Security Review Report
NM-0250 Dinero Pirex

NETHERMIND
SECURITY

(Jul 29, 2024)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 System Overview 4
- 4.1 Deposits . . . 4
- 4.2 Cross-chain synchronization . . . 5
- 4.3 Withdrawals . . . 5
- 4.4 Rebases . . . 5

5 Risk Rating Methodology 6

6 Issues 7
- 6.1 Issues found during initial audit engagement . . . 7
  - 6.1.1 [Critical] A user can increase shares for free by performing a self-transfer . . . 7
  - 6.1.2 [Critical] Double-subtraction of assets amount during withdraw . . . 8
  - 6.1.3 [Critical] Lack of `totalStaked` increase in `_lzReceive(...)` . . . 9
  - 6.1.4 [Critical] The `unwrap(...)` function burns more tokens than it should . . . 10
  - 6.1.5 [Critical] The withdrawal amount decreases the `pendingDeposit` twice . . . 10
  - 6.1.6 [High] Incorrect update of `RateLimiter` causes lock of funds on Mode Network . . . 11
  - 6.1.7 [High] Rebase prevention . . . 12
  - 6.1.8 [High] The slow-sync process finalization can be DOSed . . . 13
  - 6.1.9 [Medium] Contracts cannot be paused . . . 14
  - 6.1.10 [Medium] Deposit and sync can't be paused . . . 14
  - 6.1.11 [Medium] Oracle may return a stale rate, which would decrease the `totalStaked` . . . 15
  - 6.1.12 [Medium] The ETH bridge will fail if the Pirex deposit fee is non-zero . . . 16
  - 6.1.13 [Medium] `RateLimiter` may be set to `address(0)` . . . 17
  - 6.1.14 [Low] Add transaction `ID` to ensure correct cross-chain message finality . . . 17
  - 6.1.15 [Low] The `L1` Ether deposits will revert for non-zero PirexETH deposit fee . . . 18
  - 6.1.16 [Low] `LayerZero` does not enforce the correct order of message consumption . . . 19
  - 6.1.17 [Info] Incorrect call to `RateLimiter` during the deposit on the L2 sync pool . . . 19
- 6.2 Issues found during fix review phase . . . 20
  - 6.2.1 [High] Incorrect move of funds from pending to staked state . . . 20
  - 6.2.2 [High] The `_calculateWithdrawalAmount(...)` should not be done on `pendingDeposit` . . . 21
  - 6.2.3 [Medium] Increasing deposit fee during the bridging of funds causes mint of unbacked `pxETH` . . . 21
  - 6.2.4 [Medium] L2 deposits are allowed when the protocol is paused . . . 21
  - 6.2.5 [Medium] Unnecessary centralization risk in `sweep(...)` function . . . 22
  - 6.2.6 [Low] Incorrect calculation of rebase fee . . . 22
  - 6.2.7 [Low] Pirex deposit fee updates require an upgrade of `L2ExchangeRateProvider` . . . 23
  - 6.2.8 [Low] The `_updateSyncQueue(...)` function might cause the LayerZero transaction to run out of gas . . . 23
  - 6.2.9 [Low] The rebase fee might be applied even if there are no funds staked . . . 24
  - 6.2.10 [Info] Implementation contracts can be initialized . . . 24
  - 6.2.11 [Info] Incorrect event emission . . . 25
  - 6.2.12 [Info] Incorrect update of `totalUnbackedTokens` . . . 25
  - 6.2.13 [Info] No check of zero amount in `_update(...)` . . . 25
  - 6.2.14 [Info] Withdrawals might revert out of gas if `minSyncAmount` is low . . . 26
  - 6.2.15 [Best Practices] Missing `address(0)` checks on setter functions . . . 26

7 Documentation Evaluation 27

8 Test Suite Evaluation 28

9 About Nethermind 34

1 Executive Summary

This document presents the security review performed by Nethermind Security for the Dinero. This security engagement focused on the smart contracts extending the `Pirex` protocol towards the `Mode` Layer 2 (L2) network. `Pirex` is a delegated staking protocol deployed on the Ethereum mainnet. The newly added functionalities enable the users to earn the `apxETH` yield on L2 chains. Users can deposit native tokens on L1 or L2 and receive the liquid staking token shares on L2. The native tokens from Layer 2 are bridged to L1 and deposited into `Pirex`.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** thirty-two points of attention, where five are classified as `Critical`, five are classified as `High`, eight are classified as `Medium`, seven are classified as `Low`, and seven are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

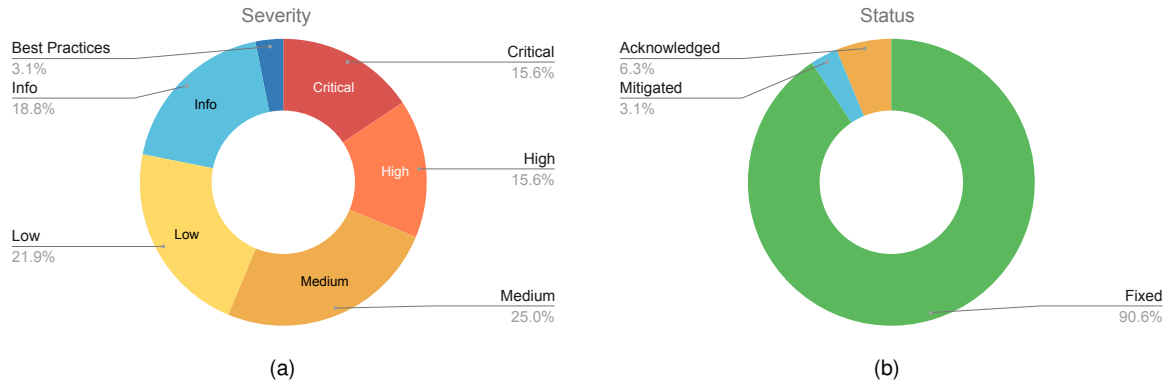

Fig. 1: Distribution of issues: Critical (5), **High** (5), **Medium** (8), **Low** (7), **Undetermined** (0), **Informational** (6), **Best Practices** (1). **Distribution of status: Fixed** (29), **Acknowledged** (2), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun 28, 2024
Response from Client	Regular responses during audit engagement
Final Report	Jul 29, 2024
Repository	dinero-pirex-eth
Commit (Audit)	2a005ea9cd2c499f081f09e851947a60ef16a010
Commit (Final)	d463e089a042b4291c6a8e06d238ac45722db9a1
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	DineroERC20RebaseUpgradeable.sol	159	196	123.3%	52	407
2	RateLimiter.sol	42	29	69.0%	6	77
3	L2ExchangeRateProvider.sol	42	7	16.7%	12	61
4	WrappedLiquidStakedToken.sol	85	50	58.8%	27	162
5	LiquidStakingTokenLockbox.sol	431	220	51.0%	127	778
6	L2ModeSyncPoolETH.sol	108	49	45.4%	19	176
7	L1SyncPoolETH.sol	91	56	61.5%	30	177
8	L1ModeReceiverETH.sol	20	17	85.0%	6	43
9	LiquidStakingToken.sol	318	168	52.8%	77	563
10	libraries/Errors.sol	11	25	227.3%	8	44
11	libraries/Constants.sol	11	38	345.5%	8	57
	Total	**1318**	**855**	**64.9%**	**372**	**2545**

3 Summary of Issues

	Finding	Severity	Update
1	A user can increase his shares for free by performing a self-transfer	Critical	Fixed
2	Double-subtraction of assets amount during withdraw	Critical	Fixed
3	Lack of `totalStaked` increase in `_lzReceive(...)`	Critical	Fixed
4	The `unwrap(...)` function burns more tokens than it should	Critical	Fixed
5	The withdrawal amount decreases the `pendingDeposit` twice	Critical	Fixed
6	Incorrect move of funds from pending to staked state	High	Fixed
7	Incorrect update of `RateLimiter` causes lock of funds on Mode Network	High	Fixed
8	Rebase prevention	High	Fixed
9	The `_calculateWithdrawalAmount(...)` should not be done on `pendingDeposit`	High	Fixed
10	The slow-sync process finalization can be DOSed	High	Fixed
11	Contracts cannot be paused	Medium	Fixed
12	Deposit and sync can't be paused	Medium	Fixed
13	Increasing deposit fee during the bridging of funds causes mint of unbacked `pxETH`	Medium	Mitigated
14	L2 deposits are allowed when the protocol is paused	Medium	Fixed
15	Oracle may return a stale rate, which would decrease the `totalStaked`	Medium	Fixed
16	The ETH bridge will fail if the Pirex deposit fee is non-zero	Medium	Fixed
17	Unnecessary centralization risk in `sweep(...)` function	Medium	Fixed
18	`RateLimiter` may be set to `address(0)`	Medium	Fixed
19	Add transaction `ID` to ensure correct cross-chain message finality	Low	Fixed
20	Incorrect calculation of rebase fee	Low	Fixed
21	Pirex deposit fee updates require an upgrade of `L2ExchangeRateProvider`	Low	Fixed
22	The `L1` Ether deposits will revert for non-zero PirexETH deposit fee	Low	Fixed
23	The `_updateSyncQueue(...)` function might cause the LayerZero transaction to run out of gas	Low	Fixed
24	The rebase fee might be applied even if there are no funds staked	Low	Acknowledged
25	`LayerZero` does not enforce the correct order of message consumption	Low	Fixed
26	Implementation contracts can be initialized	Info	Fixed
27	Incorrect call to `RateLimiter` during the deposit on the L2 sync pool	Info	Fixed
28	Incorrect event emission	Info	Fixed
29	Incorrect update of `totalUnbackedTokens`	Info	Fixed
30	No check of zero amount in `_update(...)`	Info	Fixed
31	Withdrawals might revert out of gas if `minSyncAmount` is low	Info	Acknowledged
32	Missing `address(0)` checks on setter functions	Best Practices	Fixed

4 System Overview

The `LiquidStakingToken` contract is the extension of the `Pirex` protocol for the `Mode` Layer 2 (L2). Instead of holding the `AutoPxEth` vault shares on Layer 1 (L1), users can hold the Liquid Staking Token (`LST`) on L2 and benefit from the same `apxETH` yield. The `LST` can be obtained by performing a deposit on either L1 or L2. On L1, the accepted tokens include `Ether`, `pxETH`, and `apxETH`, while on L2, only the Layer 2 `Ether` is accepted. Unlike deposits, the withdrawals can be initiated exclusively from the L2 side. In exchange for the `LST`, users receive an appropriate amount of `pxETH` tokens on L1 based on the conversion ratio from the `AutoPxEth` vault.

The following sections delve into the system's components and their interactions. The diagram below showcases a high-level view of the system's architecture.

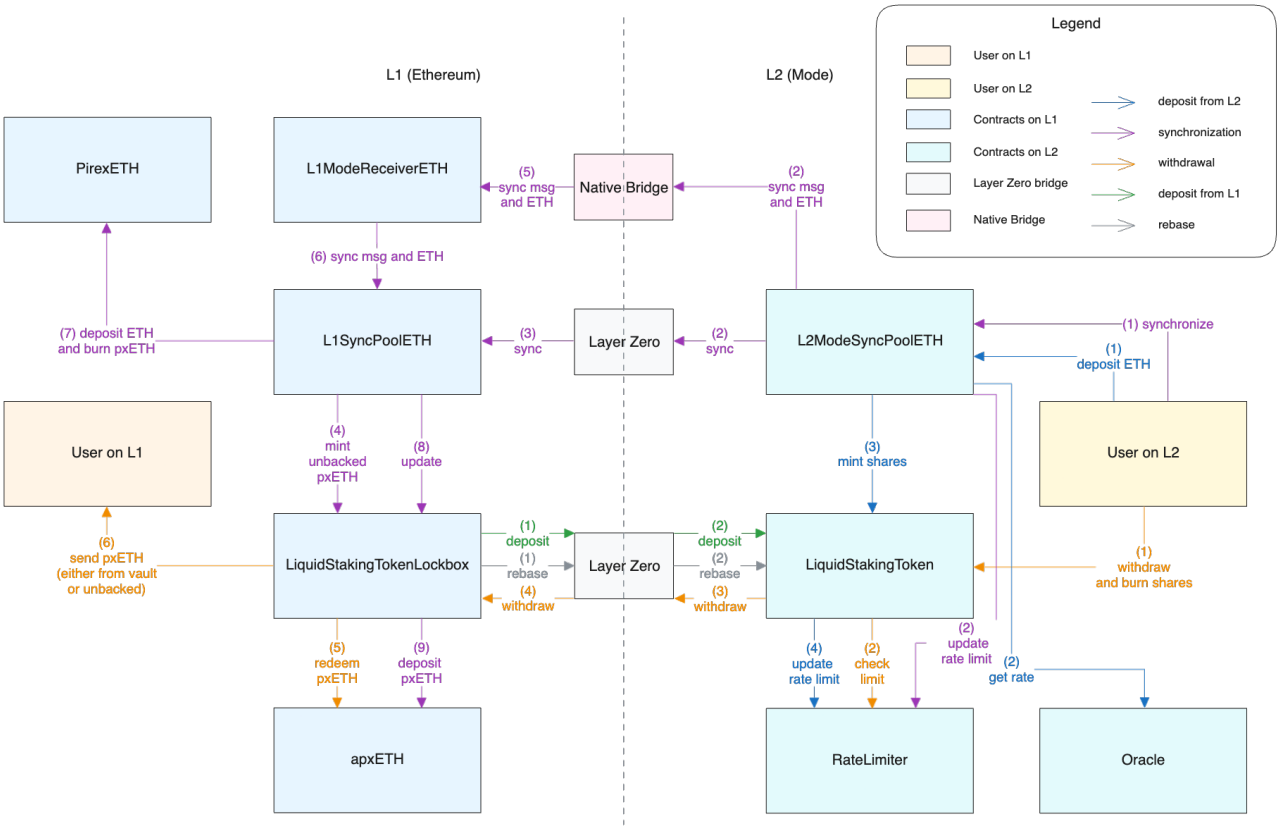

Fig. 2: Pirex cross-chain communication overview.

4.1 Deposits

Layer 1 deposits are facilitated through the `LiquidStakingTokenLockbox` contract. The Lockbox is responsible for handling the `Ether`, `pxETH`, and `apxETH` token deposits into the `Pirex` protocol. Communication between L1 and L2 is done with the LayerZero messaging system. During the deposit, the Lockbox sends the LayerZero message to the L2 chain with the information about the L1 state changes. The `LiquidStakingToken` contract that receives this message on L2 can use the provided information to mint an appropriate amount of `LST` shares as well as perform a rebase using the new assets per share ratio from the `AutoPxEth` vault.

Layer 2 deposits are facilitated through the `L2ModeSyncPool` contract. Since the L2 `Ether` cannot directly be used for staking in the `Pirex` protocol, it must be first bridged to the L1 `Ether` via the native Layer 2 bridge. In the case of the Mode network, on which the project will be deployed first, it is the Standard Bridge from the Optimism Stack. The L2 deposits are batched together and sent to L1 during the cross-chain syncing process. The synchronization mechanism itself is explained in greater detail in the following section. Once `Ether` is released from the L1 native bridge, it is deposited into the `Pirex` protocol to start the `Ether` staking process. The validator staking rewards generate the yield for the `LST` and `apxETH` token holders.

4.2 Cross-chain synchronization

The L1 chain is unaware of the user deposits on L2 until the two chains are synchronized. The syncing process can be done by calling the `sync(...)` function on the `L2ModeSyncPoolETH` contract. To keep the chain states up to date, the off-chain keeper will trigger the synchronization regularly. This action is permissionless, meaning that anyone can call the `sync(...)` function once a certain threshold of deposits is reached.

The synchronization mechanism is split into two parts: the slow sync and the fast sync. **The slow sync** process sends the native Layer 2 `Ether` to Layer 1 over the native bridge. Due to the nature of optimistic rollups and the design of the fault-proof system, this process takes at least 7 days to finalize (for OP Stack-based rollups, e.g., Mode Network). The message won't be relayed during that time, and the `Ether` won't be released on L1. To mitigate this limitation, **the fast sync** message is sent via LayerZero omnichain messaging protocol to inform the L1 about the deposit on L2. The `L1SyncPoolETH` contract receives this message and mints `pxETH` tokens in anticipation of the `Ether` that is yet to be released from the bridge. The newly minted `pxETH` tokens are sent to the `LiquidStakingTokenLockbox` contract, where they await for the slow sync process to finish. If, during the waiting period, users request withdrawals on L2, whenever possible, they will be provided with the `pxETH` tokens from the Lockbox first instead of withdrawing funds from the `AutoPxEth` vault. The fast sync mechanism enables immediate liquidity for the L2 users without affecting the existing `AutoPxEth` deposits before the actual `Ether` arrives from the bridge.

4.3 Withdrawals

Unlike deposits, withdrawals can only be initiated on the L2 side. Users can call the `withdraw(...)` function on the `LiquidStakingToken` contract and specify the amount of assets that they want to withdraw. Their LST shares will be burned on L2, and a withdrawal message will be sent to L1 via LayerZero. Once the `LiquidStakingTokenLockbox` contract receives the message, it will transfer the `pxETH` tokens to the user on L1. As mentioned in the previous section, the Lockbox will first attempt to use all the `pxETH` that it currently holds, and only after that will it start withdrawing additional `pxETH` tokens from the vault.

4.4 Rebases

Whenever new rewards are distributed into the `AutoPxEth` vault, the price of an individual `apxETH` share increases. Since `LST` shares are L2 representations of the `apxETH` shares, this share price increase must also be reflected on the Layer 2 chain. The rebase mechanism informs the L2 about the newest assets per share ratio from the `AutoPxEth` vault. The current L1 share price is used to update the internal accounting on L2. Similarly to the synchronization mechanism, calling `rebase(...)` is permissionless but will be regularly called by the Keeper.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 Issues found during initial audit engagement

6.1.1 [Critical] A user can increase shares for free by performing a self-transfer

File(s): `src/layer2/DineroERC20RebaseUpgradeable.sol`

Description: The `LiquidStakingToken` contract is a rebasing token that rewards users with its shares in exchange for depositing funds into the protocol. The user's token balances are based on the total `pxETH` assets controlled by the protocol, the total shares minted, and the user's shares. The token balances change when the assets are deposited or withdrawn and when shares are minted or burned.

One of the functions at the user's disposal is the `transferShares(...)` function, which allows the user to transfer shares to any other account. Provided that the sender has enough shares, the token transfer of the `LiquidStakingToken` will update the internal accounting by decreasing the sender's shares and increasing the recipient's.

```
function _transferShares(
    address _sender,
    address _recipient,
    uint256 _shares
) internal {
    // ...
    DineroERC20RebaseStorage storage $ = _getDineroERC20RebaseStorage();

    // @audit If _sender and _recipient is the same address,
    // these two variables hold the same amount.
    uint256 currentSenderShares = $.shares[_sender];
    uint256 currentRecipientShares = $.shares[_recipient];

    if (_shares > currentSenderShares) revert Errors.InvalidAmount();

    // @audit The first variable is decremented and
    // account's share balance is decreased.
    $.shares[_sender] = currentSenderShares - _shares;
    // @audit However the second one will overwrite the first one
    // and increase the account's share balance.
    $.shares[_recipient] = currentRecipientShares + _shares;
}
```

The problem present in this function is that it does not check if the `_sender` and `_recipient` are the same address. If so, the `currentSenderShares` and `currentRecipientShares` variables will hold the same value. The former will be decreased by the `_shares` amount being transferred and used to update the sender's balance. The latter, which still holds the pre-transfer amount of shares, will be increased and used to update the sender's balance again, effectively overwriting the previous assignment. As a result, the caller was able to increase own share balance by the specified amount.

Since the transfer operation does not change the total supply of shares, the malicious user could repeatedly perform this operation to increase his token balance without diluting the individual share price in terms of assets. Using the shares, the user would withdraw the assets up to the withdrawal limit, leaving the other users' shares worthless.

Recommendation(s): Consider updating the balances using the latest balances read from storage instead of cached variables. Also, consider implementing a check to ensure that self-transfers are not possible.

Status: Fixed

Update from the client: Fixed: fb71390

6.1.2 [Critical] Double-subtraction of assets amount during withdraw

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The `LiquidStakingToken` is a rebasing token deployed on `L2`, e.g., `Mode Network`, that represents the `apxETH` tokens on `Ethereum Mainnet`. The `LiquidStakingToken` tracks the amount of total assets (underlying) as a sum of two global variables: `totalStaked` and `pendingDeposit`, where the first represents `ETH` being actively staked, and second the `ETH` that was deposited on `L2`, but is waiting in the bridge. The `totalStaked` increases in the `_lzReceive(...)` when the funds are deposited on `L1`. The `pendingDeposit` is increased in `mint(...)` when the funds are deposited on `L2`. However, during withdrawal, if the `pendingDeposit` is greater than the `_amount`, the withdrawn amount is subtracted from both `totalStaked` and `pendingDeposit`:

```
function withdraw(
    address _receiver,
    address _refundAddress,
    uint256 _amount,
    bytes calldata _options
) external payable nonReentrant whenNotPaused {
    // ...
    uint256 pendingDeposit = $.pendingDeposit;

    if (pendingDeposit > 0) {
        if (pendingDeposit > _amount) {
            // @audit-note: First subtraction of _amount from pendingDeposit
            $.pendingDeposit -= _amount;
            ////////////////////////////////////////////////
            // @audit-issue: missing update of _amount = 0
            ////////////////////////////////////////////////
        } else {
            $.pendingDeposit = 0;
            _amount -= pendingDeposit;
        }
    }
    // @audit-note: Second subtraction of _amount from totalStaked
    $.totalStaked -= _amount;

    emit Withdrawal(msgReceipt.guid, msg.sender, _receiver, _amount);
}
```

As a result, the withdrawn amount is subtracted twice, and `totalAssets` is smaller than it should be. Consider the following scenario:

– user `A` deposits `2ETH` on L2 through `L2ModeSyncPoolETH.deposit(...)`;

- current state in `LST`: `pendingDeposit = 2ETH`, `totalStaked = 0`

– user `B` deposits `1ETH` on L1 through `LiquidStakingTokenLockbox.depositEth(...)` (assume `LZ` message reached `L2`);

- current state in `LST`: `pendingDeposit = 2ETH`, `totalStaked = 1ETH`

– user `B` withdraws `1ETH` on L2 through `LiquidStakingToken.withdraw(...)`;

- current state in `LST`: `pendingDeposit = 1ETH`, `totalStaked = 0` - but state should be: `pendingDeposit = 1ETH`, `totalStaked = 1ETH`

The `totalAssets` is decreased by `2ETH` instead of `1ETH`. As a result, the `totalAssets` is lower than it should be, and subsequent withdrawals would calculate more shares than they should have. Consequently, other users won't be able to withdraw the funds because the `RateLimiter` was not updated with enough shares but also because the required shares to burn are higher than the total shares minted.

Recommendation(s): To avoid double asset subtraction during withdrawal, consider updating the `_amount` to `0` in the branch `if (pendingDeposit > _amount)`.

Status: Fixed

Update from the client: Fixed: bb51717

6.1.3 [Critical] Lack of `totalStaked` increase in `_lzReceive(...)`

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The `LiquidStakingToken` splits the amount of total assets between `totalStaked` and `pendingDeposit`, so that only `totalStaked` amount is rebased. The `pendingDeposit` is the amount that was deposited on `L2` and was not yet bridged to `L1` for staking, and the `totalStaked` is the amount of `ETH` that is already staked on `L1`. The `pendingWithdraw` variable is used to communicate between `L1` and `L2` that the funds deposited on `L2` were already bridged to `L1` and possibly already used for staking. The `pendingWithdraw` is increased by the amount of bridged `ETH` when the bridge process is finalized, and this bridged `ETH` is deposited further to `Pirex`. Then `pendingWithdraw` is sent to the `L2` side. The `LiquidStakingToken._lzReceive(...)` uses `pendingWithdraw` to decrease `pendingDeposit`:

```
function _lzReceive() internal override {
    // ...
    _pendingWithdraw > $.pendingDeposit
        ? $.pendingDeposit = 0
        : $.pendingDeposit -= _pendingWithdraw;

    $.lastAssetsPerShare = _assetsPerShare;

    // @audit We decreased "pending" but did not increase "staked".
}
```

This update means that we can decrease the `pendingDeposit` by the `pendingWithdraw` because this amount is now not pending but is being staked. However, the increase of the `totalStaked` variable is missing. That causes the decrease of `totalAssets`, where actual `totalAssets` should stay the same because the desired action is to move funds from the "pending" to the "staking" state. As a consequence of the artificial lowering of the `totalAssets`, the user won't be able to withdraw because the number of shares calculated in `convertToShares(...)` will be higher than the total shares minted and higher than the limit set in `RateLimiter`.

Recommendation(s): Consider increasing `totalStaked` when decreasing `pendingDeposit`.

Status: Fixed

Update from the client: Fixed: ad5c5ee

Update from the Nethermind Security: Introduced changes solve the described problem. However, there are two problems with the current solution:

1. The `_updateTotalStaked(...)` at the end of `_lzReceive(...)` causes the double rebasing of the `totalAssets`, since the rebase was previously done before;
2. The whole amount of `_pendingWithdraw` is added to the `totalStaked` even if only part of `_pendingWithdraw` was subtracted from the `pendingDeposit`. In some cases, it may cause artificial increase of `totalAssets`;

Update from the client: Fixed double scaling: beebc32 Fixed increasing more than pending deposit: 47accdb

6.1.4 [Critical] The `unwrap(...)` function burns more tokens than it should

File(s): `src/layer2/WrappedLiquidStakedToken.sol`

Description: Due to the rebasable nature of the `LiquidStakingToken`, the token balances of the users are not constant, and they frequently change whenever the underlying assets change. Many DeFi protocols don't support rebasable tokens, so they need to be wrapped into a token that behaves like the standard `ERC20`. The users can use the `WrappedLiquidStakedToken` contract to `wrap(...)` and `unwrap(...)` their tokens. When calling the `unwrap(...)` function, the user specifies the `_amount` of wrapped tokens to convert back to liquid staking tokens. The function converts the `_amount`, which represents shares of LST, to `assets`. The `assets` variable is passed to the `lst.transfer(...)`, which would convert this amount back to shares and transfer those shares to the receiver.

Based on the current conversion ratio, the user will receive an `assets` amount of LST in return. The `unwrap(...)` function also has to burn the `_amount` of wrapped tokens taken from the user.

```
function unwrap(uint256 _amount) external returns (uint256) {
    // ...
    uint256 assets = lst.convertToAssets(_amount);

    // @audit-issue It should burn `_amount` of shares as opposed
    // to `assets` amount that represents LST tokens.
    _burn(msg.sender, assets);

    lst.transfer(msg.sender, assets);

    return assets;
}
```

The problem present in the `unwrap(...)` function is that it burns an incorrect amount of shares from the user. Instead of burning the `_amount` specified by the user, it burns the computed `assets` amount. Since the share price is ever-increasing, the burn amount will increase over time. The longer the user holds the wrapped token, the more he will lose once he unwraps. The transactions might also revert when the burn amount is higher than the user's current balance.

Recommendation(s): Consider burning the `_amount` of wrapped tokens specified by the user instead of the `assets` amount returned from the conversion.

Status: Fixed

Update from the client: Fixed: aa7f7ea

6.1.5 [Critical] The withdrawal amount decreases the `pendingDeposit` twice

File(s): `src/layer2/LiquidStakingTokenLockbox.sol`

Description: On the withdrawal action, the withdrawal amount is subtracted from the `pendingDeposit` (and, if needed, from the `totalStaked`). However, during this withdrawal, the withdrawn amount is also sent to the `L1`, where it is used to increase the `pendingWithdraw`. Then `pendingWithdraw` is sent back to the `L2` (e.g., during `rebase(...)`), where it decreases the `pendingDeposit` again. As a result, a single withdrawal decreases the `pendingDeposit` twice. Consider the following scenario:

- user `A` deposits on `L2` `2ETH`;

- current state in `LST`: `pendingDeposit = 2ETH`

- user `A` withdraws `1ETH`. Message to `L1` is sent, and so `pendingWithdraw` is increased by `1ETH`;

- current state in `LST`: `pendingDeposit = 1ETH`

- `rebase(...)` is called on `L1` and `pendingWithdraw=1ETH` is sent to the `L2`. The message is processed in `LiquidStakingToken`, and `pendingDeposit` is decreased by `pendingWithdraw`;

- current state in `LST`: `pendingDeposit = 0`

- user `A` is not able to withdraw the remaining `1ETH`;

Due to the artificial lowering of the `totalAssets`, the user won't be able to withdraw because the number of shares calculated in `convertToShares(...)` will be higher than the total shares minted and higher than the limit set in `RateLimiter`.

Recommendation(s): Consider not increasing `pendingWithdraw` in the `LiquidStakingTokenLockbox._lzReceive(...)`.

Status: Fixed

Update from the client: Fixed: 173a3a4

6.1.6 [High] Incorrect update of `RateLimiter` causes lock of funds on Mode Network

File(s): `src/layer2/L2ModeSyncPoolETH.sol`

Description: The `RateLimiter` contract tracks the withdraw limit, which is used to limit the amount of tokens that can be withdrawn to L1. Withdrawal limit increases on deposits from L1 by directly increasing `withdrawLimit`, or during a syncing process by increasing the limit by the value of unsynced limit `withdrawLimit += unsyncedLimit`. Withdrawing (on L2) causes the withdrawal limit to decrease and prevents withdrawals higher than the limit. The relevant code snippet with the described logic is presented below:

```
function updateRateLimit(
    address,
    address token,
    uint256 amountIn,
    uint256 amountOut
) external override {
    if (msg.sender == modeEth && token == modeEth) {
        if (amountIn > 0) {
            withdrawLimit += amountIn;
        } else {
            if (withdrawLimit < amountOut) {
                revert Errors.WithdrawLimitExceeded();
            }
            withdrawLimit -= amountOut;
        }
    } else if (msg.sender == modeEth && token == address(0)) {
        unsyncedLimit += amountIn;
    // @audit The branch below is supposed to be executed during `_sync(...)`
    } else if (msg.sender == syncPool && token == syncPool) {
        withdrawLimit += unsyncedLimit;
        unsyncedLimit = 0;
    }
}
```

However, during the syncing process in the `L2ModeSyncPoolETH._sync(...)` function, the `RateLimiter.updateRateLimit(...)` is called with incorrect parameters:

```
IRateLimiter(getRateLimiter()).updateRateLimit(
    address(this),
    address(0), // @audit: address(0) instead of address(this)
    amountIn,
    amountOut
);
```

The `token` parameter value is `address(0)`, which during the execution of `updateRateLimit(...)` won't trigger the last branch, with the condition `token == syncPool`. The `withdrawLimit` won't be increased by the `unsyncedLimit` during syncing. As a result, all the funds that were deposited on L2 (registered in `unsyncedLimit`) will be stuck in the contract since users won't be able to withdraw them.

Recommendation(s): Consider calling the `RateLimiter.updateRateLimit(...)` during the `_sync(...)` execution with the correct parameters. Additionally, as a best practice, add a revert condition in the new last branch in `updateRateLimit(...)`, which would prevent the function from silently failing.

Status: Fixed

Update from the client: Fixed: 172df42, dd7431d

6.1.7 [High] Rebase prevention

File(s): `src/layer2/LiquidStakingTokenLockbox.sol`

Description: A malicious actor may prevent moving funds on `L2` from `pendingDeposit` to `totalStaked`. Since only the `totalStaked` is being rebased, this action would prevent rebasing. This can be done by using the `L1` deposit mechanism with an incorrect `_receiver = address(0)` argument. This way, the deposit function in the `LiquidStakingTokenLockbox` (`L1`) would execute, but the `LiquidStakingToken._lzReceive(...)` (`L2`) would fail in `_mintShares(...)` because of the incorrect `_receiver`:

```
function _mintShares(
    address _recipient,
    uint256 _shares
) internal returns (uint256) {
    ////////////////////////////////////////////
    // @audit: Message receiver on L2 fails,
    // but the sender on L1 executed correctly.
    ////////////////////////////////////////////
    if (_recipient == address(0)) revert Errors.ZeroAddress();
    ...
}
```

As a result, the `Lockbox` considers `pendingWithdraw` value as sent, and so it is set to `0`:

```
function depositApxEth(...) external payable nonReentrant whenNotPaused {
    _sendDeposit(
        pxEthAmount,
        assetsPerShare,
        // @audit: The pendingWithdraw is considered as sent to L2.
        $.pendingWithdraw,
        _receiver,
        _refundAddress,
        address(autoPxEth),
        _options
    );
    // @audit: So now it can be set to 0.
    $.pendingWithdraw = 0;
}
```

But if the `_receiver` is `address(0)`, the `LiquidStakingToken._lzReceive(...)` would fail (on `L2`), and the `pendingWithdraw` won't be used to decrease the `pendingDeposit` and to increase the `totalStaked`. Consequently, the rebasing won't be done on the correct amount, leading to a loss of funds since the users won't be able to withdraw all the `pxEth` they hold. **Note** that the description of this issue assumes that the *Lack of totalStaked increase in _lzReceive(...)* issue was fixed.

Recommendation(s): Consider checking if the value of the provided `_receiver` parameter is not an `address(0)` in the `LiquidStakingTokenLockbox` before the cross-chain message is sent.

Status: Fixed

Update from the client: Fixed: 1346333

6.1.8 [High] The slow-sync process finalization can be DOSed

File(s): `src/layer1/LiquidStakingTokenLockbox.sol`

Description: The `depositSync(...)` function is used to reflect the state changes on the `L1` chain caused by the deposits done on the `L2` chain. During the fast-sync process, the `depositSync(...)` function transfers the unbacked `pxETH` tokens from the `L1SyncPoolETH` contract to the `LiquidStakingTokenLockbox` contract. It also increases the `pendingDeposit` variable to reflect the `pxETH` tokens that will be deposited into the `AutoPxEth` vault once the Ether arrives from the bridge to back them up.

```
function depositSync(uint256 _amount, bool _shouldCompound)
    external
    onlySyncPool
{
    // ...
    if (_shouldCompound && $.pendingDeposit > 0) {
        // @audit This branch will be executed later to finalize the slow-sync process.
        // ...
    } else if (!_shouldCompound) {
        // @audit This branch is executed during  the fast-sync process.
        $.pxEth.safeTransferFrom(msg.sender, address(this), _amount);
        $.pendingDeposit += _amount;
        // ...
    }
}
```

When the Ether is released from the bridge, the `_finalizeDeposit(...)` function from the `L1SyncPoolETH` contract will deposit the Ether into the `PirexEth` contract, burn the received (excess) `pxETH` tokens, and call the `depositSync(...)` function to reflect this state change. The `_amount` of `pxETH` tokens intended to be deposited into the `AutoPxEth` vault must be compared to the current value stored in the `pendingDeposit`. If there was an `L2` withdrawal of `pxETH` tokens, the `pendingDeposit` was decreased by the withdrawal amount. In such a case, the Lockbox would hold fewer `pxETH` tokens, and the deposit `amount` would need to be adjusted.

```
function depositSync(uint256 _amount, bool _shouldCompound)
    external
    onlySyncPool
{
    LockboxStorage storage $ = _getLockboxStorage();
    if (_shouldCompound && $.pendingDeposit > 0) {
        IAutoPxEth autoPxEth = $.autoPxEth;
        // @audit-note: The `amount` to be deposited is adjusted in case
        // the pendingDeposit changed e.g. due to the withdrawal.
        uint256 amount =
            _amount > $.pendingDeposit ? $.pendingDeposit : _amount;
        $.pendingDeposit -= amount;
        $.pendingWithdraw += amount;
        // @audit-issue: It should deposit the adjusted `amount`, not
        // the `_amount` passed in as a parameter.
        uint256 shares = autoPxEth.deposit(_amount, address(this));
        // ...
    } else if (!_shouldCompound) {
        // ...
    }
}
```

The problem present in the `depositSync(...)` function is that the value that is passed into the `autoPxEth.deposit(...)` call is not the adjusted `amount` but the bridged Ether `_amount` sent to this function by the Sync Pool. The deposit will attempt to transfer more `pxETH` tokens than the Lockbox currently holds, and the transaction will revert.

The described scenario will happen during the standard operation of the protocol whenever users withdraw `pxETH` tokens before the slow-sync process is finalized. As a result, the whole transaction initiated by the `CrossDomainMessenger` contract would revert. It would need to be repeated by interacting with the native bridge. This would only work after supplying the Lockbox with new `pxETH` tokens covering the missing deposit amount, e.g., by performing a deposit.

A malicious user might use this attack vector to perform a denial of service (DOS) attack to prevent the slow-sync process from finalizing by performing repeated withdrawals of `pxETH` tokens from the Lockbox. In both cases, the finalization would be rejected, and the Ether would not be used to earn the staking rewards. Note that the attacker doesn't lose funds during malicious behavior.

Recommendation(s): Consider using the adjusted `amount` when depositing `pxETH` tokens into the `AutoPxEth` vault in the `depositSync(...)` function.

Status: Fixed

Update from the client: Fixed: 42bcbf8

6.1.9 [Medium] Contracts cannot be paused

File(s): `src/layer2/LiquidStakingTokenLockbox.sol`, `src/layer2/LiquidStakingToken.sol`

Description: The two main contracts of the protocol are the `LiquidStakingTokenLockbox` and `LiquidStakingToken`. They coordinate the deposits and withdrawals of assets between the `L1` and `L2` chains. The core functions that the users will interact with implement the `whenNotPaused` modifier inherited from the `PausableUpgradeable` module from OpenZeppelin. In case of an emergency, it enables the pause of the deposit of assets and rebases on `L1`, as well as the withdrawal of assets on `L2`. The problem present in the current implementation is that the internal `_pause(...)` and `_unpause(...)` functions inherited from the `PausableUpgradeable` contract are not exposed in a public function. Without the public-facing wrapper, no one can call these internal functions. As a result, the protocol operation cannot be paused and later unpaused.

Recommendation(s): Consider wrapping the internal `_pause(...)` and `_unpause(...)` functions in new public functions `pause(...)` and `unpause(...)`. These functions should be access controlled so that only a trusted actor can call them e.g. the Governance.

Status: Fixed

Update from the client: Fixed: d1fdf02

6.1.10 [Medium] Deposit and sync can't be paused

File(s): `src/layer2/L2ModeSyncPoolETH.sol`

Description: The protocol considers a case in which all user interactions are paused. This is achieved by applying a `whenNotPaused` modifier to user-facing functions in `LiquidStakingTokenLockbox` and `LiquidStakingToken`. However, the external functions `deposit(...)` and `sync(...)` in `L2ModeSyncPoolETH` contract don't have this modifier and can't be paused.

Recommendation(s): Consider implementing a pausing mechanism to the `L2ModeSyncPoolETH` to allow pausing all protocol entry points in a critical situation.

Status: Fixed

Update from the client: Fixed: f47d0be

6.1.11 [Medium] Oracle may return a stale rate, which would decrease the `totalStaked`

File(s): `src/layer2/LiquidStakingToken.sol`

Description: Whenever the user deposits tokens on the `L2` chain through the `L2` Sync Pool, the `L2ExchangeRateProvider` contract will be called to get the latest `assetsPerShare` ratio from the `AutoPxEth` vault. After the LST shares are minted to the user, the `assetsPerShare` ratio returned from the oracle is used in a rebase to update the `totalStaked` value on `L2`.

```
// @audit _assetsPerShare is the ratio received from the Oracle
function mint(address _to, uint256 _assetsPerShare, uint256 _amount)
    external
    onlySyncPool
{
    // ...
    _mintShares(_to, shares);

    // @audit The ratio from the Oracle is used in a rebase.
    _updateTotalStaked(0, _assetsPerShare);

    // @audit Store this ratio as the most recent one.
    $.lastAssetsPerShare = _assetsPerShare;
    // ...
}
```

Due to the limitation in how frequently the Oracle data can be updated, the ratio used to update the `totalStaked` on `L2` during the deposit might be older (and smaller) than the most recent ratio sent from `L1`. This discrepancy can occur during an `L1` deposit or rebase. Consider the following scenario:

- The Keeper (Data Provider) updated the oracle with the current conversion ratio from the vault. Let's assume the ratio is `1:2`, meaning that 1 share is worth 2 tokens;
- The staking rewards arrived in the vault, and someone performed a deposit or called a `rebase(...)`. The ratio in the vault on `L1` changed to `1:3`, and this information was sent and stored on `L2` as the most recent `$.lastAssetsPerShare`. The assets are rebased with the new, higher ratio;
- Before the Keeper updates the oracle with the new ratio, a user performs a deposit on `L2`. The `mint(...)` function will use the older, lower ratio to perform a rebase and store this ratio as the most recent one in `$.lastAssetsPerShare`;

As a result, any subsequent user interactions on `L2` will use the stale ratio of `1:2`. Deposits will be cheaper, and withdrawals will be more expensive in terms of shares taken from the user, which would cause temporary loss of rewards.

Recommendation(s): Before performing a rebase or storing the most recent vault ratio on `L2`, consider checking that the ratio returned from the oracle is not lower than the currently stored one.

Status: Fixed

Update from the client: Fixed: 9a14e8c

Update from the Nethermind Security: The current solution reverts deposit on `L2` if the rate fetched from the oracle is lower than the currently stored in the `LiquidStakingToken`. This will result in users' deposits being rejected until the fresh rate is pushed to the oracle. However, `L2` deposits will be blocked again with the next `L1` deposit, bringing a new (potentially higher) rate. If the new records were pushed much more frequently than the rewards update in the `apxETH` vault, this solution would be sufficient, as the reverts wouldn't happen often, and users would be protected from the stale rate. However, if the opposite, consider updating the `lastAssetsPerShare` only if `$.lastAssetsPerShare < _assetsPerShare` to avoid reverts and still use the most recent rate available.

Update from the client: Fixed. Not revert on lagged rates.

6.1.12 [Medium] The ETH bridge will fail if the Pirex deposit fee is non-zero

File(s): `src/layer2/L1SyncPoolETH.sol`

Description: The `_finalizeDeposit(...)` function in the `L1SyncPoolETH` contract is called when the `ETH` is bridged from the Mode to Ethereum. This function will deposit the received `ETH` to the `PirexEth` contract. Next, it will burn the received `PxETH` tokens. Then the `depositSync(...)` in the `LiquidStakingTokenLockbox` contract is called, which will deposit locked `PxETH` tokens into the `AutoPxEth` vault. Below, we present the described fragment of code:

```
function _finalizeDeposit(...) internal virtual override {
    if (tokenIn != Constants.ETH_ADDRESS) revert Errors.OnlyETH();
    if (amountIn != msg.value) revert Errors.InvalidAmount();

    L1BaseSyncPoolStorage storage $ = _getL1BaseSyncPoolStorage();

    $.totalUnbackedTokens > amountIn
        ? $.totalUnbackedTokens -= amountIn
        : $.totalUnbackedTokens = 0;
    // sent the ETH to PirexETH
    IPirexEth(getPlatform()).deposit{value: amountIn}(address(this), false);
    ////////////////////////////////////////////////////////
    // @audit: incorrect assumption that minted pxETH == amountIn
    ////////////////////////////////////////////////////////
    // burn the pxEth
    IDineroERC20(getTokenOut()).burn(address(this), amountIn);

    // notify the lockbox to deposit up to amountIn into the vault
    LiquidStakingTokenLockbox(getLockBox()).depositSync(amountIn, true);
}
```

The issue arises from the fact that the `_finalizeDeposit(...)` function assumes that the minted amount of `PxETH` is always equal to the deposited ETH, i.e., `amountIn`. This is only true if the deposit fee in the `PirexEth` contract is `0` (which is the current state). However, if the deposit fee changes to a non-zero value (set by the governance) the received amount of `pxETH` will be less than the deposited ETH in `amountIn`, since the fee amount of `pxETH` is sent to the fee recipient. In consequence, the `burn(address(this), amountIn)` would fail since there won't be enough `pxETH` tokens present in the contract, and the bridging procedure won't be able to finalize.

Recommendation(s): The incorrect assumption can't be resolved by using `postFeeAmount` for both `burn(...)` and `depositSync(...)`, e.g.:

```
////////////////////////////////////////////////////////
// using the postFeeAmount that represents received pxETH
////////////////////////////////////////////////////////
(postFeeAmount, _) = IPirexEth(getPlatform()).deposit{value: amountIn}(address(this), false);
    // burn the pxEth
IDineroERC20(getTokenOut()).burn(address(this), postFeeAmount);

// notify the lockbox to deposit up to amountIn into the vault
LiquidStakingTokenLockbox(getLockBox()).depositSync(postFeeAmount, true);
```

Even though now the function would not revert for a non-zero deposit fee, such a solution is still incorrect. This is because the amount of `pxETH` tokens that was minted in `_anticipatedDeposit(...)` to the `LiquidStakingTokenLockbox` was also higher (without deducing the deposit fee). The call `depositSync(postFeeAmount, true)` (in `_finalizeDeposit(...)`) would not burn all the "dummy" `pxETH`. In consequence, there would be more `pxETH` in the system than `ETH`, which would break the `1pxETH:1ETH` invariant. Therefore, we propose to burn `postFeeAmount` in `_finalizeDeposit(...)` as described above but then call `depositSync` with the `amountIn` to burn all the `pxETH` minted in `_anticipatedDeposit(...)`.

Status: Fixed

Update from the client: Fixed: b9a7cb4

6.1.13 [Medium] `RateLimiter` may be set to `address(0)`

File(s): `src/layer2/WrappedLiquidStakedToken.sol`

Description: The `RateLimiter` contract ensures that only the synced funds can be withdrawn. It is optionally called in `LiquidStakingToken` and `L2ModeSyncPoolETH`. If the `RateLimiter` contract is being used, it imposes the following user interaction flow on `L2`: deposit, sync, and withdraw. However, if the address of the `RateLimiter` is not set, users may call withdraw without the sync. This would lead to correctly executing the `withdraw(...)` function in the `LiquidStakingToken`. The `LiquidStakingTokenLockbox._lzReceive(...)` would also execute and release the `pxETH` to the receiver if there are enough `pendingDeposit` or `totalShares`. However, if there are not enough unbacked `pxETH` tokens stored in the Lockbox, they will be withdrawn from the `AutoPxEth` vault. This should not happen since those withdrawn funds should be paid in the unbacked `pxETH` minted during the fast sync. Additionally, if the `Lockbox` doesn't have enough apxETH, withdrawing from the vault would be impossible. As a result, the withdrawal would succeed on `L2` but fail on `L1`.

Recommendation(s): Consider setting the non-zero address of the `RateLimiter` during the initialization of the `LiquidStakingToken` and `L2ModeSyncPoolETH` contracts. Additionally, consider removing the optional call to `RateLimiter` since there is no scenario in which the lack of it allows the protocol to work correctly.

Status: Fixed

Update from the client: Fixed: cec8ba6

6.1.14 [Low] Add transaction `ID` to ensure correct cross-chain message finality

File(s): `src/layer2/LiquidStakingTokenLockbox.sol`

Description: One of the core assumptions of the project is that during the synchronization, the `LayerZero` fast message always reaches `L1` faster than the native bridge slow message. This is the most probable case. However, there are potential scenarios where the slow message reaches `L1` before the fast message:

- error on `L1` while receiving a fast message;
- incorrect configuration of gas on `lzReceive(...)` execution;
- in integrations with `L2s` other than `Mode Network`:;

- there may be a lower number of quality `DVNs`, which increases the risk of `DVNs` going down - faster native bridge increases the possibility of the `LZ` message reaching later

The incorrect order of the received messages would result in a lack of update of the `totalShares` and `pendingWithdraw` variables, which would lead to a loss of users' funds.

Recommendation(s): To mitigate such a scenario, consider ensuring that each call to `sync(...)` on `L2` results in correct calls on `L1` to: first `_anticipatedDeposit` and second to `_finalizeDeposit`, in the `L1SyncPoolETH`. This may be done by including `transaction ID` to both slow and fast messages and then ensuring they are consumed in the correct order in the `L1SyncPoolETH`.

Status: Fixed

Update from the client: Added: cfccf59

6.1.15 [Low] The L1 Ether deposits will revert for non-zero PirexETH deposit fee

File(s): src/layer2/LiquidStakingTokenLockbox.sol

Description: The depositEth(...) function from the LiquidStakingTokenLockbox contract is meant to be used by users to deposit L1 Ether and relay the message to L2. The user specifies the _amount of Ether he wishes to deposit and sends the funds as msg.value. The native tokens sent in the transaction must also cover the nativeFee paid to LayerZero for relaying the message.

The function first deposits _amount of Ether into the PirexETH contract from where the deposit fee is subtracted. The resulting postFeeAmount is what is meant to be sent to L2 in the _sendDeposit(...) function.

```
function depositEth(
    address _receiver,
    address _refundAddress,
    uint256 _amount,
    bytes calldata _options
) external payable nonReentrant whenNotPaused {
    if (msg.value <= _amount) revert Errors.InvalidAmount();
    // ...
    // @audit Deposit _amount of ETH in PirexETH.
    // It means that we have msg.value - _amount left to cover the LayerZero fee
    (uint256 postFeeAmount,) =
        $.pirexEth.deposit{value: _amount}(address(this), true);
    // ...
    // @audit postFeeAmount is _amount - PirexETH deposit fee
    _sendDeposit(
        postFeeAmount,
        // ...,
        _receiver,
        _refundAddress,
        address(0), // ETH
        _options
    );
    // ...
}
```

The problem present in the _sendDeposit(...) function is that to compute the LayerZero nativeFee, the postFeeAmount is used instead of the actual Ether amount left to be used in the transaction.

```
function _sendDeposit(
    uint256 _amount,
    // ...
    address _receiver,
    address _refundAddress,
    address _asset,
    bytes calldata _options
) internal {
    // ...
    // @audit _amount is postFeeAmount which is smaller than
    // initial deposit amount. As a result calculated nativeFee will be bigger.
    uint256 nativeFee =
        _asset == address(0) ? msg.value - _amount : msg.value;
    // ...
}
```

Consider the following scenario: The user wants to deposit 10 ETH into the contract. He calculated that the LayerZero native fee will be 1 ETH, so he sends 11 ETH in total as msg.value. The 10 ETH gets deposited as _amount into the PirexEth contract. The Pirex deposit fee was 1 ETH, so the returned postFeeAmount is 9 ETH.

At this point in time, 10 out of 11 ETH sent have already been deposited into the PirexEth contract. The _sendDeposit(...) function will calculate the LayerZero nativeFee based on the postFeeAmount as 11 - 9 = 2 ETH. Since there is only 1 Ether left to spend, the deposit transaction will revert.

The situation described above won't cause any problems as of now since the deposit fee is currently set to 0 in the PirexEth contract. However, as soon as the governance decides to charge the users a deposit fee, the deposits in the LiquidStakingTokenLockbox contract will start reverting.

Recommendation(s): Consider revisiting the logic to calculate the LayerZero native fee to use the Ether amount based on the msg.value originally sent by the user and the _amount of assets to be deposited as opposed to the postFeeAmount.

Status: Fixed

Update from the client: Fixed: 52f8de1

6.1.16 [Low] `LayerZero` does not enforce the correct order of message consumption

File(s): `src/layer2/LiquidStakingTokenLockbox.sol`

Description: The `LayerZero` allows for unordered execution of the verified messages, meaning that messages sent later may be executed first. If the messages are executed as soon as verified and not in the order defined by nonces, the outcome may be different than expected. The `RateLimiter` enforces to users that the funds deposited natively on `L2` have to be synchronized before the withdrawal. However, if the `Executor` does not follow the nonce order, the withdrawal on `L1` may be made before the sync. While the described situation is unlikely to happen, it may lead to the problems described in the issue: " `RateLimiter` may be set to `address(0)`".

Recommendation(s): This problem can be solved by using a custom `Executor` or enforcing on-chain ordered delivery.

Status: Fixed

Update from the client: Added `addExecutorOrderedExecutionOption` to enforced Lockbox receive options a587dae and implement nonce enforcement in lockbox contract 6783def

Update from the Nethermind Security: Introduced changes do not solve the described problem. Current checks ensure that withdrawals are executed in order on `L1`. However, it does not ensure that if on `L2`, the user called `withdraw(...)` and `sync(...)`, they will be executed in the same order as on `L1`. Those two messages are sent from different senders on `L2`, `LiquidStakingToken`, and `L2ModeSyncPoolETH`, and consumed by two different receivers on `L1`: `LiquidStakingTokenLockbox` and `L1ModeReceiverETH`. To ensure that all messages sent from `L2` are consumed in the correct order on `L1`, all contracts on both layers need a common transaction ID that would be consistently sent from both `L2` contracts and checked in both `L1` contracts.

Update from the client: After merging `L2SyncPool` and `LiquidStakingToken`, there is only one sender from `L2` to `L1`, and the messages are enforced to be executed in the correct order in the `LiquidStakingTokenLockbox`.

6.1.17 [Info] Incorrect call to `RateLimiter` during the deposit on the L2 sync pool

File(s): `src/layer2/L2BaseSyncPoolUpgradeable.sol`

Description: In the `L2BaseSyncPoolUpgradeable.deposit(...)` the `RateLimiter.updateRateLimit(...)` is called. No execution will happen, because the provided arguments are incorrect. Also, even if the provided arguments were correct, the same update of the `RateLimiter` is done in the `LiquidStakingToken.mint(...)`, therefore the additional call to `RateLimiter` would be incorrect.

Recommendation(s): Consider perfomring call to `RateLimiter` either in `L2BaseSyncPoolUpgradeable.deposit(...)` or in `LiquidStakingToken.mint(...)`.

Status: Fixed

Update from the client: Fixed. Removed RateLimiter call in `L2BaseSyncPoolUpgradeable.deposit(...)`: 09a5514

6.2 Issues found during fix review phase

6.2.1 [High] Incorrect move of funds from pending to staked state

File(s): `L2ModeSyncPoolETH.sol`

Description: The `pendingDeposit` represents funds deposited on `L2` that have not yet been staked on `L1`. The `finalizedDeposit` is sent from `L1` to inform `LST` that pending funds are now being staked, and in `LST`, the `pendingDeposit` can be moved to `totalStaked` - which is being rebased. However, the `pendingDeposit` on `L2` includes both synchronized and unsynchronized funds. This can lead to treating a pending deposit not sent to `L1` yet as actively being staked. Consider the following scenario:

- user `A` deposits `100ETH` on `L2`;

- L1 state: `pendingDeposit = 0`, `finalizedDeposit = 0` - L2 state: `pendingDeposit = 100ETH`, `totalStaked = 0`

- funds are synchronized, both fast and slow;

- L1 state: `pendingDeposit = 0`, `finalizedDeposit = 100ETH` - L2 state: `pendingDeposit = 100ETH`, `totalStaked = 0`

- user `A` withdraws `100ETH` on `L2` and receives the equivalent in `pxETH` on `L1`;

- L1 state: `pendingDeposit = 0`, `finalizedDeposit = 100ETH` - L2 state: `pendingDeposit = 0`, `totalStaked = 0`

- another user, `B` deposits `100ETH` on `L2`, but does not sync;

- L1 state: `pendingDeposit = 0`, `finalizedDeposit = 100ETH` - L2 state: `pendingDeposit = 100ETH`, `totalStaked = 0`

- rebase is called and `finalizedDeposit = 100ETH` is sent from `L1` to `L2`. It moves the current `pendingDeposit -= 100ETH` from pending to staked, so `totalStaked += 100ETH`;

In the scenario described above, the funds that were deposited on `L2` by user `B` are treated as staked, even though they are not being staked. Those funds will be rebased but should not be, resulting in an incorrect increase of existing shares' worth.

The issue arises from two problems. First is that the `pendingDeposit` in `LiquidStakingToken` actually represents two things: 1) tokens that are not yet staked but are synchronized (in the process of bridging to `L1` or already bridged) and 2) tokens that are not yet staked and also not synchronized. The lack of this distinguishment doesn't allow for managing only synced funds. The second problem arises from the fact that the `finalizedDeposit` sent from `L1` to `L2` is not specific to the funds that were bridged to `L1`. Because of this, even if we split `pendingDeposit` into synced and unsynced and use `finalizedDeposit` only to update synced funds, the synced funds will still be affected by a similar problem as described in this issue. Imagine the scenario where user `A` deposits funds on `L2`, sync is called (not finalized yet), user `A` withdraws funds, and another user `B` deposits funds on `L2` and syncs. After the first sync is finalized, the `rebase(...)` is called, and `finalizedDeposit` is sent from `L1` to `L2`. It moves synced pending deposit to `totalStaked`, but in fact, those funds are not being staked but are still being bridged.

Recommendation(s): To fix the described issue, the two problems mentioned above need to be addressed. The first problem can be solved by splitting `pendingDeposit` into two variables, e.g., `syncedPendingDeposit` and `unsyncedPendingDeposit`. The `unsyncedPendingDeposit` is increased during `L2` deposits. During the sync, all funds from `unsyncedPendingDeposit` are moved to `syncedPendingDeposit`. When funds in `syncedPendingDeposit` reach `L1`, the `finalizedDeposit` should move them to `totalStaked`. The second problem can be solved in two ways:

- Ensure that `sync(...)` is called after the previous slow message is delivered. This can be enforced by allowing the next call to `sync(...)` after, e.g., ten days after the previous `sync(...)` call. The specified time must be longer than native bridge processing. Applying this check would enforce flow: 1) sync funds to `L1`, 2) deposit funds on `L1` to staking, 3) send information back to `L2` that funds in `finalizedDeposit` are staked. This solution is simple to implement but has drawbacks that need consideration: it introduces some time in which a group of users may not be able to withdraw funds and assumes constant time of native bridging;
- Introduce a queue, which would ensure that `finalizedDeposit` is specific to funds that were sent during synchronization. During synchronization, the current funds from `unsyncedPendingDeposit` can be added to the queue. All queue elements would constitute `syncedPendingDeposit`. When the synchronization is finalized, the information sent to `L2` in `finalizedDeposit` will be applied only to the first element of the queue and then removed this element. Users who want to withdraw funds would withdraw from elements of the queue, starting from the first one, and if more is needed, continue to the following elements. During withdrawal, only the balance of elements in the queue is changed, but elements are not removed. So if all funds from the first element are used, its balance is zero, but this element will be only removed when the `finalizedDeposit` reaches the `L2`;

Status: Fixed

Update from the client: Split `pendingDeposit` on L2 side between `unsyncedPendingDeposit` and `syncedPendingDeposit` as suggested: 73f5957

Update from the Nethermind Security: Applied changes solve only part of the problem. Moreover, the current solution is implemented incorrectly since funds from `unsyncedPendingDeposit` never go to `syncedPendingDeposit`.

Update from the Nethermind Security: After the further changes, the mechanism for moving funds from pending to staked state has been heavily refactored, and the problem described is fixed.

6.2.2 [High] The `_calculateWithdrawalAmount(...)` should not be done on `pendingDeposit`

File(s): `LiquidStakingTokenLockbox.sol`

Description: The withdrawal process starts on `L2` by calling `LiquidStakingToken.withdraw(...)`. Funds are first taken from the `pendingDeposit` and only later from the `totalStaked`. Next, on `L1`, the `LiquidStakingTokenLockbox` receives the withdrawal message in the `_lzReceive(...)`. It follows the same order as on `L2`: first, the `pendingDeposit` is used, and later funds from the `apxETH` vault. However, in `_lzReceive(...)`, the `_amount` of assets is first passed to `_calculateWithdrawalAmount(...)`. In this function, the `_amount` of assets is transformed in the following ways:

- the `_amount` is rebased with the newest rate;

```
uint256 assets = _amount.mulDivDown(assetsPerShare, _assetsPerShare);
```

- the withdrawal penalty is applied;

```
uint256 previewRedeem = assets - penalty;
```

Those actions are only correct for funds that are staked (rebasing) and are in the vault (withdrawal penalty). But the passed parameter `_amount` may represent funds in pending deposits. As a result, the following problems arise:

- The funds from `pendingDeposit` (not staked) are rebased but should not be. As a result, those funds are artificially increased;
- The withdrawal penalty is applied on `pendingDeposit`. Consider the following scenario: user `A` deposits `100ETH` on `L2`, calls sync (only fast message is delivered), and withdraws `100ETH`. In `Lockbox._lzRecieve(...)`, the `5%` withdrawal penalty is applied, and the user receives `95pxETH`, and `5pxETH` stays in `pendingDeposit`. A Slow message arrives, and in `depositSync(...)`, this `5pxETH` is used as a deposit to the `AutoPxEth` vault, minting an according amount of shares. However, the minted `apxETH` are locked in the `Lockbox` and can't be used by anyone. On `L2`, the user has burned all of his LST shares, which means that on `L1`, no `apxETH` shares should be minted at all. The newly minted 5 `apxETH` is not "owned" by anyone on `L2`. This discrepancy between `apxETH` shares held by the Lockbox and the sum of LST shares held by `L2` users can increase over time;

Recommendation(s): Consider calling `_calculateWithdrawalAmount(...)` only after using funds in `pendingDeposit`, so that rebasing and withdrawal fee are done on the staked funds. Note that the `_calculateWithdrawalAmount(...)` must be changed so that `pendingDeposit` is not added to the `totalAssets`.

Status: Fixed

Update from the client: Fixed: Only applying `withdrawPenalty` over staked assets 854b93

6.2.3 [Medium] Increasing deposit fee during the bridging of funds causes mint of unbacked `pxETH`

File(s): `L1SyncPoolETH.sol`

Description: The unbacked `pxETH` tokens can be minted if the deposit fee (on `PirexEth`) is increased between the delivery of fast and slow sync messages. This would happen because the amount of `pxETH` minted after the fast message is received would be higher than the amount of `pxETH` burned when the slow message is received. As a result, `pxETH` tokens would be minted, but they wouldn't be backed by `ETH`.

Recommendation(s): In the current design, the problem can't be easily mitigated by smart contract level changes. Therefore, we propose to:

- Increase the deposit fee only when there are no currently bridged funds. This can be additionally enforced by pausing `L2ModeSyncPoolETH`;
- Alternatively, burn the unbacked fee amount of `pxETH` since it is sent to the treasury address controlled by the protocol team;

Status: Mitigated

Update from the client: Acknowledged. We don't plan to ever increase the deposit fee, but in the case we do, we'll first pause l2 deposits to clear the sync queue.

6.2.4 [Medium] L2 deposits are allowed when the protocol is paused

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The `_mint(...)` function from the `LiquidStakingToken` contract is used to mint the new LST shares in the `L2` deposit flow. With the latest change in the inheritance structure, the `LiquidStakingToken` contract inherits the `L2SyncPool` contract. The visibility of the old external `mint(...)` function has been changed to the new internal `_mint(...)` function. However, the new function signature differs not only in visibility but also in the modifiers invocated. Minting was not allowed previously when the protocol was paused. The new `_mint(...)` function does not have the `whenNotPaused` modifier, which enables users to deposit funds on `L2` while the rest of the protocol is paused.

Recommendation(s): Consider adding the `whenNotPaused` modifier to the `deposit(...)` function to prevent users from interacting with the smart contract during emergencies.

Status: Fixed

Update from the client: Fixed: c534b15

6.2.5 [Medium] Unnecessary centralization risk in `sweep(...)` function

File(s): `src/layer2/L1SyncPool.sol`

Description: The lockbox now inherits the `L1SyncPool` contract. In the previous version, the sync pool was deployed as a standalone contract. The previously used `L1BaseSyncPoolUpgradeable` contract had a `sweep(...)` function to transfer any remaining tokens left in the sync pool contract. This function was there under the assumption that the sync pool is not supposed to hold any tokens in the long term since this was the responsibility of the Lockbox. However, this assumption no longer holds with the new change in the inheritance structure. Since the `LiquidStakingTokenLockbox` contract is supposed to hold `apxETH` shares long-term, the newly inherited `sweep(...)` function can be used to withdraw all the `apxETH` tokens from the contract. If the admin (governance) account were to be compromised, all funds could be stolen, and users would be left with LST shares of no value.

Recommendation(s): If the ability to transfer any token from the `LiquidStakingTokenLockbox` contract is not required, consider disallowing this functionality.

Status: Fixed

Update from the client: Removed: 239db71

6.2.6 [Low] Incorrect calculation of rebase fee

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The rebase fee is taken during the rebasing in the `_lzReceive(...)` function. A relevant fragment of the function is presented below:

```
// ...
} else if (_messageType == Constants.MESSAGE_TYPE_REBASE) {
    _updateTotalStaked(0, _assetsPerShare);

    uint256 fee = _amount.mulDivDown(
        $.rebaseFee,
        Constants.FEE_DENOMINATOR
    );

    uint256 shares;
    if (fee > 0) {
        // @audit The fee is higher than it should be,
        // because we divide by lower assets (minus fee).
        shares = fee.mulDivDown(_totalShares, _totalAssets() - fee);

        _mintShares($.treasury, shares);
    }
// ...
```

The maximum rebase fee is `20%` (`200_000/1_000_000`). The amount of shares minted to a treasury as a fee is calculated by `fee.mulDivDown(_totalShares, _totalAssets() - fee)`. The problem is that the maximum fee in the current equation results in an amount that is higher than `20%`. Assume that `rebaseFee = 200_000` (maximum fee). The computed `fee` value would be `20%` of the `_amount`. To express this number of assets in shares, the following calculation should be done: `fee * _totalShares/_totalAssets()`. However, the actual computation subtracts the `fee` from `_totalAssets()`, which would decrease the divisor and increase the computed number of shares. The resulting fee value may be higher than the restricted `20%`.

Recommendation(s): Consider removing the fee subtraction from the divisor.

Status: Fixed

Update from the client: Fixed: 532c239

6.2.7 [Low] Pirex deposit fee updates require an upgrade of `L2ExchangeRateProvider`

File(s): `src/layer2/L2ExchangeRateProvider.sol`, `src/layer2/L2SyncPool.sol`

Description: The `deposit(...)` function in the `L2SyncPool` contract mints the LST to the user in exchange for the provided `L2` native tokens. The conversion ratio between the provided `amountIn` and the received `amountOut` is taken from the `L2ExchangeRateProvider` contract's `getConversionAmount(...)` function. Currently, this function does not consider the Pirex deposit fee. The computed `amountOut` is equal to the provided `amountIn`. This means that for the provided native tokens, the user will receive LST in a `1:1` ratio. However, if the Pirex deposit fee were ever increased, the current implementation of the `L2ExchangeRateProvider` would still return `amountIn` from `getConversionAmount(...)`, and as a result unbacked `pxETH` tokens will be minted similarly to the scenario described in *[Medium] Increasing deposit fee during the bridging of funds causes mint of unbacked* `pxETH`. Note that to solve this problem the `L2ExchangeRateProvider` would have to be upgraded.

Recommendation(s): Consider including the logic to handle the Pirex deposit fee inside the `getConversionAmount(...)` function along with an access-controlled setter function to control the fee. As of now, it will be set to the value of `0`. If that assumption would change in the future, the only required action would be to update the fee using the setter as opposed to an implementation contract upgrade.

Status: Fixed

Update from the client: Added fee calculation through `rateParameters.depositFee` in `getConversionAmount(...)`. It has a different precision from PirexEth but should still provide an adequate solution. Commit: d463e08

6.2.8 [Low] The `_updateSyncQueue(...)` function might cause the LayerZero transaction to run out of gas

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The `_updateSyncQueue(...)` function updates the `L2` state based on the `finalizedDeposit` sent from `L1` via LayerZero. Given the ID of the last finalized deposit on `L1`, the `_updateSyncQueue(...)` function will iterate over all sync queue elements with indexes smaller than or equal to the provided ID. The problem in the current implementation is that the number of processed indexes is unbounded. Consider the following scenario:

- A malicious user (e.g., a well-funded market competitor of Dinero) performs 100 `L2` deposits of `minSyncAmount` of `L2` Ether;
- Each deposit is followed by an immediate call to `sync(...)`;

As a result, 100 distinct native bridge messages were sent. If these were performed in short succession, after seven days, they would all be received and finalized on `L1` almost simultaneously. All deposits will be added to `finalizedDeposit`, and the ID of the last deposit received will be saved in `lastSyncId`. On the next `L1` rebase or deposit, the `finalizedDeposit`, and `lastSyncId` will be sent to `L2` to be processed by the `_updateSyncQueue(...)` function.

If we assume that `L1` received the deposits in order, the `lastSyncId` will hold the ID of the 100th deposit. The `for` loop in the `_updateSyncQueue(...)` function will start at the 100th deposit and will iterate 99 more times to process all the elements that need to be updated. Since this transaction is initiated by LayerZero, if it happens to hit the specified gas limit, it will revert. Once the allowed LZ gas limit is increased, the transaction will need to be retried. Transactions with deposit IDs higher than 100 will continue to revert on the `L2` side until the gas limit is adjusted.

Theoretically, if the attacker is willing to sacrifice `L2` funds to cover the gas and deposit costs, the attacker can repeat this process to increase the index far enough to make the loop execution exceed the `L2` block gas limit. This would result in a permanent DOS and force an upgrade of the protocol.

Recommendation(s): Consider implementing a pagination mechanism to split the execution into manageable chunks so the transaction won't exceed the LayerZero gas limit set during deployment. This could be implemented in the `LiquidStakingTokenLockbox`, where only a defined maximum amount of IDs would be sent to `L2`.

Status: Fixed

Update from the client: Fixed: b265f1b

Update from the Nethermind Security: While the new batching mechanism constrains the first `for` loop in the `_updateSyncQueue(...)` function, the second `for` loop remains unbounded. If the attacker follows the `deposit` -> `sync` process with an immediate `withdraw`, the second loop won't break until all queue elements are processed. This occurs because the immediate withdrawal uses the funds from the `$.syncIndexPendingAmount[i]`, so that the `break` condition `$.syncIndexPendingAmount[i] > 0` won't be fulfilled, and the loop will continue iterating until it finds an element that has not been withdrawn yet.

Update from the client: Fixed: 8a4558d

6.2.9 [Low] The rebase fee might be applied even if there are no funds staked

File(s): `src/layer2/LiquidStakingToken.sol`

Description: The `LiquidStakingTokenLockbox.rebase(...)` function calculates the rebase fee, which is later used to mint additional shares.

The rebase fee is computed based on the increase of the currently staked funds. When this amount is sent to `L2`, the staked funds on `L2` might have already been decreased, but the rebase fee computed previously on `L1` will still be applied. This may cause an issue in the following scenario:

- user `A` deposits funds to `LST`, and the funds are in unsynced pending state;
- user `B` withdraws all the staked funds from the `LST`;
- before the message from `L2` is received, the rebase is called on `L1`;
- the rebase fee is applied on `LST` to the unsynced pending funds, even though they are not being rebased;

Recommendation(s): Consider taking a rebase fee on `L2` only if the staked funds are non-zero.

Status: Acknowledged

Update from the client: The rebase fee is applied over the `amount` sent from L1 to L2, which represents rewards accrued by the staked amount on L1. If the staked amount is zero, the rebase fee is also zero. If the staked amount is non-zero when the `LiquidStakingLockbox.rebase()` is called, but it is zero when the msg is received on L2, the rebase fee will be applied as it should be. This is because it represents a portion of the rewards that was already accrued by the staked amount on L1. In this case, if the user managed to withdraw before the rebase msg reaches L2, they will not receive the yield accrued from the staked amount, because the withdrawal amount doesn't rebase.

6.2.10 [Info] Implementation contracts can be initialized

File(s): `LiquidStakingTokenLockbox.sol`, `L1ModeReceiverETH.sol`, `L1SyncPoolETH.sol`, `L2ExchangeRateProvider.sol`, `L2ModeSyncPoolETH.sol`, `LiquidStakingToken`, `WrappedLiquidStakedToken.sol`

Description: The implementation contracts do not disable the `initialize(...)` functions. This allows anyone to call `initialize(...)` with arbitrary values, which may lead to unwanted behavior. One example of such malicious behavior is initiating a call from the implementation contracts to sanctioned or black-listed contracts, which may lead to legal issues for the project.

Recommendation(s): Consider removing the possibility of calling the initializers directly on the implementation contracts. This may be done by calling the `_disableInitializers(...)` in the constructor, which will disable calling functions with the `initializer` modifier.

Status: Fixed

6.2.11 [Info] Incorrect event emission

File(s): `LiquidStakingToken.sol`

Description: The `_amount` emitted in the `Withdrawal` event is different from the one used in the `payload` sent via `_lzSend(...)`

```
function withdraw(
        address _receiver,
        address _refundAddress,
        uint256 _amount,
        bytes calldata _options
) external payable nonReentrant whenNotPaused {
        //...
        bytes memory payload = abi.encode(
                //...
                _amount,
                //...
        );
        //...
        MessagingReceipt memory msgReceipt = _lzSend(
                //...
                payload,
                //...
        );
        //...
        // @audit-issue emitted amount differs from the sent one
        _amount -= pendingDeposit;
        //...
        emit Withdrawal(msgReceipt.guid, msg.sender, _receiver, _amount);
}
```

Recommendation(s): Consider caching the `_amount` before subtractions and emit the cached value.

Status: Fixed

6.2.12 [Info] Incorrect update of `totalUnbackedTokens`

File(s): `L1SyncPoolETH.sol`

Description: The `totalUnbackedTokens` variable is increased by `amountOut` in the `_anticipatedDeposit(...)`, but in `_finalizeDeposit(...)`, it is decreased by `amountIn`. If the deposit fee on `PirexEth` is non-zero, the `amountIn != amountOut`, so the `totalUnbackedTokens` is incorrectly updated during the finalization of the sync process. Note that the `totalUnbackedTokens` variable is not used in any other process, so the incorrect update has no direct impact.

Recommendation(s): Consider decreasing `totalUnbackedTokens` by `amountOut`.

Status: Fixed

6.2.13 [Info] No check of zero amount in `_update(...)`

File(s): `DineroERC20RebaseUpgradeable.sol`

Description: The `_update(...)` takes the amount of assets, converts it to shares, and then transfers those shares. However, if the computed amount of shares is `0`, the function proceeds. During `transferFrom(...)`, this may result in spending allowance without sending any shares.

Recommendation(s): Consider reverting if the computed `shares` are zero in the `_update(...)` function.

Status: Fixed

6.2.14 [Info] Withdrawals might revert out of gas if `minSyncAmount` is low

File(s): `src/layer2/LiquidStakingToken.sol`

Description: During the withdrawals, funds are first withdrawn from the pending deposits and then from the staked funds. The `_withdrawPendingDeposit(...)` handles the former. It iterates over the indexes that are pending and empties them one by one until the `_withdrawAmount` requested by the user is fulfilled.

```
function _withdrawPendingDeposit(uint256 _withdrawAmount)
    internal
    returns (uint256)
{
    // ...
    uint256 lastPendingIndex = $.lastPendingSyncIndex;
    uint256 lastCompletedIndex = $.lastCompletedSyncIndex;
    uint256 remaining = _withdrawAmount;

    // @audit If there were many small deposits, this loop will iterate
    // many times and could run out of gas.
    for (uint256 i = lastCompletedIndex + 1; i <= lastPendingIndex; i++) {
        uint256 pendingAmount = $.syncIndexPendingAmount[i];

        if (pendingAmount > remaining) {
            $.syncIndexPendingAmount[i] -= remaining;
            remaining = 0;
            break;
        }

        remaining -= pendingAmount;
        $.syncIndexPendingAmount[i] = 0;
    }
    // ...
}
```

The problem present in the `_withdrawPendingDeposit(...)` function is similar to that described in the *[Low] The `_updateSyncQueue(...)` function might cause the LayerZero transaction to run out of gas.* A malicious user may perform many small L2 deposits and sync the minimum amount, which currently is `0.01 ETH`. As a result, the `syncIndexPendingAmount` array would be filled with large amount of records holding small amount of `ETH`. A large withdrawal would cause iterating over many items of the array, which would cost much gas, and could block users from withdrawing due to out-of-gas error. The described issue is of informational severity since smaller withdrawals will still go through. The likelihood of this scenario happening is directly correlated with the `minSyncAmount` set by the protocol team. Lower minimal sync amounts will make it easier to fill the protocol with many sync records and cause inconvenience to other users.

Recommendation(s): Consider monitoring the average deposit and withdrawal sizes as well as the total funds under the control of an average protocol user. The `minSyncAmount` might need to be readjusted depending on who interacts with the protocol, e.g., whales or small retail users.

Status: Acknowledged

Update from the client: Acknowledged. On the mainnet deployment, the `minSyncAmount` will be set to a higher value and we will monitor to prevent the gap between the `lastCompletedSyncIndex` and `lastPendingSyncIndex` from growing too large.

6.2.15 [Best Practices] Missing `address(0)` checks on setter functions

File(s): `src/layer2/L2SyncPool.sol`

Description: The setter functions in the `L2SyncPool` contract are missing input validation on the new values being set. Since the direct dependency on LayerZero contracts has been removed it is now easier to add these checks without the overhead stemming from the additional function overrides. One example is the missing `address(0)` check in the `_setRateLimiter(...)` function, which was previously performed in the `LiquidStakingToken` contract.

Recommendation(s): Consider adding input validation checks on the setter functions in the `L2SyncPool` contract.

Status: Fixed

Update from the client: Fixed: d8d8dd9 and 49e6ae6

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Dinero Pirex documentation

The documentation for the Dinero Pirex protocol is contained in the project's GitHub readme file. It provides a high-level overview of the project and contains helpful diagrams visualising the contract flows.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project.

8 Test Suite Evaluation

```
src/scripts/forgeTest.sh
[] Compiling...
[] Compiling 1 files with Solc 0.8.23
[] Compiling 270 files with Solc 0.8.25
[] Solc 0.8.23 finished in 24.32ms
[] Solc 0.8.25 finished in 30.04s
Compiler run successful with warnings:
Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:292:9:
    |
292 |         address _institutionalPirexEth,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:40:5:
   |
40 |     InstitutionalPirexEth _institutionalPirexEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:294:9:
    |
294 |         address _institutionalApxEth,
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:38:5:
   |
38 |     AutoPxEth _institutionalApxEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2519): This declaration shadows an existing declaration.
   --> src/scripts/InstitutionalDeploy.s.sol:359:9:
    |
359 |         address _institutionalApxEth
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> src/scripts/InstitutionalDeploy.s.sol:38:5:
   |
38 |     AutoPxEth _institutionalApxEth;
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (2018): Function state mutability can be restricted to view
   --> test/AutoPxEth.t.sol:288:5:
    |
288 |     function testAssetsPerShare() external {
    |     ^ (Relevant source part starts here and spans across multiple lines).

Warning (2018): Function state mutability can be restricted to view
  --> test/UpxEth.t.sol:57:5:
   |
57 |     function testSupportInterface() external {
   |     ^ (Relevant source part starts here and spans across multiple lines).


Ran 3 tests for test/UpxEth.t.sol:UpxEthTest
[PASS] testBurnBatch() (gas: 90406)
[PASS] testMintBatch() (gas: 73659)
[PASS] testSupportInterface() (gas: 9356)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 7.85s (198.10ms CPU time)

Ran 9 tests for test/RewardRecipient.t.sol:RewardRecipientTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 11604)
[PASS] testCannotHarvestUnauthorised() (gas: 11974)
[PASS] testCannotSetContractUnauthorized() (gas: 15575)
[PASS] testCannotSetContractUnrecognised() (gas: 10550)
[PASS] testCannotSetContractZeroAddress() (gas: 8979)
[PASS] testCannotSlashValidatorNoEthAllowed() (gas: 17169)
[PASS] testCannotSlashValidatorUnauthorised() (gas: 13288)
[PASS] testReceiveEther() (gas: 13217)
[PASS] testSetContract() (gas: 23737)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 7.86s (212.87ms CPU time)
```

```
Ran 7 tests for test/OracleAdapter.t.sol:OracleAdapterTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 12437)
[PASS] testCannotSetContractUnauthorized() (gas: 15654)
[PASS] testCannotSetContractUnrecognised() (gas: 10456)
[PASS] testCannotSetContractZeroAddress() (gas: 8863)
[PASS] testRequestVoluntaryExitUnauthorised() (gas: 11492)
[PASS] testSetContract() (gas: 23811)
[PASS] testVountaryExit(bytes) (runs: 100, : 20862, ~: 20558)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 7.87s (225.24ms CPU time)

Ran 9 tests for test/PxEth.t.sol:PxEthTest
[PASS] testBurn(uint224) (runs: 100, : 93854, ~: 93862)
[PASS] testCannotBurnNoBurnerRole() (gas: 12018)
[PASS] testCannotBurnZeroAddress() (gas: 37092)
[PASS] testCannotBurnZeroAmount() (gas: 37029)
[PASS] testCannotMintNoMinterRole() (gas: 12029)
[PASS] testCannotMintZeroAddress() (gas: 36949)
[PASS] testCannotMintZeroAmount() (gas: 37084)
[PASS] testCannotOperatorApproveZeroAddress() (gas: 38918)
[PASS] testMint(uint224) (runs: 100, : 90662, ~: 90662)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 7.89s (244.93ms CPU time)

Ran 4 tests for test/PirexFees.t.sol:PirexFeesTest
[PASS] testCannotSetRecipientNotAuthorized() (gas: 11609)
[PASS] testCannotSetRecipientZeroAddress() (gas: 8473)
[PASS] testDistributeFees(uint96) (runs: 100, : 169707, ~: 169707)
[PASS] testSetRecipient() (gas: 19123)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 35.72ms (26.32ms CPU time)

Ran 22 tests for test/AutoPxEth.t.sol:AutoPxEthTest
[PASS] testAssetsPerShare() (gas: 30621)
[PASS] testBeforeWithdrawAssetsGreaterThanTotalStaked() (gas: 319558)
[PASS] testCannotSetPirexEthUnauthorised() (gas: 34310)
[PASS] testCannotSetPirexEthZeroAddress() (gas: 13666)
[PASS] testCannotSetPlatformFeeUnauthorised() (gas: 34254)
[PASS] testCannotSetPlatformUnauthorised() (gas: 34318)
[PASS] testCannotSetPlatformZeroAddress() (gas: 13709)
[PASS] testCannotSetSetPlatformFeeGreaterThanMaxPlatformFee() (gas: 13539)
[PASS] testCannotSetWithdrawPenaltyGreaterThanMaxWithdrawalPenalty() (gas: 13604)
[PASS] testCannotSetWithdrawalPenaltyUnauthorised() (gas: 34340)
[PASS] testDeposit(uint96) (runs: 100, : 179660, ~: 179660)
[PASS] testDepositWithPendingRewards(uint96,uint96,uint32) (runs: 100, : 249188, ~: 249188)
[PASS] testPreviewWithdraw() (gas: 140032)
[PASS] testRevertNotifyRewardAmountUnathorised() (gas: 32692)
[PASS] testRevertNotifyRewardAmountnoRewards() (gas: 46616)
[PASS] testSetPirexEth() (gas: 19840)
[PASS] testSetPlatform() (gas: 22722)
[PASS] testSetPlatformFee() (gas: 22295)
[PASS] testSetWithdrawalPenalty() (gas: 22413)
[PASS] testTransfer() (gas: 155379)
[PASS] testTransferFrom() (gas: 177117)
[PASS] testVaultHarvest(uint96,uint32) (runs: 100, : 251056, ~: 251007)
Suite result: ok. 22 passed; 0 failed; 0 skipped; finished in 8.24s (479.42ms CPU time)

Ran 8 tests for test/RewardRecipientGateway.t.sol:RewardRecipientGatewayTest
[PASS] testCannotDissolveValidatorUnauthorised() (gas: 17653)
[PASS] testCannotHarvestUnauthorised() (gas: 17038)
[PASS] testCannotSlashValidatorNoEthAllowed() (gas: 22256)
[PASS] testCannotSlashValidatorUnauthorised() (gas: 18429)
[PASS] testDissolveValidator(uint96,bool) (runs: 100, : 7176504, ~: 6577563)
[PASS] testHarvest(uint96,uint96,uint96,uint8,uint96) (runs: 100, : 819212, ~: 930762)
[PASS] testReceiveEther() (gas: 18113)
[PASS] testSlashValidator(uint96,uint96,bool,bool,bool) (runs: 100, : 10015092, ~: 10483573)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 45.37s (58.00s CPU time)
```

```
Ran 95 tests for test/PirexEth.t.sol:PirexEthTest
[PASS] testBulkRedeemWithUpxEth(uint96,uint32,bool) (runs: 100, : 6841942, ~: 6152916)
[PASS] testCannotBulkRedeemWithUpxEthNotDissolved() (gas: 20079)
[PASS] testCannotBulkRedeemWithUpxEthPaused() (gas: 17411)
[PASS] testCannotBulkRedeemWithUpxEthZeroAddress() (gas: 15397)
[PASS] testCannotBulkRedeemWithUpxEthZeroAmount() (gas: 15296)
[PASS] testCannotDepositIncorrectValidatorParam() (gas: 330849)
[PASS] testCannotDepositPaused() (gas: 22952)
[PASS] testCannotDepositPrivilegedDepositEtherPaused() (gas: 22994)
[PASS] testCannotDepositPrivilegedUnauthorized() (gas: 20736)
[PASS] testCannotDepositReusePubKeyForInitializedValidator() (gas: 779147)
[PASS] testCannotDepositZeroAddress() (gas: 22515)
[PASS] testCannotDepositZeroAmount() (gas: 13950)
[PASS] testCannotDissolveValidatorInvalidAmount() (gas: 241628)
[PASS] testCannotDissolveValidatorNotWithdrawable() (gas: 243860)
[PASS] testCannotDissolveValidatorUnathorized() (gas: 220870)
[PASS] testCannotEmergencyWithdrawDepositEtherNotPaused() (gas: 11351)
[PASS] testCannotEmergencyWithdrawInvalidToken() (gas: 24717)
[PASS] testCannotEmergencyWithdrawNotPaused() (gas: 17152)
[PASS] testCannotEmergencyWithdrawUnauthorised() (gas: 16226)
[PASS] testCannotEmergencyWithdrawZeroAddress() (gas: 22544)
[PASS] testCannotEmergencyWithdrawZeroAmount() (gas: 22543)
[PASS] testCannotHarvestUnauthorized() (gas: 8583)
[PASS] testCannotInitiateRedemptionNotEnoughValidators() (gas: 8990429)
[PASS] testCannotInititateRedemptionNoPartialInitiateRedemption() (gas: 9219220)
[PASS] testCannotInititateRedemptionNoValidatorExit() (gas: 8989045)
[PASS] testCannotInititateRedemptionPaused() (gas: 16269)
[PASS] testCannotInititateRedemptionZeroAddress() (gas: 14051)
[PASS] testCannotInititateRedemptionZeroAmount() (gas: 13993)
[PASS] testCannotRedeemWithPxEthBuffer() (gas: 132335)
[PASS] testCannotRedeemWithPxEthPaused() (gas: 16086)
[PASS] testCannotRedeemWithPxEthTransferFailure() (gas: 188266)
[PASS] testCannotRedeemWithPxEthZeroAddress() (gas: 13825)
[PASS] testCannotRedeemWithPxEthZeroAmount() (gas: 13832)
[PASS] testCannotRedeemWithUpxEthEthTransferFailed() (gas: 5825231)
[PASS] testCannotRedeemWithUpxEthNotDissolved() (gas: 18621)
[PASS] testCannotRedeemWithUpxEthNotEnoughETH() (gas: 5851544)
[PASS] testCannotRedeemWithUpxEthPaused() (gas: 16165)
[PASS] testCannotRedeemWithUpxEthZeroAddress() (gas: 13919)
[PASS] testCannotRedeemWithUpxEthZeroAmount() (gas: 13904)
[PASS] testCannotSetContractUnauthorized() (gas: 41730)
[PASS] testCannotSetContractUnrecognised() (gas: 10865)
[PASS] testCannotSetContractZeroAddress() (gas: 9167)
[PASS] testCannotSetFeeGreaterThanDenominator(uint8) (runs: 100, : 12088, ~: 12088)
[PASS] testCannotSetFeeInvalidFee(uint8) (runs: 100, : 13212, ~: 13212)
[PASS] testCannotSetFeeInvalidMaxFee(uint8) (runs: 100, : 38675, ~: 38675)
[PASS] testCannotSetFeeUnauthorized() (gas: 16035)
[PASS] testCannotSetMaxBufferPctExceedsMax() (gas: 11026)
[PASS] testCannotSetMaxBufferPctUnauthorized() (gas: 15842)
[PASS] testCannotSetMaxFeeUnauthorized(uint8) (runs: 100, : 16708, ~: 16708)
[PASS] testCannotSetMaxProcessedValidatorCountInvalidValue() (gas: 8860)
[PASS] testCannotSetMaxProcessedValidatorCountUnauthorized() (gas: 15823)
[PASS] testCannotSlashMismatchValidator(uint96,uint96,bool,bool) (runs: 100, : 9093053, ~: 9093055)
[PASS] testCannotSlashValidatorAccountNotApproved(uint96,uint96,bool) (runs: 100, : 9136655, ~: 9136657)
[PASS] testCannotSlashValidatorInvalidAmount(uint96,uint96,bool) (runs: 100, : 9100086, ~: 9100086)
[PASS] testCannotSlashValidatorNotEnoughBuffer(uint96,uint96,bool) (runs: 100, : 9138620, ~: 9138621)
[PASS] testCannotSlashValidatorStatusNotWithdrawableOrStaking(uint96,uint96,bool) (runs: 100, : 5605588, ~: 5605588)
[PASS] testCannotSlashValidatorSumMismatch(uint96,uint96,bool) (runs: 100, : 9138647, ~: 9138649)
[PASS] testCannotSlashValidatorUnathorized() (gas: 17103)
[PASS] testCannotToggleBurnerAccountsUnauthorised() (gas: 18747)
[PASS] testCannotToggleDepositEtherUnathorized() (gas: 15742)
[PASS] testCannotTogglePauseStateUnauthorized() (gas: 15745)
[PASS] testCannotTopUpStakeAccountNotApproved(uint8,uint96) (runs: 100, : 8319594, ~: 8411438)
[PASS] testCannotTopUpStakeInvalidParams() (gas: 5957128)
[PASS] testCannotTopUpStakeNoEth(uint8,uint96) (runs: 100, : 8220364, ~: 8384511)
[PASS] testCannotTopUpStakeNoEthAllowed(uint8,uint96) (runs: 100, : 8228906, ~: 8196388)
[PASS] testCannotTopUpStakeNotEnoughBuffer(uint8,uint96) (runs: 100, : 8274923, ~: 8410365)
```

```
[PASS] testCannotTopUpStakeSumMismatch(uint8,uint8,uint96) (runs: 100, : 8270583, ~: 8436971)
[PASS] testCannotTopUpStakeUnauthorized() (gas: 32802)
[PASS] testCannotTopUpStakeValidatorNotStaking() (gas: 28186)
[PASS] testDeposit(uint96,uint32,bool,uint32) (runs: 100, : 327562, ~: 327797)
[PASS] testDepositPaused() (gas: 18237)
[PASS] testDepositPrivileged(uint96) (runs: 100, : 5703532, ~: 5703532)
[PASS] testDepositWithBufferNoFeeNoCompound(uint96,uint96) (runs: 100, : 5766718, ~: 5767137)
[PASS] testDissolveValidator(uint96) (runs: 100, : 6728921, ~: 5977435)
[PASS] testEmergencyWithdraw() (gas: 1550046)
[PASS] testHarvest(uint96,uint96,uint96,uint96) (runs: 100, : 7089089, ~: 6001283)
[PASS] testInitiateRedemptionByApxEthTransfer(uint96,uint32) (runs: 100, : 7204556, ~: 6193108)
[PASS] testInitiateRedemptionByApxEthTransferFrom(uint96,uint32) (runs: 100, : 7325425, ~: 6532336)
[PASS] testInitiateRedemptionByPxEth(uint96,uint32,bool) (runs: 100, : 6808836, ~: 6057140)
[PASS] testInstantRedeemWithPxEth(uint96,uint8,uint32,uint32) (runs: 100, : 276920, ~: 287829)
[PASS] testRedeemWithUpxEth(uint96,uint32,bool) (runs: 100, : 7031432, ~: 6027639)
[PASS] testRevertEmergencyWithdrawEthTransferFailure() (gas: 42779)
[PASS] testSetContract() (gas: 79067)
[PASS] testSetFee(uint8,uint32) (runs: 100, : 36155, ~: 38941)
[PASS] testSetMaxBufferSizePct(uint256) (runs: 100, : 36349, ~: 36946)
[PASS] testSetMaxFee(uint8,uint32) (runs: 100, : 20073, ~: 20697)
[PASS] testSetMaxProcessedValidatorCount(uint8) (runs: 100, : 17693, ~: 17693)
[PASS] testSlashValidator(uint96,uint96,bool,bool) (runs: 100, : 9220802, ~: 9220288)
[PASS] testSlashValidatorWhenExiting(uint96,bool,bool) (runs: 100, : 9218773, ~: 9208564)
[PASS] testToggleBurnerAccounts() (gas: 32452)
[PASS] testTogglePauseDepositEther() (gas: 19437)
[PASS] testTogglePauseState() (gas: 18195)
[PASS] testTopUpStake(uint8,uint96,bool) (runs: 100, : 8290380, ~: 8457092)
[PASS] testValidatorSpinup() (gas: 635087)
[PASS] testValidatorSpinupWithLimit(uint8) (runs: 100, : 6526575, ~: 6105538)
Suite result: ok. 95 passed; 0 failed; 0 skipped; finished in 130.46s (276.94s CPU time)

Ran 29 tests for test/ValidatorQueue.t.sol:ValidatorQueueTest
[PASS] testAddInitializedValidators() (gas: 4945431)
[PASS] testCannotAddInitializedValidatorsDepositEtherUnpaused() (gas: 21490)
[PASS] testCannotAddInitializedValidatorsUnauthorized() (gas: 35868)
[PASS] testCannotAddUsedInitializedValidators() (gas: 727692)
[PASS] testCannotClearInitializedValidatorsDepositEtherUnpaused() (gas: 8488)
[PASS] testCannotClearInitializedValidatorsUnauthorized() (gas: 22927)
[PASS] testCannotGetNextValidatorQueueEmpty() (gas: 20964)
[PASS] testCannotPopInitializedValidatorOutOfBounds(uint256) (runs: 100, : 5294945, ~: 5410176)
[PASS] testCannotPopInitializedValidatorsDepositEtherUnpaused() (gas: 8493)
[PASS] testCannotPopInitializedValidatorsUnauthorized() (gas: 22994)
[PASS] testCannotPopValidatorQueueEmpty() (gas: 11501)
[PASS] testCannotRemoveInitializedMismatchValidators(uint256,bool) (runs: 100, : 5567717, ~: 5567720)
[PASS] testCannotRemoveInitializedValidatorsDepositEtherUnpaused() (gas: 9359)
[PASS] testCannotRemoveInitializedValidatorsUnauthorized() (gas: 23760)
[PASS] testCannotRemoveOrderedOutOfBounds() (gas: 10961)
[PASS] testCannotRemoveUnorderedOutOfBounds() (gas: 11102)
[PASS] testCannotSwapInitializedValidatorsDepositingEtherUnpaused() (gas: 8632)
[PASS] testCannotSwapInitializedValidatorsInvalidIndex() (gas: 19353)
[PASS] testCannotSwapInitializedValidatorsUnauthorized() (gas: 23111)
[PASS] testCannotSwapOutOfBounds() (gas: 455113)
[PASS] testCannotSwapValidatorQueueEmpty() (gas: 8600)
[PASS] testClearInitializedValidator() (gas: 5557918)
[PASS] testGetInitializedValidatorAt(uint8) (runs: 100, : 5582273, ~: 5582273)
[PASS] testGetInitializedValidatorCount() (gas: 5566973)
[PASS] testGetStakingValidatorAt(uint8) (runs: 100, : 9083564, ~: 9083564)
[PASS] testPopInitializedValidator(uint256) (runs: 100, : 5672812, ~: 5750968)
[PASS] testRemoveInitializedValidator(uint256,bool) (runs: 100, : 5479114, ~: 5469919)
[PASS] testStakingValidatorCount(uint8) (runs: 100, : 6654651, ~: 6311072)
[PASS] testSwapInitializedValidator(uint256,uint256) (runs: 100, : 5629249, ~: 5629249)
Suite result: ok. 29 passed; 0 failed; 0 skipped; finished in 235.66s (264.22s CPU time)

Ran 101 tests for test/InstitutionalPirexEth.t.sol:InstitutionalPirexEthTest
[PASS] testBulkRedeemWithIUpxEth(uint96,uint32,bool) (runs: 100, : 7625778, ~: 6812624)
[PASS] testCannotBulkRedeemWithIUpxEthEmptyArray() (gas: 29052)
[PASS] testCannotBulkRedeemWithIUpxEthMismatchArrayLength() (gas: 29050)
[PASS] testCannotBulkRedeemWithIUpxEthNotDissolved() (gas: 35427)
[PASS] testCannotBulkRedeemWithIUpxEthPaused() (gas: 25923)
[PASS] testCannotBulkRedeemWithIUpxEthZeroAddress() (gas: 30693)
[PASS] testCannotBulkRedeemWithIUpxEthZeroAmount() (gas: 30614)
[PASS] testCannotDepositIncorrectValidatorParam() (gas: 653792)
[PASS] testCannotDepositPaused() (gas: 31186)
```

```
[PASS] testCannotDepositPrivilegedDepositEtherPaused() (gas: 38225)
[PASS] testCannotDepositPrivilegedUnauthorized() (gas: 26370)
[PASS] testCannotDepositReusePubKeyForInitializedValidator() (gas: 1289192)
[PASS] testCannotDepositUnauthorised() (gas: 24347)
[PASS] testCannotDepositZeroAddress() (gas: 36330)
[PASS] testCannotDepositZeroAmount() (gas: 27786)
[PASS] testCannotDissolveValidatorInvalidAmount() (gas: 263570)
[PASS] testCannotDissolveValidatorNotWithdrawable() (gas: 266009)
[PASS] testCannotDissolveValidatorUnathorized() (gas: 225544)
[PASS] testCannotEmergencyWithdrawDepositEtherNotPaused() (gas: 16781)
[PASS] testCannotEmergencyWithdrawInvalidToken() (gas: 41939)
[PASS] testCannotEmergencyWithdrawNotPaused() (gas: 32866)
[PASS] testCannotEmergencyWithdrawUnauthorised() (gas: 22223)
[PASS] testCannotEmergencyWithdrawZeroAddress() (gas: 41881)
[PASS] testCannotEmergencyWithdrawZeroAmount() (gas: 41925)
[PASS] testCannotHarvestUnauthorized() (gas: 13526)
[PASS] testCannotInitiateRedemptionNotEnoughValidators() (gas: 10472520)
[PASS] testCannotInititateRedemptionNoPartialInitiateRedemption() (gas: 12280264)
[PASS] testCannotInititateRedemptionNoValidatorExit() (gas: 6141069)
[PASS] testCannotInititateRedemptionPaused() (gas: 24740)
[PASS] testCannotInititateRedemptionZeroAddress() (gas: 26162)
[PASS] testCannotInititateRedemptionZeroAmount() (gas: 26126)
[PASS] testCannotRedeemWithIPxEthBuffer() (gas: 489848)
[PASS] testCannotRedeemWithIPxEthPaused() (gas: 24664)
[PASS] testCannotRedeemWithIPxEthTransferFailure() (gas: 390513)
[PASS] testCannotRedeemWithIPxEthZeroAddress() (gas: 25501)
[PASS] testCannotRedeemWithIPxEthZeroAmount() (gas: 25464)
[PASS] testCannotRedeemWithIUpxEthEthTransferFailed() (gas: 6269129)
[PASS] testCannotRedeemWithIUpxEthNotDissolved() (gas: 30308)
[PASS] testCannotRedeemWithIUpxEthNotEnoughETH() (gas: 6253971)
[PASS] testCannotRedeemWithIUpxEthPaused() (gas: 24718)
[PASS] testCannotRedeemWithIUpxEthZeroAddress() (gas: 25573)
[PASS] testCannotRedeemWithIUpxEthZeroAmount() (gas: 25580)
[PASS] testCannotRemoveInitializedMismatchValidators(uint256,bool) (runs: 100, : 5688883, ~: 5688886)
[PASS] testCannotSetContractUnauthorized() (gas: 47929)
[PASS] testCannotSetContractUnrecognised() (gas: 22705)
[PASS] testCannotSetContractZeroAddress() (gas: 20962)
[PASS] testCannotSetFeeGreaterThanDenominator(uint8) (runs: 100, : 23776, ~: 23776)
[PASS] testCannotSetFeeInvalidFee(uint8) (runs: 100, : 25340, ~: 25340)
[PASS] testCannotSetFeeInvalidMaxFee(uint8) (runs: 100, : 53073, ~: 53073)
[PASS] testCannotSetFeeUnauthorized() (gas: 21611)
[PASS] testCannotSetMaxBufferPctExceedsMax() (gas: 22357)
[PASS] testCannotSetMaxBufferPctUnauthorized() (gas: 21504)
[PASS] testCannotSetMaxFeeUnauthorized(uint8) (runs: 100, : 22152, ~: 22152)
[PASS] testCannotSetMaxProcessedValidatorCountInvalidValue() (gas: 20063)
[PASS] testCannotSetMaxProcessedValidatorCountUnauthorized() (gas: 21375)
[PASS] testCannotSlashMismatchValidator(uint96,uint96,bool,bool) (runs: 100, : 10352392, ~: 10352393)
[PASS] testCannotSlashValidatorAccountNotApproved(uint96,uint96,bool) (runs: 100, : 10398633, ~: 10398632)
[PASS] testCannotSlashValidatorInvalidAmount(uint96,uint96,bool) (runs: 100, : 10359782, ~: 10359781)
[PASS] testCannotSlashValidatorNotEnoughBuffer(uint96,uint96,bool) (runs: 100, : 10401917, ~: 10401917)
[PASS] testCannotSlashValidatorStatusNotWithdrawableOrStaking(uint96,uint96,bool) (runs: 100, : 5722881, ~: 5722881)
[PASS] testCannotSlashValidatorSumMismatch(uint96,uint96,bool) (runs: 100, : 10401893, ~: 10401892)
[PASS] testCannotSlashValidatorUnathorized() (gas: 22743)
[PASS] testCannotToggleBurnerAccountsUnauthorised() (gas: 24312)
[PASS] testCannotToggleDepositEtherUnathorized() (gas: 21247)
[PASS] testCannotTogglePauseStateUnauthorized() (gas: 21294)
[PASS] testCannotTopUpStakeAccountNotApproved(uint8,uint96) (runs: 100, : 9487587, ~: 9682482)
[PASS] testCannotTopUpStakeInvalidParams() (gas: 6163480)
[PASS] testCannotTopUpStakeNoEth(uint8,uint96) (runs: 100, : 9308285, ~: 9443070)
[PASS] testCannotTopUpStakeNoEthAllowed(uint8,uint96) (runs: 100, : 9651961, ~: 9964400)
[PASS] testCannotTopUpStakeNotEnoughBuffer(uint8,uint96) (runs: 100, : 9551647, ~: 9686189)
[PASS] testCannotTopUpStakeSumMismatch(uint8,uint8,uint96) (runs: 100, : 9542754, ~: 9705056)
[PASS] testCannotTopUpStakeUnauthorized() (gas: 38470)
[PASS] testCannotTopUpStakeValidatorNotStaking() (gas: 42478)
[PASS] testClearInitializedValidator() (gas: 5681001)
[PASS] testDeposit(uint96,uint32) (runs: 100, : 450170, ~: 456463)
[PASS] testDepositPaused() (gas: 27373)
[PASS] testDepositPrivileged(uint96) (runs: 100, : 6009033, ~: 6002361)
[PASS] testDepositWithBufferNoFee(uint96,uint96) (runs: 100, : 6064742, ~: 6065150)
[PASS] testDissolveValidator(uint96) (runs: 100, : 7317305, ~: 6663867)
[PASS] testEmergencyWithdraw(uint96) (runs: 100, : 2078745, ~: 2078745)
```

```
[PASS] testHarvest(uint96,uint96,uint96,uint96) (runs: 100, : 7582065, ~: 6229576)
[PASS] testInitiateRedemptionByIPxEth(uint96,uint32,bool) (runs: 100, : 7832231, ~: 7267402)
[PASS] testInstantRedeemWithIPxEth(uint96,uint8,uint32,uint32) (runs: 100, : 622782, ~: 630732)
[PASS] testPopInitializedValidator(uint256) (runs: 100, : 5720394, ~: 5700906)
[PASS] testRedeemWithIUpxEth(uint96,uint32,bool) (runs: 100, : 7750541, ~: 6812611)
[PASS] testRemoveInitializedValidator(uint256,bool) (runs: 100, : 5601449, ~: 5570746)
[PASS] testRevertEmergencyWithdrawEthTransferFailure() (gas: 72839)
[PASS] testSetContract() (gas: 184943)
[PASS] testSetFee(uint8,uint32) (runs: 100, : 48830, ~: 51019)
[PASS] testSetMaxBufferSizePct(uint256) (runs: 100, : 93352, ~: 93750)
[PASS] testSetMaxFee(uint8,uint32) (runs: 100, : 33817, ~: 34441)
[PASS] testSetMaxProcessedValidatorCount(uint8) (runs: 100, : 58856, ~: 58856)
[PASS] testSlashValidator(uint96,uint96,bool,bool) (runs: 100, : 10609034, ~: 10682200)
[PASS] testSlashValidatorWhenExiting(uint96,bool,bool) (runs: 100, : 10830997, ~: 10916587)
[PASS] testSwapInitializedValidator(uint256,uint256) (runs: 100, : 5753581, ~: 5753581)
[PASS] testToggleBurnerAccounts() (gas: 42018)
[PASS] testTogglePauseDepositEther() (gas: 64229)
[PASS] testTogglePauseState() (gas: 27353)
[PASS] testTopUpStake(uint8,uint96,bool) (runs: 100, : 9513719, ~: 9664833)
[PASS] testValidatorSpinup() (gas: 923822)
[PASS] testValidatorSpinupWithLimit(uint8) (runs: 100, : 7238368, ~: 6696930)
Suite result: ok. 101 passed; 0 failed; 0 skipped; finished in 247.47s (513.12s CPU time)

Ran 10 test suites in 248.52s (698.71s CPU time): 287 tests passed, 0 failed, 0 skipped (287 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.