Three Sigma

Code Audit

Hoenn

Hoenn Unified Restaking Liquidity

Disclaimer

Code Audit

Hoenn Unified Restaking Liquidity

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Hoenn Unified Restaking Liquidity

Table of Contents

Summary

Code Audit

Hoenn Unified Restaking Liquidity

Summary

Three Sigma audited Hoenn in a 1.2 person week engagement. The audit was conducted from 03/02/2025 to 05/02/2025.

Protocol Description

Hoenn is a unified liquidity protocol designed to enhance the liquidity of diverse restaking strategies while isolating slashing risks. Users can deposit any Liquid Restaking Tokens (LRTs) as collateral in dedicated vaults to draw a unified synthetic token pegged 1:1 to the restaked asset. The synthetic token can be used for DeFi while continuing to earn (re)staking rewards and is redeemable at anytime.

Scope

Code Audit

Hoenn Unified Restaking Liquidity

Scope

Filepath	nSLOC
contracts/CvToken.sol	304
contracts/protocol/RsEthProtocol.sol	51
contracts/protocol/EthXProtocol.sol	34
contracts/protocol/MellowProtocol.sol	34
contracts/protocol/EEthProtocol.sol	30
contracts/protocol/WeEthProtocol.sol	25
contracts/Vault.sol	17
contracts/protocol/WstEthProtocol.sol	15
contracts/Common.sol	11
contracts/protocol/StEthProtocol.sol	10

Assumptions

External integrated contracts are considered secure.

Methodology

Code Audit

Hoenn Unified Restaking Liquidity

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Hoenn Unified Restaking Liquidity

Project Dashboard

Application Summary

Name	Hoenn
Commit	13f35ce
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	03/02/2025 to 05/02/2025
№ of Auditors	2
Review Time	1.2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	2	2	0
Medium	3	2	1
Low	5	4	1

None	4	4	0

Category Breakdown

Suggestion	2
Documentation	0
Bug	13
Optimization	1
Good Code Practices	0

Findings

Code Audit

Hoenn Unified Restaking Liquidity

Findings

3S-Hoenn-C01

An attacker can steal any user's cvETH

Id	3S-Hoenn-C01
Classification	Critical
Severity	Critical
Likelihood	High
Category	Bug
Status	Addressed in #48216fc.

Description

The **`cvToken::repay()`** function allows a user to specify the payer while repaying the minted **`cvETH`**. However, there is no authorization check to ensure that **`msg.sender`** is authorized to spend the payer's **`cvETH`**. As a result, an attacker can use any user's **`cvETH`** to repay their own minted **`cvETH`**, effectively stealing the user's funds.

Recommendation

Add allowance check to repay function .

```
    function repay(address payer, address target, uint256 amount) public
nonReentrant whenNotPaused returns (uint256) {
        if (amount <= 0) {
            revert Errors.InvalidAmount();
        }
+       if (payer != msg.sender)
+           _spendAllowance(payer, msg.sender, amount);
        return _repay(payer, target, amount, true);
    }
```

3S-Hoenn-H01

Full liquidation from minor insolvency incurs significant loss to users

Id	3S-Hoenn-H01
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #d349878.

Description

Full liquidation of positions with minor insolvency can lead to significant losses for users. In the current system, once a position becomes liquidatable, a liquidator can repay all cvTokens regardless of the extent to which the health factor has decreased. In return, the liquidator receives an equivalent value of LRT along with a certain percentage of the seized collateral as a liquidation bonus. Consequently, the larger the repaid amount, the greater the liquidation bonus.

However, in cases where the health factor has only slightly decreased, repaying a small portion of the assets should be sufficient to restore the position to a healthy state.

Example:

Let's assume the following:

- The current price of AgETH is **1.02 ETH**.

- The max loan-to-value (LTV) percentage of AgETH is set to **80%**.

- The liquidation threshold percentage is set to **2%**.

Scenario:

1. The user deposits **100 AgETH** into the protocol.

2. The user mints the maximum possible amount, which is **81.6 cvETH**.

3. Due to a price decrease (due to slashing), the price of AgETH drops to **1 AgETH = 0.97 ETH**.

4. As a result, the collateral value becomes: **`100 AgETH * 0.97 ETH = 97 ETH`**

5. The health factor becomes: **`(81.6 / 97) * 100 = 84%`**. This crosses the liquidation threshold of **82%**, making the position liquidatable.

6. The liquidator repays **81.6 cvETH**, and in return, they receive: **`(100 * 81.6 * 1.02) / 97 = 85.78 AgETH`**. This includes a bonus of **1.63 AgETH** (worth **4337.43 USD**).

However, to restore the position to a healthy state, the liquidator only needs to repay **18.8 cvETH**. This brings the health factor close to **80%**.

In this case, the liquidator receives: **`(100 * 18.8 * 1.02) / 97 = 19.76 AgETH`**, including a bonus of **0.38 AgETH** (worth **1011.18 USD**).

After this partial repayment, the position becomes healthy again, with the new health factor: **`(81.6 - 18.8) / (97 - 19.76) = 0.81`**

So the user will lose nearly **`3326.25 USD`** in the above case due to full liquidation.

Recommendation

This can be addressed in two steps:

1. Implement support for partial liquidations.

2. When liquidating a position, calculate the maximum liquidation amount. This should be determined based on the amount the liquidator needs to repay in order to restore the position to a healthy state. Only allow the liquidator to repay up to this calculated maximum liquidation amount.

3S-Hoenn-H02

Adding eETH as LRT will lead to accounting problems as eETH is a rebasing token

Id	3S-Hoenn-H02
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #d361fe4.

Description

Based on the protocol's deployment scripts, it appears that the team intends to add **`eETH`** as an LRT. However, since **`eETH`** is a rebasing token, adding it as an LRT will lead to accounting issues because its balance continuously changes.

For example:

1. A user deposits **`1.02 eETH`** today.

2. The protocol records the deposited amount as **`1.02 eETH`**.

3. After one year, the user's deposited **`eETH`** amount increases to **`1.10 eETH`** due to rebasing.

4. However, since the protocol still records only **`1.02 eETH`** in its accounting, the user can only withdraw the initially deposited **`1.02 eETH`**, losing their yield on **`eETH`**.

Recommendation

Avoid using rebasing LRT tokens as collateral. Instead, only allow the wrapped versions of rebasing LRT tokens.

3S-Hoenn-M01

Every LRT should have seperate liquidateThresholdPect

Id	3S-Hoenn-M01
Classification	Medium
Severity	Medium
Likelihood	High
Category	Bug
Status	Acknowledged with note “This update will require lots of changes to the code and the liquidation logic. We are aware of the issue, and agree that this kind of more sophisticated parameters are required. But not sure if we can decide the values and the logic at this stage as slashing in restaking is very early. We will keep trying to find a better solution as the restaking protocols are growing, but it seems not suitable to make the liquidation logic more complex from the early stage. The liquiateThresholdPect parameter is subject to change, as 2% is just a random number for testing.”

Description

Currently, the protocol uses the same **`liquidateThresholdPect`** for all supported LRTs. Users are allowed to mint cvETH up to the **`maxLTV`** percentage, and **`liquidateThresholdPect`** serves as the buffer percentage beyond which a user's position will be liquidated.

If a specific asset is considered more risky (volatile), the protocol sets a lower **`maxLTV`** for it. However, the **`liquidateThresholdPect`** should also vary based on the LRT's volatility.

For example:

If eETH is considered more volatile and only 50% **`maxLTV`** is allowed, setting **`liquidateThresholdPect`** to just 2% could make users' positions too easily liquidatable.

Recommendation

Allow different liquidation threshold percentages for each LRT based on its volatility.

3S-Hoenn-M02

User's position might be bricked when an LRT is removed

Id	3S-Hoenn-M02
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	Addressed in #ff62ee2.

Description

When an LRT is removed using the **`CvToken::removeLrtProtocol()`** function, but some users still have deposits of that token, their positions will become inaccessible due to the following check in the **`getLRTUnderlyingValue()`** function. This check is triggered while calculating the health of a user's position through the **`getUserLoanInfo()`** function:

```
if (protocol == address(0)) {
    revert Errors.ProtocolNotFound(lrt);
}
```

Recommendation

Modify **`getUserLoanInfo()`** to skip removed assets while calculating a user's loan info, as shown below:

```
if (userDeposit.amount == 0 || lrtToConfig[userDeposit.lrt].protocol ==
address(0)) {
       continue;
}
```

3S-Hoenn-M03

Add max length bound to user's LRT deposits array

Id	3S-Hoenn-M03
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	Addressed in #eef7fb7.

Description

`getUserLoanInfo` iterates over all user deposits to calculate the value of deposited LRTs. The maximum length of LRTs would be the number of LRTs configured in the **`CvToken`** contract.

Additionally, the value of the **`RsEth`** LRT is calculated using the **`RsEthProtocol`** protocol. The **`getUnderlyingValue`** function is used to determine the value of a given amount of **`RsEth`** tokens. However, the **`RsEth`** deposit pool supports multiple LST tokens. Currently, to determine the **`RsEth`** value, **`getUnderlyingValue`** iterates over all supported LSTs to retrieve their asset balances. Since there is no maximum limit on the number of supported LSTs in the **`RsEth`** protocol, this number may increase in the future.

Due to these two unbounded loops, there could be scenarios in the future where **`getUserLoanInfo`** reverts. For example, if a user has deposited multiple LRTs, including **`RsEth`**, and the number of supported LSTs for **`RsEth`** and deposited LRTs is sufficiently large, the transaction could exceed the block gas limit, causing it to revert.

Although the likelihood of this happening is low, if it does occur, the user would be unable to withdraw all their deposits. Core functions such as liquidation and repayment would also fail, while only deposits would continue to work, increasing potential losses for the user.

Recommendation

Limit the maximum number of LRTs a user can deposit.

3S-Hoenn-L01

Incorrect penalty calculation during liquidation

Id	3S-Hoenn-L01
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #698eac7.

Description

During liquidation, if the LTV exceeds a significant amount (for example: 96%), the fee calculation becomes flawed. As a result, the vault loses its fee, and the liquidator gains additional profit.

Example Scenario

Configuration:

- **`liquidateBonusPect`** = 2%

- **`liquidatePenaltyPect`** = 5%

- Vault fee percentage = **`liquidatePenaltyPect - liquidateBonusPect`** = 3%

Given Values:

- Total deposit value = 100

- Repay value = 96.078

- Liquidation bonus = 2% of 96.078 = 1.921

Calculation:

- **`payerPect`** = **`96.078 + 1.921`** = **97.99%**

- **`penaltyPect`** = 3% of 96.078 = **2.88%**

As **`payerPect + penaltyPect`** crosses 100% below check passes:

```
if (payerPect + penaltyPect > 1e18) {
    // Open possibilities for repaying bad debt, as LRT may have more value
behind
    payerPect = 1e18;
    penaltyPect = 0;
}
```

- **`payerPect`** becomes 100%

- **`penaltyPect`** becomes 0

As a result, the liquidator, who should receive only a 1.921 (2%) bonus, instead receives 3.921 (~4%), leading to unintended extra profit at the vault's expense.

Recommendation

This issue can be fixed by modifing fee calculation logic to

1. If **`repay value + liquidation bonus`** is greater than 100%, the liquidator should get **`100% - repay value`**. Vault should get 0

2. If **`repay value + liquidation bonus`** is less than 100%, liquidator should get **`liquidation bonus`** and vault should get **`100% - (repay value + liquidation bonus)`**

3S-Hoenn-L02

Add a delay mechanism to LTV updates to avoid unfair liquidations

Id	3S-Hoenn-L02
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #309a2dd.

Description

Currently, LTV parameter updates take effect immediately, which can lead to unfair liquidations, especially for users whose positions are already close to the liquidation threshold.

Recommendation

Implement a delay mechanism for LTV updates to ensure users have sufficient time to adjust their positions before the changes take effect, preventing unexpected liquidations.

3S-Hoenn-L03

Incorrect maxLtvPect validation in addLrtProtocol

Id	3S-Hoenn-L03
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Addressed in #9482a83.

Description

While adding a new LRT using **`addLrtProtocol`**, certain sanity checks are performed. These checks ensure that **`maxLtvPect`** does not exceed 100% and that the liquidation threshold does not exceed 100%.

```
if (maxLtvPect >= 1e18 || maxLtvPect + config.liquidatePenaltyPect > 1e18) {
    revert Errors.InvalidMaxLtvPect(lrt, maxLtvPect);
}
```

However, instead of using **`liquidateThresholdPect`**, the check incorrectly uses **`liquidatePenaltyPect`**.

Recommendation

Use **`liquidateThresholdPect`** instead of **`liquidatePenaltyPect`**

```
-if (maxLtvPect >= 1e18 || maxLtvPect + config.liquidatePenaltyPect > 1e18)
{
+if (maxLtvPect >= 1e18 || maxLtvPect + config.liquidateThresholdPect >
1e18) {
    revert Errors.InvalidMaxLtvPect(lrt, maxLtvPect);
}
```

3S-Hoenn-L04

Minting fee charged to users will be slightly higher due to over collateralization

Id	3S-Hoenn-L04
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Acknowledged with note “We expect users to buy more cvETH from the market to withdraw fully, thus the remaining collateral in the vault is not locked forever.”

Description

Since the fee is charged to users in **`cvETH`**, they will end up paying slightly more value in **`ETH`** to the protocol than intended.

For example:

1. A user supplies **`200 ETH`** worth of collateral and mints **`100 ETH`** (assuming a max LTV of 50%).

2. When burning **`cvETH`**, a **`1 ETH`** minting fee is charged(0.5%).

3. However, as the user has to maintain **`200%`** collateral (over-collateralization), they are unable to withdraw **`2 ETH`** worth of their LRTs while effectively paying a **`1%`** fee.

Recommendation

Consider modifying the fee mechanism if this behaviour is not intended.

3S-Hoenn-L05

Bad debt will never get liquidated

Id	3S-Hoenn-L05
Classification	Low
Severity	Medium
Likelihood	Low
Category	Bug
Status	Addressed in #d349878.

Description

If bad debt occurs, it means the value of the collateral has fallen below the repayment amount. In such a case, if a liquidator attempts to liquidate the position, they must repay the full amount, as partial liquidations are not allowed currently. This results in bad debt remaining uncleared since liquidators have no incentive to perform such liquidations.

However, if partial liquidations were permitted, a liquidator could choose to repay a significant amount of the bad debt while still receiving an incentive. This would leave only a small amount of bad debt, whereas in the previous scenario, the bad debt would be significantly higher.

Recommendation

Implement support bad partial liquidations

3S-Hoenn-N01

Implement receiver parameter for mint and deposit functions

Id	3S-Hoenn-N01
Classification	None
Category	Bug
Status	Addressed in #5a35c44.

Description

Currently, the **`deposit`** and **`mint`** functions do not have a **`receiver`** parameter, unlike the **`withdraw`** function. This limitation prevents users or smart contracts from depositing collateral or minting **`cvETH`** to a different address.

Recommendation

Add a **`receiver`** parameter to the **`deposit`** and **`mint`** functions, allowing users to specify a different address for receiving the deposited collateral or minted **`cvETH`**. This will improve flexibility for both users and smart contracts.

3S-Hoenn-N02

No max cap for minting fee

Id	3S-Hoenn-N02
Classification	None
Category	Suggestion
Status	Addressed in #d140758.

Description

Currently, there is no max cap for the minting fee. Implementing a max fee cap will ensure that users are guaranteed the fee will not exceed the predefined cap value.

Recommendation

Implement a max fee cap.

3S-Hoenn-N03

Protocol contracts can be simplified by using provided exchange rate functions

Id	3S-Hoenn-N03
Classification	None
Category	Suggestion
Status	Addressed in #de37eb5.

Description

Currently, LRT pricing contracts calculate the price of LRTs from scratch using **`totalAssets`** and **`totalSupply`** variables instead of utilizing the provided exchange rate function, such as **`convertToAssets`**.

Recommendation

For example, **`MellowProtocol::getUnderlyingValue()`** can be updated as follows:

```
contract MellowProtocol is IProtocol, Ownable {
    function getUnderlyingValue(address token, uint256 amount) public view
returns (uint256) {
        address asset = IERC4626(token).asset();
        uint256 assets = IERC4626(token).convertToAssets(amount);
        return ICvToken(cvToken).getLSTUnderlyingValue(asset, assets);
    }
}
```

`EthXProtocol::getUnderlyingValue()` can be simplified:

```
contract EthXProtocol is IProtocol {
    function getUnderlyingValue(address token, uint256 amount) public view
returns (uint256) {
```

```
        address config = IEthX(token).staderConfig();
        address manager = IStaderConfig(config).getStakePoolManager();
        return IPoolManager(manager).convertToAssets(amount);
    }
```

3S-Hoenn-N04

No need to calculate exchange rate of eETH as its 1:1 pegged to ETH

Id	3S-Hoenn-N04
Classification	None
Category	Bug, Optimization
Status	Addressed in #d5b32d5.

Description

The **`EEthProtocol::getUnderlyingValue()`** function currently calculates the exchange rate of eETH to ETH using **`totalPooledEther`** and **`totalSupply`**, which always returns 1 because eETH is pegged 1:1 to ETH.

Recommendation

The **`EEthProtocol::getUnderlyingValue()`** function can simply return the **`amount`** without unnecessary calculations.

```
contract EEthProtocol is IProtocol {
    function getUnderlyingValue(address token, uint256 amount) public view
returns (uint256) {
        require(token == Constants.E_ETH_LRT, "Invalid token");
        return amount;
    }
}
```

Similarly WeETH contract can be updated

```
contract WeEthProtocol is IProtocol, Ownable {
function getUnderlyingValue(address token, uint256 amount) public view
returns (uint256) {
        uint256 eEthBalance = IERC20(Constants.E_ETH_LRT).balanceOf(token);
        uint256 totalSupply = IERC20(token).totalSupply();
        if (totalSupply == 0) {
            return 0;
```

```
        }
        uint256 amountInEEth = amount * eEthBalance / totalSupply;
        return amountInEEth;
    }
}
```