// Security Assessment

04.28.2025 - 04.30.2025

Foundational Hooks - Custom Fee Uniswap V4 Hook

Renzo Protocol

HALBORN

Foundational Hooks - Custom Fee Uniswap V4 Hook - Renzo Protocol

Prepared by: HALBORN

Last Updated 05/05/2025

Date of Engagement: April 28th, 2025 - April 30th, 2025

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	0	1	1	4

TABLE OF CONTENTS

1. Summary

Renzo engaged Halborn to conduct a security assessment on their smart contracts beginning on April 29th, 2025 and ending on May 2nd, 2025. The security assessment was scoped to smart contracts in the GitHub repository provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. ASSESSMENT SUMMARY

The team at Halborn assigned a full-time security engineer to assess the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which should be addressed by the Renzo team. The main ones were the following:

- Apply the minimum fee only when the swap direction reduces the deviation, ensuring that any trade which increases the distance from the reference price is always subject to the dynamic fee.
- Cap the pip count at type(uint24).max or revert once the deviation exceeds a sensible upper bound before performing the cast.
- Add a zero-address check for the pool manager address in the constructor to align with the existing validations and prevent accidental misdeployment.

3. CAVEATS

This assessment is limited to the code introduced in the custom Uniswap V4 hook under review. External dependencies, including Uniswap V4 core contracts, utility libraries, and other protocol components, are treated as black boxes and assumed to behave as documented. Any issues stemming from incorrect assumptions about these external systems, or from upstream integration behavior, fall outside the scope of this review. The evaluation focuses strictly on the logic implemented within the hook contract itself.

4. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local or public testnet deployment (`Foundry` , `Remix IDE`).

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: foundational-hooks

(b) Assessed Commit ID: 8b51d43

(c) Items in scope:

- src/interfaces/IRateProvider.sol
- src/libraries/SqrtPriceLibrary.sol
- src/PegStabilityHook.sol
- src/RenzoStability.sol

Out-of-Scope: Third party dependendencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - MINIMUM FEE APPLIED TO SWAPS THAT ENLARGE THE OVER PEG DEVIATION	MEDIUM	PENDING
HAL-02 - EXTREME DEVIATION MAY COLLAPSE THE FEE TO A SMALL VALUE	LOW	PENDING
HAL-03 - POOL MANAGER ADDRESS IS NOT VALIDATED IN THE CONSTRUCTOR	INFORMATIONAL	PENDING
HAL-04 - UNUSED HELPER INFLATES BYTECODE SIZE	INFORMATIONAL	PENDING
HAL-05 - CONSTANT NAMES SUGGEST BASIS POINTS WHILE VALUES ARE PIPS	INFORMATIONAL	PENDING
HAL-06 - COMMENTS DO NOT MATCH THE IMPLEMENTED FEE LOGIC	INFORMATIONAL	PENDING

8. FINDINGS & TECH DETAILS

8.1 (HAL-01) MINIMUM FEE APPLIED TO SWAPS THAT ENLARGE THE OVER PEG DEVIATION

// MEDIUM

Description

In **RenzoStability.sol** the function `_calculateFee` returns `minFeeBps` whenever `zeroForOne` is **true** or `poolSqrtPriceX96` is lower than `referenceSqrtPriceX96`. When the pool is **over peg** (`poolSqrtPriceX96 < referenceSqrtPriceX96`, meaning ezETH is already more expensive than the reference) a `zeroForOne` swap sells ETH for ezETH and moves the price even farther from the peg, yet this trade still pays only the minimum fee because the second part of the condition is already satisfied:

```
119     function _calculateFee(
120         Currency,
121         Currency,
122         bool zeroForOne,
123         uint160 poolSqrtPriceX96,
124         uint160 referenceSqrtPriceX96
125     ) internal view override returns (uint24) {
126         // pool price is less than reference price (over pegged), or zeroForOne trades are moving towards the reference price
127         if (zeroForOne || poolSqrtPriceX96 < referenceSqrtPriceX96)
128             return minFeeBps; // minFee bip
129 
130         // computes the absolute percentage difference between the pool price and the reference price
131         // i.e. 0.005e18 = 0.50% difference between pool price and reference price
132         uint256 absPercentageDiffWad = SqrtPriceLibrary
133             .absPercentageDifferenceWad(
134                 uint160(poolSqrtPriceX96),
135                 referenceSqrtPriceX96
136             );
137 
138         // convert percentage WAD to pips, i.e. 0.05e18 = 5% = 50_000
139         // the fee itself is the percentage difference
140         uint24 fee = uint24(absPercentageDiffWad / 1e12);
141         if (fee < minFeeBps) {
142             // if % depeg is less than min fee %. charge minFee
143             fee = minFeeBps;
144         } else if (fee > maxFeeBps) {
145             // if % depeg is more than max fee %. charge maxFee
146             fee = maxFeeBps;
147         }
148         return fee;
149     }
```

The hook therefore under charges the very swaps that worsen an over peg, weakening the stabilisation logic and lowering fee income for liquidity providers.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (5.0)

Recommendation

It is recommended to apply the minimum fee only when the swap direction reduces the deviation, ensuring that any trade which increases the distance from the reference price is always subject to the dynamic fee. For reference, see the following example:

```
if ((poolSqrtPriceX96 > referenceSqrtPriceX96) == zeroForOne) return minFeeBps;
```

8.2 (HAL-02) EXTREME DEVIATION MAY COLLAPSE THE FEE TO A SMALL VALUE

// LOW

Description

In **RenzoStability.sol** the function `_calculateFee` casts the deviation (expressed in pips) from `uint256` to `uint24` without verifying its size. If the price gap grows beyond roughly **1,700 percent** the pip count overflows 24 bits, wraps to a much smaller number and can drop below `minFeeBps` :

```
119     function _calculateFee(
120         Currency,
121         Currency,
122         bool zeroForOne,
123         uint160 poolSqrtPriceX96,
124         uint160 referenceSqrtPriceX96
125     ) internal view override returns (uint24) {
126         // pool price is less than reference price (over pegged), or zeroForOne trades are moving towards the reference price
127         if (zeroForOne || poolSqrtPriceX96 < referenceSqrtPriceX96)
128             return minFeeBps; // minFee bip
129 
130         // computes the absolute percentage difference between the pool price and the reference price
131         // i.e. 0.005e18 = 0.50% difference between pool price and reference price
132         uint256 absPercentageDiffWad = SqrtPriceLibrary
133             .absPercentageDifferenceWad(
134                 uint160(poolSqrtPriceX96),
135                 referenceSqrtPriceX96
136             );
137 
138         // convert percentage WAD to pips, i.e. 0.05e18 = 5% = 50_000
139         // the fee itself is the percentage difference
140         uint24 fee = uint24(absPercentageDiffWad / 1e12);
141         if (fee < minFeeBps) {
142             // if % depeg is less than min fee %. charge minFee
143             fee = minFeeBps;
144         } else if (fee > maxFeeBps) {
145             // if % depeg is more than max fee %. charge maxFee
146             fee = maxFeeBps;
147         }
148         return fee;
149     }
```

This can only happen in an extreme market dislocation, yet in that scenario liquidity providers would receive far less compensation than intended because the fee would be calculated from the truncated value.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:N/Y:M (3.4)

Recommendation

It is recommended to cap the pip count at `type(uint24).max` or revert once the deviation exceeds a sensible upper bound before performing the cast, ensuring that even in extreme conditions the fee cannot shrink unexpectedly.

8.3 (HAL-03) POOL MANAGER ADDRESS IS NOT VALIDATED IN THE CONSTRUCTOR

// INFORMATIONAL

Description

The `constructor` of **RenzoStability.sol** verifies that the rate provider and ezETH token addresses are non-zero but omits the same check for `_poolManager`:

```
47      constructor(
48          IPoolManager _poolManager,
49          IRateProvider _rateProvider,
50          uint24 _minFee,
51          uint24 _maxFee,
52          address _ezETH
53      ) PegStabilityHook(_poolManager) {
54          // check for 0 value inputs
55          if (
56              address(_rateProvider) == address(0) ||
57              _minFee == 0 ||
58              _maxFee == 0 ||
59              _ezETH == address(0)
60          ) revert InvalidZeroInput();
61
62          // check for maxFee
63          if (_maxFee > MAX_FEE_BPS) revert InvalidMaxFee();
64
65          // check for minFee range
66          if (_minFee > _maxFee || _minFee < MIN_FEE_BPS) revert InvalidMinFee();
67
68          rateProvider = _rateProvider;
69          minFeeBps = _minFee;
70          maxFeeBps = _maxFee;
71          ezETH = _ezETH;
72      }
```

Deploying with a zero pool manager would leave the hook unusable and require a new deployment.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (1.7)

Recommendation

It is recommended to add a zero-address check for `_poolManager` in the constructor to align with the existing validations and prevent accidental misdeployment.

8.4 (HAL-04) UNUSED HELPER INFLATES BYTECODE SIZE

// INFORMATIONAL

Description

The **SqrtPriceLibrary.sol** library declares the `absDifferenceX96` function but no contract references it:

```
29      /// @notice Calculates the absolute difference between two sqrt prices
30      function absDifferenceX96(uint160 sqrtPriceAX96, uint160 sqrtPriceBX96) internal pure returns (uint160) {
31          return sqrtPriceAX96 < sqrtPriceBX96 ? (sqrtPriceBX96 - sqrtPriceAX96) : (sqrtPriceAX96 - sqrtPriceBX96);
32      }
```

The unused function increases bytecode size and audit surface without providing functionality, which is considered a poor development practice.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (1.7)

Recommendation

It is recommended to remove the unused function or integrate it where the absolute difference is required, thereby reducing deployment size and maintenance overhead.

8.5 (HAL-05) CONSTANT NAMES SUGGEST BASIS POINTS WHILE VALUES ARE PIPS

// INFORMATIONAL

Description

In **RenzoStability.sol** the constants `MAX_FEE_BPS`, `MIN_FEE_BPS`, `maxFeeBps` and `minFeeBps` are suffixed **BPS** even though the math treats them as pips (1 unit = 0.0001 percent):

```
25      // Fee bps range where 1_000_000 = 100 %
26      uint24 public constant MAX_FEE_BPS = 10_000; // 1% max fee allowed, 1% = 10_000
27      uint24 public constant MIN_FEE_BPS = 100; // 0.01% mix fee allowed
28
29      uint24 public immutable maxFeeBps;
30      uint24 public immutable minFeeBps;
```

Mislabeling units is a documentation antipattern that can mislead integrators and cause configuration errors.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to rename the constants or update their comments to clarify that they represent pips rather than basis points, ensuring consistent understanding across the codebase and external documentation.

8.6 (HAL-06) COMMENTS DO NOT MATCH THE IMPLEMENTED FEE LOGIC

// INFORMATIONAL

Description

In **RenzoStability.sol** the block comment above `_calculateFee` claims that the fee is one tenth of the percentage deviation and that a `zeroForOne` trade is treated as moving the price toward the reference only when that is actually **true**. The code instead applies the full deviation as the base fee and gives the minimum fee to every `zeroForOne` trade even when the pool is over peg, which makes the deviation larger:

```
110     /**
111      * @notice  Calculates the price for a swap
112      * @dev      linearly scale the swap fee as a tenth of the percentage difference between pool price and reference price
113      *           i.e. if pool price is off by 0.05% the fee is 0.05%, if the price is off by 0.50% the fee is 0.5%
114      * @param   zeroForOne  True if buying ezETH, false if selling ezETH
115      * @param   poolSqrtPriceX96  Current pool price
116      * @param   referenceSqrtPriceX96  Reference price obtained from the rate provider
117      * @return  uint24  Fee charged to the user - fee in pips, i.e. 3000 = 0.3%
118      */
119     function _calculateFee(
120         Currency,
121         Currency,
122         bool zeroForOne,
123         uint160 poolSqrtPriceX96,
124         uint160 referenceSqrtPriceX96
125     ) internal view override returns (uint24) {
126         // pool price is less than reference price (over pegged), or zeroForOne trades are moving towards the reference price
127         if (zeroForOne || poolSqrtPriceX96 < referenceSqrtPriceX96)
128             return minFeeBps; // minFee bip
129 
130         // computes the absolute percentage difference between the pool price and the reference price
131         // i.e. 0.005e18 = 0.50% difference between pool price and reference price
132         uint256 absPercentageDiffWad = SqrtPriceLibrary
133             .absPercentageDifferenceWad(
134                 uint160(poolSqrtPriceX96),
135                 referenceSqrtPriceX96
136             );
137 
138         // convert percentage WAD to pips, i.e. 0.05e18 = 5% = 50_000
139         // the fee itself is the percentage difference
140         uint24 fee = uint24(absPercentageDiffWad / 1e12);
141         if (fee < minFeeBps) {
142             // if % depeg is less than min fee %. charge minFee
143             fee = minFeeBps;
144         } else if (fee > maxFeeBps) {
145             // if % depeg is more than max fee %. charge maxFee
146             fee = maxFeeBps;
147         }
148         return fee;
149     }
```

These discrepancies can mislead auditors and integrators who rely on the comments to understand the economic behaviour.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to update the comments so they describe the current behavior accurately or to adjust the code to follow the documented intent, ensuring that documentation and implementation remain consistent and avoid confusion for future maintenance.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.