Minimal delegation

Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Cccz, Security Researcher

May 10, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Execute calls can be front-run . . . 4
  - 3.1.2 Execute calls can be forced to fail with an out of gas error . . . 4
- 3.2 Low Risk . . . 5
  - 3.2.1 Potential double-counting allowance risk . . . 5
  - 3.2.2 Off-by-one issue in `isExpired()` . . . 5
  - 3.2.3 Potential privilege escalation in nonce management . . . 6
  - 3.2.4 `_checkExpiry` will revert if the signature is expired in `validateUserOp` . . . 6
- 3.3 Informational . . . 7
  - 3.3.1 `ModeDecoder` implements a subset of EIP-7821 . . . 7
  - 3.3.2 Incorrect comment . . . 7
  - 3.3.3 `HookData` is not included in the signature digest . . . 8
  - 3.3.4 `MinimalDelegation` EntryPoint compatibility . . . 8
  - 3.3.5 Use of unlicensed smart contracts . . . 10
  - 3.3.6 When transferring 0 amount, ERC7914 is better to return `true` . . . 10
  - 3.3.7 `_execute()` may call `handleAfterExecute()` on stale hook . . . 11
  - 3.3.8 `validateUserOp()` should not return early when the signature is invalid . . . 11

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

From Apr 15th to Apr 19th the Cantina team conducted a review of minimal-delegation on commit hash 732247c5. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	1	1
Medium Risk	0	0	0
Low Risk	4	2	2
Gas Optimizations	0	0	0
Informational	8	7	1
Total	**14**	**10**	**4**

3 Findings

3.1 High Risk

3.1.1 Execute calls can be front-run

Severity: High Risk

Context: MinimalDelegation.sol#L66

Description: The function

```
function execute(SignedBatchedCall memory signedBatchedCall, bytes memory wrappedSignature) public payable
```

implemented in the `MinimalDelegation` contract is publicly callable, enabling any external address to invoke it if a valid signature is provided. This implementation allows anyone to front-run any `execute` call as the code simply checks the signature and does not confirm the identity of the caller. Since there is no field for the intended executor address in the signed digest, any party that obtains the signature can submit it first.

A very detrimental scenario could be a malicious user supplying no Ether (e.g., `msg.value == 0`) in a front-run `execute` call that was supposed to use it, potentially forcing part of the batched calls to revert. If `signedBatchedCall.shouldRevert = false`, the attacker can easily break the intended call flow. Meanwhile, the legitimate user's subsequent call will revert because the same signature and nonce have already been consumed.

Recommendation: Incorporate an intended executor or caller address into the signature's domain data to ensure only the authorized caller can use that signature. Specifically:

1. Introduce a field (e.g. `executor`) in the struct that is hashed and checked in the EIP-712 domain.
2. If `executor` is not zero, verify at runtime `require(msg.sender == executor)`; if zero, fallback to the existing "anyone can execute" logic if truly intended.

Uniswap: Fixed in PR 150.

Cantina Managed: Fix verified.

3.1.2 Execute calls can be forced to fail with an out of gas error

Severity: High Risk

Context: MinimalDelegation.sol#L66

Description: In the `execute(SignedBatchedCall memory signedBatchedCall, bytes memory wrappedSignature) public payable` flow, a malicious user can specify a gas limit for the overall transaction that is large enough for the "high-level" portion of the `execute` call to succeed but leaves insufficient gas for the low-level call performed in `_dispatch` → `_execute`, where:

```
(success, output) = to.call{value: _call.value}(_call.data);
```

Due to the EIP-150 "63/64 gas" rule, only 63/64 of the remaining gas is forwarded to a subcall. If the subcall fails for insufficient gas and `signedBatchedCall.shouldRevert == false`, the entire batch may partially complete with no revert, thus forcing the intended function call to fail. As a result, the user's signed batch is sabotaged by the attacker controlling the available gas, while the high-level transaction still succeeds consuming the signature's nonce.

Recommendation: Consider following OpenZeppelin's MinimalForwarder.sol implementation.

Another alternative is storing a minimum intended gas limit in the signed data: if the `execute` call is given a gas limit below that limit, revert.

Uniswap: Acknowledged. we lean towards not fixing this because:

- Introducing the `executor` field in `SignedBatchedCall` allows for users to require a specific relayer for their actions which mitigates the severity of this attack.

- We believe that it would be relatively easy to detect when this is happening, and that the impact would be relatively small for users (if you are sending a batch which can partially revert, you should be aware of all of the possibilities).

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 Potential double-counting allowance risk

Severity: Low Risk

Context: ERC7914.sol#L14-L25

Description: The `MinimalDelegation` wallet supports two forms of native allowances:

1. Persistent approvals: stored in regular contract storage via `approveNative` and `transferFromNative`.
2. Transient approvals: granted by `approveNativeTransient` and used by `transferFromNativeTransient`, which reset each transaction.

Because the code separately tracks ephemeral and persistent allowances, a user's total effective approval can unintentionally stack. For example, if the user or the contract sets a persistent allowance of 1000 and then separately grants a transient allowance of 1000, the spender might see an aggregate of 2000. This could exceed the user's intended limit and lead to accidental over-spending.

Recommendation: Consider disallowing calling `approveNativeTransient` if the persistent allowance is already non-zero for that `spender`. Another option is simply informing the users that transient and persistent allowances are cumulative and stating this in the user wallet documentation.

Uniswap: Acknowledged. We lean towards not fixing this because:

- If there exists a persistent allowance AND a transient one, we don't believe that it is a double spend because there exists two intentionally set and distinct allowances for the spender.
- We expect any reasonably written spender contract to be explicit about which allowance it is requesting (persistent or transient), and to only use those amounts when needed.

Cantina Managed: Acknowledged.

3.2.2 Off-by-one issue in `isExpired()`

Severity: Low Risk

Context: SettingsLib.sol#L48

Description: When the protocol checks if the setting is expired, it considers `expiration == block.timestamp` to be invalid:

```
function isExpired(Settings settings) internal view returns (bool _isExpired, uint40 _expiration) {
    uint40 _exp = expiration(settings);
    if (_exp == 0) return (false, 0);
    return (_exp <= block.timestamp, _exp);
}
```

While in `validateUserOp()`, `expiration` is used as `validUntil` in `validationData`. And in EntryPoint, `validUntil == block.timestamp` is considered valid.

```
function _getValidationData(
    uint256 validationData
) internal virtual view returns (address aggregator, bool outOfTimeRange) {
    if (validationData == 0) {
        return (address(0), false);
    }
    ValidationData memory data = _parseValidationData(validationData);
    // solhint-disable-next-line not-rely-on-time
    outOfTimeRange = block.timestamp > data.validUntil || block.timestamp <= data.validAfter;
    aggregator = data.aggregator;
}
```

Recommendation: It is recommended to change `isExpired()` as follows to treat `expiration == block.timestamp` as valid.

```
  function isExpired(Settings settings) internal view returns (bool _isExpired, uint40 _expiration) {
      uint40 _exp = expiration(settings);
      if (_exp == 0) return (false, 0);
-     return (_exp <= block.timestamp, _exp);
+     return (_exp < block.timestamp, _exp);
  }
```

Uniswap: Fixed in PR 147.

Cantina Managed: Fix verified.

3.2.3 Potential privilege escalation in nonce management

Severity: Low Risk

Context: MinimalDelegation.sol#L156-L159

Description: In `NonceManager`, all `keyHashes` use the same underlying nonce mapping:

```
function _useNonce(uint256 nonce) internal {
    uint192 key = uint192(nonce >> 64);
    uint64 seq = uint64(nonce);
    if (!(nonceSequenceNumber[key]++ == seq)) {
        revert InvalidNonce();
    }
}
```

And by design, only `keyHashes` with admin privileges can invalidate any nonce by `invalidateNonce()`, but `keyHashes` without admin privileges can sign any `nonceKey` to invalidate any nonce, which can be used in the front-run attack to invalidate execute calls from other `keyHashes`.

```
function invalidateNonce(uint256 newNonce) external onlyThis {
    uint192 key = uint192(newNonce >> 64);
    uint64 currentSeq = uint64(nonceSequenceNumber[key]);
    uint64 targetSeq = uint64(newNonce);
    if (targetSeq <= currentSeq) revert InvalidNonce();
    // Limit the amount of nonces that can be invalidated in one transaction.
    unchecked {
        uint64 delta = targetSeq - currentSeq;
        if (delta > type(uint16).max) revert ExcessiveInvalidation();
    }
    nonceSequenceNumber[key] = targetSeq;
    emit NonceInvalidated(newNonce);
}
```

Recommendation: It is recommended to separate the nonces for different `keyHashes`.

```
- mapping(uint256 key => uint256 seq) nonceSequenceNumber;
+ mapping(bytes32 => mapping(uint192 => uint256)) public nonceSequenceNumber;
```

Uniswap: Acknowledged. We lean towards not fixing this because:

- Admin keys can invalidate an entire `nonceKey` space, which is arguably a stronger DoS than a single value.
- There is plausible front-running risk from a malicious non admin key, but we believe that the user should simply remove that key from their account if detected.
- It is nice to be able to enforce strict ordering of nonces between different keys, which would be impossible if we segmented nonces by `keyHash`.

Cantina Managed: Acknowledged.

3.2.4 `_checkExpiry` will revert if the signature is expired in `validateUserOp`

Severity: Low Risk

Context: MinimalDelegation.sol#L142

Description: Within the `validateUserOp` flow, the contract verifies the signature and then calls `_checkExpiry(settings)`. If the key is expired, `_checkExpiry(settings)` unconditionally reverts with a `KeyExpired(expiry)` error. As a result, the entire operation fails with a revert rather than returning `SIG_VALIDATION_FAILED` as stated in the EIP-4337 spec.

Recommendation: Consider returning the signature failure code (e.g. SIG_VALIDATION_FAILED) if the key has expired instead of reverting in the `validateUserOp` function.

On the other hand, in the `isValidSignature` function `_checkExpiry(settings)` is also used. In this flow, consider setting `isValid` to `false` in case that the signature is expired.

Uniswap: Fixed in PR 154.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 `ModeDecoder` implements a subset of EIP-7821

Severity: Informational

Context: ERC7821.sol#L9-L16

Description: In `ModeDecoder` and `MinimalDelegation`, particularly in the `execute(bytes32 mode, bytes calldata executionData)` and `supportsExecutionMode(bytes32 mode)` functions, only recognizes two specific "batched" modes:

```
bytes32 constant BATCHED_CALL = 0x0100000000000000000000000000000000000000000000000000000000000000;
bytes32 constant BATCHED_CAN_REVERT_CALL = 0x0101000000000000000000000000000000000000000000000000000000000000;
```

and checks them via:

```
function isBatchedCall(bytes32 mode) internal pure returns (bool) {
    return mode == BATCHED_CALL || mode == BATCHED_CAN_REVERT_CALL;
}
```

This approach diverges from the EIP-7821 specification, which defines more granular modes. For example, `0x0100000000000000000...` or `0x0100000000078210001...` for single-batch with optional `opData` and a multi-batch approach `0x0100000000078210002...`. Additionally, `MinimalDelegation` does not parse any optional `opData` nor supports "batch of batches" recursion. Instead, it decodes only a single `Call[]` and toggles revert behavior on failure, ignoring the extended modes described in EIP-7821. This design is not incorrect from a security perspective, but it means the contract remains incompatible with the standard's optional features such as multi-level batches or specialized data for authentication.

Recommendation: Provide clarity in code comments or docs that only two batch modes are supported. If partial coverage is the final design choice, specify the rationale. Otherwise, upgrade the contracts to align with the full EIP-7821 spec.

Uniswap: Fixed in PR 149.

Cantina Managed: Fix verified.

3.3.2 Incorrect comment

Severity: Informational

Context: SettingsLib.sol#L9

Description: In the `SettingsLib` library, the comment claims:

```
/// The most significant 8 bits are reserved to specify if the key is an admin key or not.
/// 6 bytes |   1 byte       | 5 bytes           | 20 bytes
```

However, the code actually shifts by 200 bits (shr(200, settings)), using 56 bits (7 bytes) for the admin portion. This is inconsistent with the stated "8 bits" approach. Consequently, bits [200..255] become the "admin region," not [248..255].

Recommendation: Consider updating the comment as shown below:

```
/// The most significant 7 bits are reserved to specify if the key is an admin key or not.
/// 6 bytes |   1 byte       | 5 bytes           | 20 bytes
```

Uniswap: Fixed in PR 156.

Cantina Managed: Fix verified.

3.3.3 `HookData` is not included in the signature digest

Severity: Informational

Context: MinimalDelegation.sol#L127

Description: `MinimalDelegation` exposes `hookData` in calls to `handleAfterVerifySignature` and `handleAfterIsValidSignature` function, but does not incorporate `hookData` into the EIP-712 signature digest. Consequently, any external caller can supply arbitrary bytes in `wrappedSignature`:

```
(bytes32 keyHash, bytes memory signature, bytes memory hookData) = abi.decode(wrappedSignature, (bytes32,
↪  bytes, bytes));
```

and the contract then passes this untrusted `hookData` to the hook. If the hook logic expects `hookData` to be genuine, an attacker can cause reverts or trigger unexpected behaviors. For example, the attacker can supply malicious input that is decoded incorrectly in the hook, forcing the transaction to revert. Since `hookData` is not signed by the user's private key, an attacker can alter it and the signature would still be valid.

The final impact is really dependant on the `hook`'s final implementation.

Recommendation: Include `hookData` in the signed digest so if its updated by a malicious user the signature will become invalid. If the current implementation is kept, consider documenting that the `hookData` can be altered, so the hook should always be able to handle malicious data gracefully (e.g., ignoring it or only running safe logic).

Uniswap: Fixed in PR 149.

Cantina Managed: Fix verified.

3.3.4 `MinimalDelegation` EntryPoint compatibility

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: After reviewing the latest `EntryPoint` versions, the `MinimalDelegation` smart wallet can only be used with the v0.7.0 and v0.8.0 versions, being incompatible with the v0.6.0 as this version does not yet support the `executeUserOp` operations. The `executeUserOp` support was introduced in the v0.7.0 version.

EntryPoint version analysis:

- EntryPoint Version v0.6.0.
  - Key Changes:
    - New `userOpHash` algorithm.
    - Nonce managed by `EntryPoint`.
    - Prevent recursion of `handleOps`.
  - Description:
    - The new `userOpHash` algorithm introduces a unique hash for each `UserOperation`, enhancing security by preventing replay attacks.
    - Nonce managed by `EntryPoint` centralizes nonce management within the `EntryPoint` contract, simplifying the logic in account contracts and reducing the chance of errors.
    - The system prevents recursion of `handleOps`, adding a safeguard against recursive calls during operation handling, which helps avoid exploits or infinite loops.

  - `MinimalDelegation` Compatibility: Incompatible as it does not support the `executeUserOp` operation and uses a different `UserOperation` struct.

- EntryPoint Version v0.7.0:
  - Key Changes:
    * Simulation functions removed from deployed `EntryPoint`.
    * Added `delegateAndRevert()` helper.
    * Added ERC-165 "supportsInterface".
    * Added a sample `TokenPaymaster`.
    * Added a 10% penalty charge for unused execution gas limit.
  - Description:
    * Simulation functions removed from the deployed contract reduces complexity and deployment costs, with simulation now handled off-chain by bundlers.
    * The `delegateAndRevert()` helper assists bundlers in simulating transactions accurately, especially on networks without state override features.
    * ERC-165 "supportsInterface" enables interface detection, making the contract more interoperable with others.
    * A sample `TokenPaymaster` provides a reference for paymasters that accept tokens instead of ETH for gas payments, encouraging broader adoption.
    * A 10% penalty for unused execution gas encourages precise gas estimation to improve network efficiency.
  - `MinimalDelegation` Compatibility: Fully compatible but lacks support for EIP-7702 authorizations.

- EntryPoint Version v0.8.0:
  - Key Changes:
    * Native support for EIP-7702 authorizations.
    * Native support for ERC-712 based `UserOperation` hash and signatures.
    * Unused gas penalty adjustment (no penalty if below 40,000 gas).
    * Native Go implementation of ERC-7562 tracer.
    * Minor relaxations to ERC-7562 validation rules.
    * Bug fixes from AA Bug Bounty Program.
  - Description:
    * EIP-7702 support integrates with the latest account abstraction proposal, allowing for more flexible and secure account management.
    * ERC-712 support adopts standardized hashes and signatures for `UserOperations`, improving compatibility with external signers (like hardware wallets) and enhancing user experience.
    * The gas penalty adjustment eliminates penalties for unused gas below 40,000, making the system more user-friendly.
    * A native Go implementation of the ERC-7562 tracer improves performance and integration for ERC-4337 bundlers.
    * Relaxed ERC-7562 validation rules provide greater flexibility in processing `UserOperations`.
    * Bug fixes from the AA Bug Bounty Program address vulnerabilities, boosting security and reliability.
  - `MinimalDelegation` Compatibility: Fully compatible with support for EIP-7702 authorizations.

Recommendation: Avoid using the `MinimalDelegation` with the v0.6.0 version of the `EntryPoint`. The recommended version is the v0.8.0 as it includes support for EIP-7702 authorizations but v0.7.0 can also be used.

Uniswap: Fixed in PR 149.

Cantina Managed: Fix verified.

3.3.5 Use of unlicensed smart contracts

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: All the smart contracts in the codebase are currently marked as unlicensed, as indicated by the SPDX license identifier at the top of the file:

```
// SPDX-License-Identifier: UNLICENSED
```

Using an unlicensed contract can lead to legal uncertainties and potential conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project and could lead to legal issues in the future.

Recommendation: It is recommended to choose and apply an appropriate open-source license to the smart contract. Some popular options for smart contract projects are:

1. MIT License: A permissive license that allows for reuse with minimal restrictions.
2. GNU General Public License (GPL): A copyleft license that ensures derivative works are also open-source.
3. Apache License 2.0: A permissive license that provides an express grant of patent rights from contributors to users.

Uniswap: Fixed in PR 152.

Cantina Managed: Fix verified.

3.3.6 When transferring 0 amount, ERC7914 is better to return `true`

Severity: Informational

Context: ERC7914.sol#L28-L37

Description: When transferring 0 amounts, ERC7914 returns false, which indicates that the transfer failed. This makes ERC7914 behave like the ERC20 token that disallows 0 amount transfers, which has caused many integration issues. When transferring 0 amount, in a sense it succeeds even if we do nothing, so returning true makes sense and can avoid integration issues.

Recommendation:

```
  function transferFromNative(address from, address recipient, uint256 amount) external returns (bool) {
-     if (amount == 0) return false;
+     if (amount == 0) return true;
      _transferFrom(from, recipient, amount, false);
      emit TransferFromNative(address(this), recipient, amount);
      return true;
  }

  /// @inheritdoc IERC7914
  function transferFromNativeTransient(address from, address recipient, uint256 amount) external returns (bool)
  ↪ {
-     if (amount == 0) return false;
+     if (amount == 0) return true;
```

Uniswap: Fixed in PR 153.

Cantina Managed: Fix verified.

3.3.7 `_execute()` may call `handleAfterExecute()` on stale hook

Severity: Informational

Context: MinimalDelegation.sol#L109-L117

Description: In `_execute()`, it calls `handleAfterExecute()` directly on hook that cached before `to.call()`:

```
IHook hook = settings.hook();
bytes memory beforeExecuteData;
if (hook.hasPermission(HooksLib.BEFORE_EXECUTE_FLAG)) {
    beforeExecuteData = hook.handleBeforeExecute(keyHash, to, _call.value, _call.data);
}

(success, output) = to.call{value: _call.value}(_call.data);

if (hook.hasPermission(HooksLib.AFTER_EXECUTE_FLAG)) hook.handleAfterExecute(keyHash, beforeExecuteData);
```

An edge case is if `to.call()` calls `KeyManagement.update()` to update the setting, it may execute calls on stale hook:

```
function update(bytes32 keyHash, Settings settings) external onlyThis {
    if (keyHash.isRootKey()) revert CannotUpdateRootKey();
    if (!isRegistered(keyHash)) revert KeyDoesNotExist();
    keySettings[keyHash] = settings;
}
```

Recommendation:

```
  function _execute(Call memory _call, bytes32 keyHash) internal returns (bool success, bytes memory output) {
      // Per ERC7821, replace address(0) with address(this)
      address to = _call.to == address(0) ? address(this) : _call.to;

      Settings settings = getKeySettings(keyHash);
      if (!settings.isAdmin() && to == address(this)) revert IKeyManagement.OnlyAdminCanSelfCall();

      IHook hook = settings.hook();
      bytes memory beforeExecuteData;
      if (hook.hasPermission(HooksLib.BEFORE_EXECUTE_FLAG)) {
          beforeExecuteData = hook.handleBeforeExecute(keyHash, to, _call.value, _call.data);
      }

      (success, output) = to.call{value: _call.value}(_call.data);
+     settings = getKeySettings(keyHash);
+     hook = settings.hook();
      if (hook.hasPermission(HooksLib.AFTER_EXECUTE_FLAG)) hook.handleAfterExecute(keyHash, beforeExecuteData);
  }
```

Uniswap: Acknowledged. We lean towards not fixing this because:

- A key would have to be an admin key to update its settings (and add a new hook).
- We enforce that functions related to key state are only accessible through execute, which applied execution hooks + checks key expiry.
- Since the `afterExecute` hook function takes in the data returned from its corresponding `beforeExecute` hook call, we believe that there could be some edge cases where the hook could change after the low level call is made.

Cantina Managed: Acknowledged.

3.3.8 `validateUserOp()` should not return early when the signature is invalid

Severity: Informational

Context: MinimalDelegation.sol#L137-L153

Description: According to the specs, `validateUserOp()` shouldn't return early even if the signature is invalid for gas estimation.

> - SHOULD not return early when returning SIG_VALIDATION_FAILED (1). Instead, it SHOULD complete the normal flow to enable performing a gas estimation for the validation function.

But in `MinimalDelegation.validateUserOp()`, the invalid signature causes early return and does not execute the following code, this makes it inconsistent with the specs.

```
// If signature verification failed, return failure immediately WITHOUT expiry as it cannot be trusted
if (!isValid) {
    return SIG_VALIDATION_FAILED;
}

Settings settings = getKeySettings(keyHash);
_checkExpiry(settings);

/// validationData is (uint256(validAfter) << (160 + 48)) | (uint256(validUntil) << 160) | (success ? 0 : 1)
/// `validAfter` is always 0.
validationData = uint256(settings.expiration()) << 160 | SIG_VALIDATION_SUCCEEDED;

IHook hook = settings.hook();
if (hook.hasPermission(HooksLib.AFTER_VALIDATE_USER_OP_FLAG)) {
    // The hook can override the validation data
    validationData = hook.handleAfterValidateUserOp(keyHash, userOp, userOpHash, hookData);
}
```

And when the `userOp` is actually executed, the invalid signature will cause the transaction to revert in EntryPoint, so this will not cause the hook call to be actually executed.

```
function _validateAccountAndPaymasterValidationData(
    uint256 opIndex,
    uint256 validationData,
    uint256 paymasterValidationData,
    address expectedAggregator
) internal virtual view {
    (address aggregator, bool outOfTimeRange) = _getValidationData(
        validationData
    );
    if (expectedAggregator != aggregator) {
        revert FailedOp(opIndex, "AA24 signature error");
    }
```

Recommendation:

```
- if (!isValid) {
-     return SIG_VALIDATION_FAILED;
- }

  Settings settings = getKeySettings(keyHash);
  _checkExpiry(settings);

  /// validationData is (uint256(validAfter) << (160 + 48)) | (uint256(validUntil) << 160) | (success ? 0 : 1)
  /// `validAfter` is always 0.
  validationData = uint256(settings.expiration()) << 160 | SIG_VALIDATION_SUCCEEDED;

  IHook hook = settings.hook();
  if (hook.hasPermission(HooksLib.AFTER_VALIDATE_USER_OP_FLAG)) {
      // The hook can override the validation data
      validationData = hook.handleAfterValidateUserOp(keyHash, userOp, userOpHash, hookData);
  }
+ if (!isValid) {
+     return SIG_VALIDATION_FAILED;
+ }
```

Uniswap: Fixed in PR 154.

Cantina Managed: Fix verified.