, consider using the `Signature Verification Workflow Deduction` method from ERC-7739 to mitigate this issue.

Update: *Resolved in pull request #187. The Uniswap team stated:*

> *This logic is removed in the latest versions, we do not special case the ETH_CALL flow.*

M-02 Hooks Do Not Receive All Relevant Data

A hook is a contract that is associated with a key's setting and is called when the key performs certain actions. Different functions can be called on a hook for different types of actions performed by the key. This allows for additional fine-grained complex logic for access control. For instance, the `handleAfterValidateUserOp` function is called in the `validateUserOp` function that can do further validation with the `keyHash, userOp, userOpHash, hookData` input.

However, when the `handleAfterValidateUserOp` function is called, an important parameter, `validationData`, is not passed to the hook. This packed variable contains the important `validAfter`, `validUntil`, and `success` values. Not having this information may hinder the hook from being able to develop efficient and complex logic.

Similarly, the `handleAfterExecute` is neither passed the `output` nor any value indicating whether the call succeeded.

Consider adding additional parameters to the hook interface for the above-mentioned functions.

Update: *Resolved in pull request #217.*

M-03 ERC-7739 Signature Verification Workflow Deviation

ERC-7739 specifies that the implementation MUST deduce the type of workflow by parsing the `signature` argument of the `isValidSignature` function:

> If the signature contains the correct data to reconstruct the hash, the isValidSignature function MUST perform the TypedDataSign workflow. Otherwise, the isValidSignature function MUST perform the PersonalSign workflow.

However, ERC-7739 does not distinguish the origin of the transaction as off-chain or on-chain. In the case of an ERC-7739-compliant off-chain, nested, typed data signature, it will be wrongly interpreted as a NestedPersonalSign signature and, thus, will fail the validation.

In the `isValidSignature` function, consider following the specified ERC-7739 signature verification workflow that prioritizes interpreting the data as `TypedDataSign` before interpreting it as `PersonalSign`.

Update: *Resolved in pull request #217.*

M-04 `requireUV` Flag Should Not Be Hardcoded to `false`

Whenever a webAuthn registration or authentication happens, the relying party can specify whether they want the authenticator to verify the user. User verification happens through biometrics (face recognition or fingerprint scanning), pin (*Yubikey*), or passwords (*1password*). Without user verification, it is not certain that the credential owner verified the transaction.

The relying party instructs the authenticator by passing `required`, `preferred`, or `discouraged` as the `userVerification` param. The authenticator then returns the signed `AuthenticatorAssertionResponse` object. This object contains the `User Verified` flag which tells the relying party whether the user was verified or not.

To prevent unauthorized access, the `requireUV` flag should be hardcoded to `true` instead of `false`. This would allow the `WebAuthn` library to check that the user was verified by the authenticator. This approach is used by the Safe Wallet as well.

To prevent unauthorized access, consider always ensuring that the authenticator verified the user during signature verification.

***Update:** Acknowledged, not resolved. The Uniswap team stated:*

> *Thanks for the suggestion. We are leaning no fix here because we believe that UV can be enforced by the client: - if requireUV is true, authenticationData MUST contain the flag; - if false, its not checked in the contract, client can always set flag to true and do its own security checks.*

Low Severity

L-01 Domain Separator Does Not Include Delegate Address

The minimal-delegation contract is meant to be delegated to by an EOA account via EIP-7702. The EIP-712 compliant domain separator contains, among other things, the chainID and the verifying contract as `address(this)`. Hence, the verifying contract is the EOA account address and does not include the delegated implementation address.

This is not a problem until later, in the improbable event, when the same EOA account re-delegates to a different account with the same domain, including name and version. The domain separator would be the same for these two different delegations since the verifying contract has the same address. In such a case, there is a risk that previously signed signatures can be replayed if the deadlines, nonce, or storage migration are not properly managed.

To ensure that the verifying contract is specific to each EOA and its delegate contract, consider including the address of the delegate contract in the domain. One option is to make use of the salt parameter in the EIP-712 domain.

Update: *Resolved in pull request #190.*

L-02 Missing Zero-Address Checks

Throughout the codebase, two instances of missing zero-address checks were identified:

- Potential approval of native tokens to the zero `spender` address: While approving the zero address to spend tokens does not directly lead to a loss of funds, it serves no practical purpose and can be an accidental operation.
- Potential transfer of native tokens to the zero `recipient` address: This action, if executed, results in the permanent loss of tokens, as tokens sent to the zero address are irretrievable.

Consider performing a zero-address check in the above-mentioned addresses and consider reverting in such cases to prevent accidental loss of value.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *We are going to keep the current behavior as is, there are libraries for ERC20s (solady namely) which allow for sending tokens to the zero address and we'd like to keep that*

L-03 Unrestricted Admin Keys

Any admin key can register other keys and update the settings for any account, including itself, overwriting any previous settings set by other admins or the root key. For instance, if the root key has set an expiry date for a particular admin, this admin can self-update to never expire.

Since any registered key can sign a transaction on behalf of the account, the security of the account behaves like a 1-of-n multisig wallet, where `n` is the number of registered accounts that can increase exponentially since any admin key can grant further admin keys.

Consider clarifying the overwriting behavior of admin keys on existing settings, particularly for admin keys. To further enhance the security of the wallet, consider adding strategies to limit admin keys, for instance, by setting a maximum number of accounts, implementing recommended hook patterns, or restricting the power of assigning admin keys to only the root key.

Update: *Partially resolved in pull request #222.*

L-04 Standard P-256 Signing Algorithms May Not Be Compatible With keccak256 Hashing Scheme

Most current signing algorithms with P-256 elliptic curve use the SHA-256 hashing function as recommended by the RFC9053 standard (section 2.1). For instance, if one signs a message with the widely supported ES256 algorithm, the message will first be hashed using the `sha256` hashing function and then signed with a P-256 private key using the ECDSA algorithm.

This is widely supported by most authenticators or passkeys. The combination of P-256 curve with keccak256 hash function using ECDSA signing algorithm is not common as of the time of this writing. Thus, to recover such P-256 keys from a signature signed by standard algorithms (such as those in COSE algorithms), one needs to have the message hashed by `sha256`.

Hence, it is required for any P-256-type key to directly sign a 32-byte raw digest hashed with the keccak256 scheme. In most cases, this is possible by using the authenticator's API to request the signing of a raw 32-byte hash instead of one with an arbitrary number of bytes. If a user intends to use a P-256-type key to sign a raw digest, it is recommended to manually validate that the signed data corresponds to the final hash to avoid blind signing and mitigate phishing risk.

To raise awareness and follow the best practices for P-256-type private keys, consider adding further documentation to remind users and dApp developers of the ways in which the behavior of this wallet deviates from that of standard wallets that use Secp256k1-type private keys.

Update: *Resolved in pull request #208.*

L-05 Different Pragma Directives Are Used

In order to clearly identify the Solidity version with which the contracts will be compiled, pragma directives should be fixed and consistent across file imports.

Throughout the codebase, multiple instances of floating pragma directives were identified:

- `ERC4337Account.sol` has the pragma directive `pragma solidity ^0.8.29;` and imports the file `BaseAuthorization.sol`, which has a different pragma directive.
- `ERC7739Utils.sol` has the pragma directive `pragma solidity ^0.8.20;` and imports the file `PersonalSignLib.sol`, which has a different pragma directive.

Consider using the same, fixed pragma directive in all the files.

***Update:** Resolved at commit 2f6b281.*

L-06 Missing Event Emissions

Throughout the codebase, multiple instances of functions updating the state without an event emission were identified:

- The `setERC1271CallerIsSafe` function in `ERC1271.sol`
- The `update` function in `KeyManagement.sol`
- The `_useNonce` function in `NonceManager.sol`

Consider emitting events whenever there are state changes to facilitate off-chain monitoring and future migration of storage variables.

***Update:** Resolved in pull request #183. The Uniswap team stated:*

> *Fixed. We decided to only emit events for updates to private member variables to save gas and bytecode size*

L-07 `revertOnFailure` Does Not Consider Hooks Revert

The code comment for the `beforeExecutionHook` function states that the function is expected to revert if the execution reverts. However, when `revertOnFailure` is set to be `false` (i.e., failed calls are not expected to revert), the execution will revert from `beforeExecutionHook` instead.

Consider clarifying the hook's revert behavior in the comments and making any required changes to the code to be in accordance with the comment.

Update: *Resolved in pull request #222.*

L-08 Misleading Docstring

This docstring states that

> by default, the EntryPoint is the zero address, meaning this must be called to enable 4337 support.

The `_CACHED_ENTRYPOINT` is the zero address by default. However, when calling the ENTRY_POINT() function, it first checks if the cached entry point has been overridden. This is determined by the most significant bit of the cached entry point. By default, this is zero, which means that it is not overridden. Thus, the entry point static address, 0x4337084D9E255Ff0702461CF8895CE9E3b5Ff108 is returned. Hence, by default ERC-43777 support is active.

Consider updating the docstring to indicate the correct behavior.

Update: *Resolved in pull request #182.*

L-09 A Newly Registered Key Has No Expiration Date

Keys registered through the `register` function are created without an expiration date, leading to permanent authorization unless explicitly modified. This violates the principle of least privilege and could result in security risks if administrators forget to set expiration dates.

When a new key is registered via the `register` function in `KeyManagement.sol`, the key is stored in the `keyStorage` mapping and added to the `keyHashes` set. However, no corresponding entry is created in the `keySettings` mapping. Due to the implementation in `SettingsLib.sol`, a key with no settings (that is, a zero value) is interpreted as having no expiration

Possible mitigations for this issue include the following:

- Requiring settings during registration.

- Using the maximum value for no expiry.
    - `uint(40).max` is roughly 34500 years.
- Set a default expiration date (say, 30 days).

Consider implementing the aforementioned mitigations to prevent newly registered keys from having no expiration date.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *Acknowledged, we are leaning towards not fixing this as it is an integration issue and we will be sure to prevent it from happening offchain.*

L-10 `keyhash` and `hookdata` Should Be Part of the Signed Hash

In the `MinimalDelegation` contract, the `keyHash` and `hookData` variables are not part of the signed hash but are directly appended to the `bytes memory signature` in three functions:

- `validateUserOp`
- `_handleVerifySignature` (`keyHash` is included in this one)
- `isValidSignature`

In functions that rely on signature verification to ascertain the origin of the calldata, any variable that is not part of the signed hash risks being manipulated by a relayer. This can result in the `hook` reverting, passing wrongly, or storing the wrong state.

Passing unsigned variables can be prevented by taking the following measures:

- In `validateUserOp`, both variables can be encoded in PackedUserOperation.calldata instead of PackedUserOperation.signature.
- The hookdata for the `_handleVerifySignature` function can be packed in the `signedBatchedCall` struct.
- There is no simple way to include both the `keyHash` and the `hookdata` in the signed hash inside the `isValidSignature` function without deviating from EIP-7739.

Where possible, consider packing `keyHash` and `hookdata` in the signed hash and figuring out the best way to do the same for the `isValidSignature` function.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *Thanks for the suggestion, we are leaning no fix here because: - mitigated by `executor` in SignedBatchedCall; - already exists DoS vector from malicious relayer because they could not send the signature; - don't want to only include keyHash in digest for 2/3 use cases (can't do for 1271).*

L-11 Deviation from ERC-7821

The `MinimalDelegation` smart contract implementation deviates from the ERC-7821 standard in its handling of execution modes, potentially impacting interoperability and compliance. According to ERC-7821, smart contracts should support the following modes:

- **Mode 1 (mandatory)**: Basic single batch (`0x01000000000000000000...`)
  - Does not support optional operation data
- **Mode 2 (mandatory)**: Enhanced single batch (`0x01000000000078210001...`)
  - Supports optional operation data
- **Mode 3 (optional)**: Batch of batches (`0x01000000000078210002...`)

However, the current implementation in `ERC7821.sol` has the following deviations:

- Only implements the first mandatory mode
- Does not support the second mandatory mode for optional operation data
- Introduces a non-standard mode in `ModeDecoder.sol` that is not defined in the EIP

Consider either correcting or documenting the deviations from ERC-7821 to enhance code clarity and maintainability.

Update: *Resolved in pull request #212.*

L-12 ERC-7914 Is Not Published

ERC-7914 is not published and was being actively reviewed during the course of this review. There is a high probability that it can change in the future, making the implementation of this contract deviate from the spec. In addition, this ERC seems to have a conflicting number with another ERC and this may result in its number being changed in the future.

This implementation differs from the current ERC spec in several ways:

- The `supportsInterface` function is not implemented.

- The `transferFromNative` function reverts on failure instead of returning false according to the ERC. The ERC spec for the `transferFromNative` function states that the function *MUST handle the Native transfer and return false on failure.*
- The `transferFromNativeTransient`, `allowance`, and `transientAllowance` functions are not defined in the ERC.
- In the `transferFromNative` function, the function returns `true` early when `amount == 0`, whereas in the reference implementation, the function returns `false` for the same case.

To enhance code clarity and prevent confusion, consider adding a link to the ERC spec and the expectation for further changes in the docstring.

Update: *Resolved in pull request #212.*

L-13 Risk of Blind Signing With WebAuthn

WebAuthn is a web2 standard that is often used for passwordless authentication in browsers. The website (relying party) generates a random `challenge` during registration or login that is signed by a private key associated with a user's authenticator. This random `challenge` serves as a nonce to prevent replayability (*Cryptographic Challenges* section).

In web3 applications, the `challenge` to be signed is most likely not random, but represents some properties of a blockchain transaction. In this wallet, the digest to be signed is associated with either the `typedDataSign` or `personalSign` from the ERC-7739 standard used inside the isValidSignature function, or to batched calls that are to be executed on behalf of the account.

As an extension of EIP-712 and EIP-1271, ERC-7739 sets a standard for the `digest` to be more structured and readable. However, as the time of this writing, most authenticators do not show the final digest on-device due to its security model, not to mention any readability regarding the structured data being signed. The authenticators are not, yet, designed to integrate with signing blockchain structured data, which can be used to execute transactions and move funds. In the web3 use case, transparency is of great security importance as demonstrated by the recent Bybit hack in 2025. Hence, this requires the relying party to be secure and clearly display the message requested to be signed before calling the authenticator's API with the final `challenge`.

Consider adding further documentation and warnings regarding the risk of blind signing with the webAuthn key type. In addition, consider encouraging further standard development for decentralized apps integrating webAuthn with blockchain use cases to promote clear signing, transparency, and security.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *Acknowledged. We will be sure to communicate these findings with our other teams working on the webauthn / client components*

L-14 Potential Early Return

Currently, repeated registration of the same key is not rejected. Instead, the supplied key is updated in the key storage and added to the `keyHash` set.

For improved efficiency, consider reverting early when registration of already registered keys is attempted.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *It adds overhead in the normal valid use case so we are not fixing.*

Notes & Additional Information

N-01 Lack of Security Contact

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 quite 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 their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are no contracts that have a security contact. Consider adding a NatSpec comment containing a security contact above each contract definition. 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 #212.*

N-02 State Variable Visibility Not Explicitly Declared

Throughout the codebase, multiple instances of state variables lacking an explicitly declared visibility were identified:

- In `KeyManagement.sol`, the `keyHashes` state variable
- In `KeyManagement.sol`, the `keyStorage` state variable
- In `KeyManagement.sol`, the `keySettings` state variable
- In `NonceManager.sol`, the `nonceSequenceNumber` state variable

For improved code clarity, consider always explicitly declaring the visibility of state variables, even when the default visibility matches the intended visibility.

Update: *Resolved in pull request #195.*

N-03 Optimizable State Reads

The storage read for the `_CACHED_ENTRYPOINT` variable in `ERC4337Account.sol` could be optimized by caching it into a `memory` variable.

Consider reducing SLOAD operations that consume unnecessary amounts of gas by caching the values in a `memory` variable.

Update: *Resolved in pull request #194.*

N-04 Unused Imports

Throughout the codebase, multiple instances of unused imports were identified:

- The import `import {TypedDataSignLib} from "./libraries/TypedDataSignLib.sol";` imports unused alias `TypedDataSignLib` in `ERC7739.sol`.

- The import `import {PersonalSignLib} from "./libraries/PersonalSignLib.sol";` imports unused alias `PersonalSignLib` in `ERC7739.sol`.
- The import `import {IHook} from "./IHook.sol";` imports unused alias `IHook` in `IKeyManagement.sol`.
- The import `import {Call} from "../libraries/CallLib.sol";` imports unused alias `Call` in `IMinimalDelegation.sol`.
- The import `import {Settings, SettingsLib} from "./SettingsLib.sol";` imports unused alias `Settings` and `SettingsLib` in `KeyLib.sol`.
- The import `import {Key, KeyLib, KeyType} from "./libraries/KeyLib.sol";` imports unused alias `KeyType` in `KeyManagement.sol`.
- The import `import {IERC4337Account} from "./interfaces/IERC4337Account.sol";` imports unused alias `IERC4337Account` in `MinimalDelegation.sol`.
- The import `import {Key, KeyLib, KeyType} from "./libraries/KeyLib.sol";` imports unused alias `KeyType` in `MinimalDelegation.sol`.
- The import `import {Static} from "./libraries/Static.sol";` imports unused alias `Static` in `MinimalDelegation.sol`.

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

Update: *Resolved in pull request #185.*

N-05 Missing `view` Keyword

Within the `IERC7914.sol` interface, the following functions are missing the `view` keyword:

- The `allowance` function
- The `transientAllowance` function

Consider adding the `view` keyword to the aforementioned functions in the `IERC7914.sol` interface.

Update: *Resolved in pull request #186.*

N-06 Potential Malformed Calldata

Within `CalldataDecoder.sol`, the `removeSelector` function subtracts 4 from `data.length` in inline assembly without verifying that `data.length` is at least 4. If the input calldata is shorter than 4 bytes, this subtraction will underflow, resulting in an invalid length, and potentially causing unintended behavior at runtime. This function is used to extract execution data in the `UserOp` struct and malformed calldata will likely result in a revert.

Consider validating data length to avoid unexpected behavior.

Update: *Resolved in pull request #209.*

N-07 Naming Suggestions

As ERC-7914 is still being actively worked on, the below two functions should be renamed and the same should be done in the upstream ERC.

- `allowance` => `nativeAllowance`
- `transientAllowance` => `transientNativeAllowance`

Consider renaming the above-mentioned functions to enhance code clarity and make a distinction from the existing notion of ERC-20 `allowance`.

Update: *Resolved in pull request #192.*

N-08 Typographical Error

Within `HooksLib.sol`, `0x000000000000000000000000000000000000000a` is equivalent to the binary value `1010` and not `0001 1010`.

Consider correcting the aforementioned typographical error to improve code clarity and maintainability.

Update: *Resolved in pull request #212.*

N-09 EIP-7913 Standard On Signature Verification

The recently proposed EIP-7913 standard is highly relevant to this smart wallet implementation. EIP-7913 specifies a standard verifier interface for different key types that are not necessary Ethereum addresses. The proposed `verify` function resembles `KeyLib.verify`:

```
function verify(bytes calldata key, bytes32 hash, bytes calldata signature) external
view returns (bytes4);
```

For future compatibility, consider being an early adopter of the EIP-7913 standard.

Update: *Acknowledged, not resolved.*

N-10 Missing ERC-7739 Recommended contentsName Validation

In ERC-7739, it is recommended to perform further validation of the parsed content name.

> For safety, it is RECOMMENDED to treat the signature as invalid if any of the following is true:
>
> - contentsName is the empty string (i.e. bytes(contentsName).length == 0).
> - contentsName starts with any of the following bytes abcdefghijklmnopqrstuvwxyz(.
> - contentsName contains any of the following bytes ,)\x00.

Lowercase-character validation is missing from the codebase. Consider adding further validation or documenting its absence appropriately.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *Acknowledged, the spec says that rejecting lowercase is recommended not enforced, and I don't see this in the implementation here.*

Client Reported

CR-01 Missing Signature Deadlines

When validating signatures, the recovered key's settings struct is checked for the expiry timestamp. Thus, a signature is valid as long as the associated key is not expired. This deviates from the usual practice of allowing a deadline for each signature even if the key is not expired.

Consider adding a deadline for each signature on top of the current expiry-able key setup to allow for fine-grained permission control over signature use and prevent further risk of signature replay.

Update: *Resolved in pull request #169.*

Conclusion

The audited codebase implements a non-upgradeable, EIP-7702-compliant smart contract wallet, featuring batched transaction execution, custom extensibility via hooks and key management, and integration with non-Ethereum-based key types such as raw P-256 and webAuthn keys.

The security model relies on the correct use of trusted keys and external systems to prevent unauthorized or replayable transactions. The codebase was found to be well-written, supporting multiple ERC standards for interoperability. We appreciate the Uniswap team for their responsiveness and support throughout the audit.