Security Review Report
NM-0117 CLAYSTACK ETH STAKING

NETHERMIND SECURITY

(Sep 4, 2023)

Contents

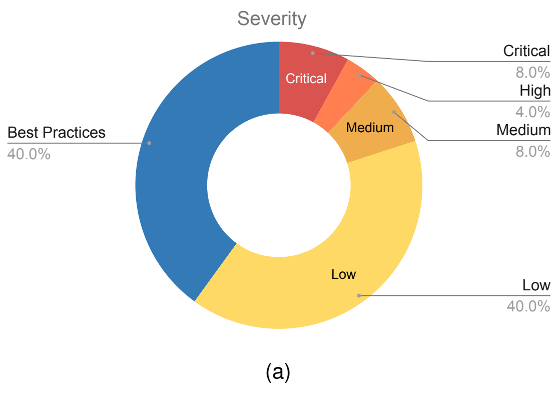
1 Executive Summary

This document presents the security review conducted by Nethermind for the ClayStack ETH Staking Contracts. `ClayStack` is a platform that enables ETH staking on the Ethereum blockchain. It allows users to participate in the staking process by delegating their stake to validators on the Ethereum Beacon chain. This staking process involves users locking up their ETH in exchange for a token called csETH. This csETH token represents their staked amount and is minted on the Ethereum network. The validator pool on `ClayStack` is managed by the platform, and those who operate and manage the validators are rewarded for their efforts. These operators can either be direct nodes or operators using a DVT based approach. In the audited version of `ClayStack`, validators are trusted/controlled entities that are not likely to be malicious.

When users want to unstake their ETH, they burn the csETH tokens they received during the staking process. Unstaking involves waiting for an unbonding period before the users can claim their unstaked ETH. This period is likely a security measure to ensure that the network remains stable even if a large number of users decide to unstake their funds simultaneously. To optimize gas costs and efficiency, `ClayStack` employs a batching process. This means that users' staking and unstaking actions are aggregated and processed in batches, aiming to reduce the overall gas fees associated with Ethereum transactions. Furthermore, the staking and unstaking process is netted, meaning that the deposits made by users who are staking can cover the withdrawal requests from users who are unstaking.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools and (c) simulation of the smart contract. **Along this document, we report** 25 points of attention, where 2 are classified as `Critical`, 1 is classified as `High`, 2 are classified as `Medium`, 10 are classified as `Low`, and 10 are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. 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.



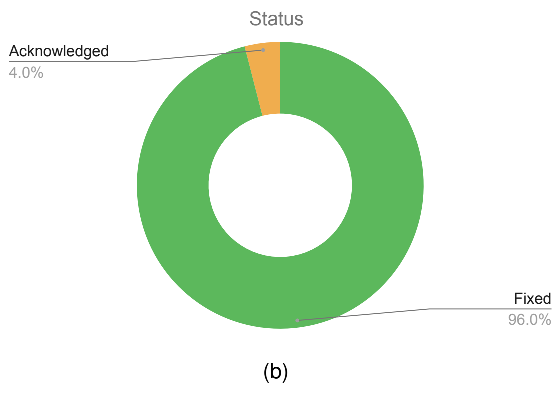


Fig. 1: Distribution of issues: Critical (2), **High** (1), **Medium** (2), **Low** (10), **Undetermined** (0), **Informational** (0), **Best Practices** (10).
Distribution of status: Fixed (24), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug. 24, 2023
Response from Client	Aug. 29, 2023
Final Report	Sep. 04, 2023
Methods	Manual Review, Automated Analysis
Repository	claystack-ethereum
Commit Hash (Audit)	cc2659a3e8782e5b1d6ed35c8488194fbba8901b
Documentation	README
Documentation Assessment	Medium
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	ClayMain.sol	537	162	30.2%	142	841
2	CsToken.sol	23	23	100.0%	7	53
3	NodeManager.sol	301	114	37.9%	83	498
4	RoleManager.sol	47	41	87.2%	12	100
5	TimeLock.sol	10	12	120.0%	3	25
6	deposits/DepositsManager.sol	176	70	39.8%	44	290
7	deposits/interfaces/IDepositsManager.sol	41	32	78.0%	7	80
8	deposits/interfaces/IrETH.sol	13	1	7.7%	10	24
9	deposits/interfaces/IStETH.sol	7	1	14.3%	4	12
10	deposits/interfaces/IStrategy.sol	20	11	55.0%	9	40
11	deposits/interfaces/IWithdrawalQueueERC721.sol	41	51	124.4%	9	101
12	deposits/interfaces/IWstETH.sol	10	38	380.0%	7	55
13	deposits/interfaces/IWToken.sol	6	2	33.3%	3	11
14	deposits/strategies/StrategyRETH.sol	80	33	41.2%	23	136
15	deposits/strategies/StrategyStETH.sol	128	52	40.6%	36	216
16	interfaces/IClayMain.sol	81	79	97.5%	33	193
17	interfaces/ICSToken.sol	8	6	75.0%	5	19
18	interfaces/IDepositContract.sol	12	15	125.0%	4	31
19	interfaces/INodeManager.sol	39	43	110.3%	13	95
20	interfaces/IRoleManager.sol	4	2	50.0%	1	7
	Total	**1584**	**788**	**49.7%**	**455**	**2827**

3 Summary of Issues

	Finding	Severity	Update
1	Breaking the invariant of ClayMain contract leads to DOS in all functions	Critical	Fixed
2	Function `claim(...)` in `StrategyStETH` has no access control, allowing attackers to steal all funds in the withdrawal requests	Critical	Fixed
3	A Denial Of Service attack could be carried out on the `oracleReport()` function	High	Fixed
4	Incorrect calculation of `amountNewRewards` from Oracles's reported data	Medium	Fixed
5	`_closeBatches()` always calls `_deleteBatchFromQueue()` with `index = 0` incorrectly	Medium	Fixed
6	Incorrect calculation of `totalPages` in getter functions	Low	Fixed
7	Skipping invariant check when `userFlow == 0` breaks its meaning	Low	Fixed
8	The `_closeBatches()` function returns a different output from the `fulfilledBatches`	Low	Fixed
9	Update of `claimablePool` Amount Prior to ETH Transfer	Low	Fixed
10	`ClayStack` protocol does not have any protection against deposit front-running	Low	Fixed
11	`getOrder` does not consider all possible Strategy statuses	Low	Fixed
12	`previousExits` updated with the total unfulfilled exits requests	Low	Fixed
13	`setClayMain()` function initializer is not restricted	Low	Acknowledged
14	`withdrawQueueNew` always resets to `0` when swapping internal and external unstake	Low	Fixed
15	`withdrawQueueNew` is not updated after closing a batch	Low	Fixed
16	Confusing parameter naming (`minClaimDays`)	Best Practices	Fixed
17	Consider making `withdrawQueueNew` a boolean instead of uint256	Best Practices	Fixed
18	Emit events on important updates	Best Practices	Fixed
19	Function `getValidators()` in the `NodeManager` could break earlier to save gas	Best Practices	Fixed
20	Lack of existence check for Strategy access	Best Practices	Fixed
21	Missing balance check in claim function	Best Practices	Fixed
22	The `getValidators()` function's documentation is incorrect	Best Practices	Fixed
23	Token strategy is not checked for existence before adding in the function `addStrategy(...)`	Best Practices	Fixed
24	Unused import of `PausableUpgradeable` contract	Best Practices	Fixed
25	`updateBalances()` function visibility can be reduced to external	Best Practices	Fixed

4 System Overview

The audit covers the following smart contracts and interfaces:

- `ClayMain.sol` & `interfaces/IClayMain.sol`
- `CsToken.sol` & `interfaces/ICSToken.sol`
- `NodeManager.sol` & `interfaces/INodeManager.sol`
- `RoleManager.sol` & `interfaces/IRoleManager.sol`
- `TimeLock.sol`
- `interfaces/IDepositContract.sol`
- `deposits/DepositsManager.sol` & `deposits/interfaces/IDepositsManager.sol`
- `deposits/strategies/StrategyRETH.sol` & `deposits/interfaces/IrETH.sol`
- `deposits/strategies/StrategyStETH.sol` & `deposits/interfaces/IStETH.sol`
- `deposits/interfaces/IStrategy.sol`
- `deposits/interfaces/IWithdrawQueueERC721.sol`
- `deposits/interfaces/IWstETH.sol`
- `deposits/interfaces/IWToken.sol`

The following schema (Fig. 2) shows a high-level overview of the relationship between contracts in the `ClayStack` protocol:

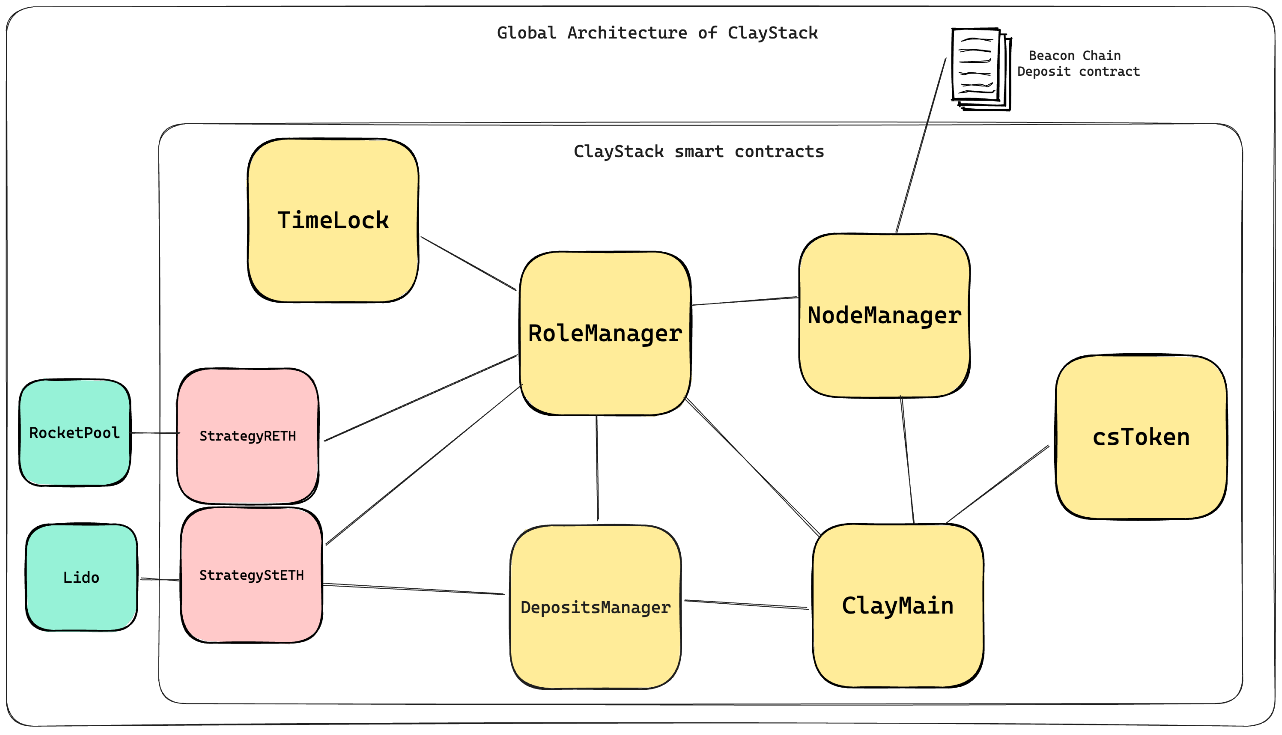


Fig. 2: General Architecture

4.1 `ClayMain.sol`

The `ClayMain` contract serves as the main entry point for users to interact with the `ClayStack` protocol. It provides users with the following features:

- Deposit ETH tokens to receive csETH tokens in return;
- Start a withdrawal process;
- Claim ETH from a withdrawal request;
- Instantly withdraw ETH.

These features are based on other external functions, such as:

- Closing batches;
- Getting exchange values from csToken to ETH and vice versa;
- Auto-balancing, which performs claiming of rewards and staking of tokens for the protocol.

To begin stacking ETH with `ClayStack`, the user must first call the `deposit()` function. In exchange, the user will receive csETH tokens. When the protocol is deployed, one ETH will be exchanged for one csETH. Over time, as rewards are gained through validators, the value of csETH tokens will be compared to that of ETH. The exchange rate between ETH and csETH can be retrieved by calling the `getExchangeRate()` function.

When users want to withdraw, they will need to deposit csETH to the contract and create a claim order that will track the amount and time to claim. During this period, penalties are still applied to users, and only after the unbonding process, users are able to claim their ETH.

4.2 `CsToken.sol`

The `CsToken` contract is responsible for tracking user shares. It inherites from OpenZeppelin's ERC20 and implements custom minting and burning functions. Both functions are only callable from the `ClayMain` contract.

4.3 `NodeManager.sol`

The `NodeManager` component is in charge of managing the deposit and withdrawal from `ClayMain` contract. Deposits are done to the Beacon Chain Deposit Contract. This contract is in charge of managing the registration and the exit of validators. It is also in charge of integrating the data from Oracle reports into the `ClayStack` protocol. The `NodeManager` component is heavily linked to the `ClayMain` contract.

Fig. 3 illustrates how a new validator registration happens:

1. Autobalance function is called on the NodeManager contract
2. NodeManager calls the Autobalance function on the ClayMain contract
3. ClayMain sends necessary ETH to be deposited to the Beacon Chain
4. NodeManager deposits the ETH to the BeaconChain Deposit Contract
5. Validator is registered

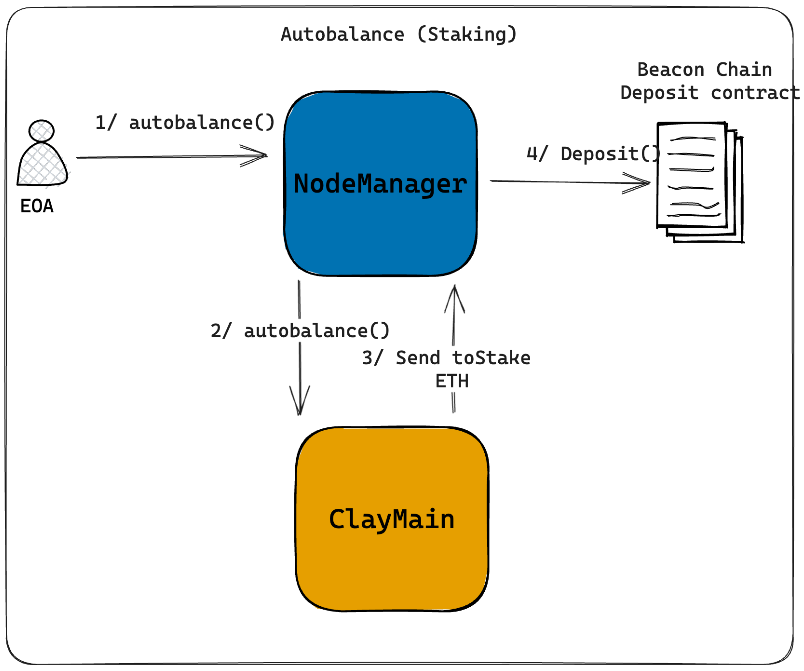


Fig. 3: Autobalance (Staking) Flow

4.4 RoleManager.sol

The `RoleManager` contract is responsible for managing access control within the system to prevent unauthorized access to critical functions. It encompasses various roles, such as:

- `CS_SERVICE_ROLE` is the role that can change the settings of the contracts including pausing and unpausing.
- `TIMELOCK_ROLE` is the role that can grant and revoke roles.
- `TIMELOCK_UPGRADES_ROLE` is the role that can upgrade the contracts.

4.5 Timelock.sol

The `TimeLock` contract serves as protection for users against critical changes in the system settings. It inherits from the `TimelockController` contract of OpenZeppelin and initializes all important parameters in its constructor. Any critical changes in settings or contract upgrades in `ClayStack` protocol must go through this contract. It provides users with a reasonable period of time to review the action and respond to it.

4.6 DepositsManager.sol

The `DepositsManager` contract assists users in migrating from one Liquidity Staking Protocol to `ClayStack`. A strategy is defined for each Liquid Staking protocol, with two strategies currently implemented:

- Rocket Pool (`StrategyRETH`)
- Lido (`StrategyStETH`)

Each strategy exposes multiple functions, the main ones are `claim()` and `withdraw()`.

For the Rocket Pool protocol, the migration can be done in a single call to the `withdraw()` function, as the user only needs to burn `rETH` to retrieve `ETH`. The `DepositManager` then deposits the funds to `ClayMain`.

Fig. 4 shows the flow of how the migration is done for the RETH (Rocket Pool).

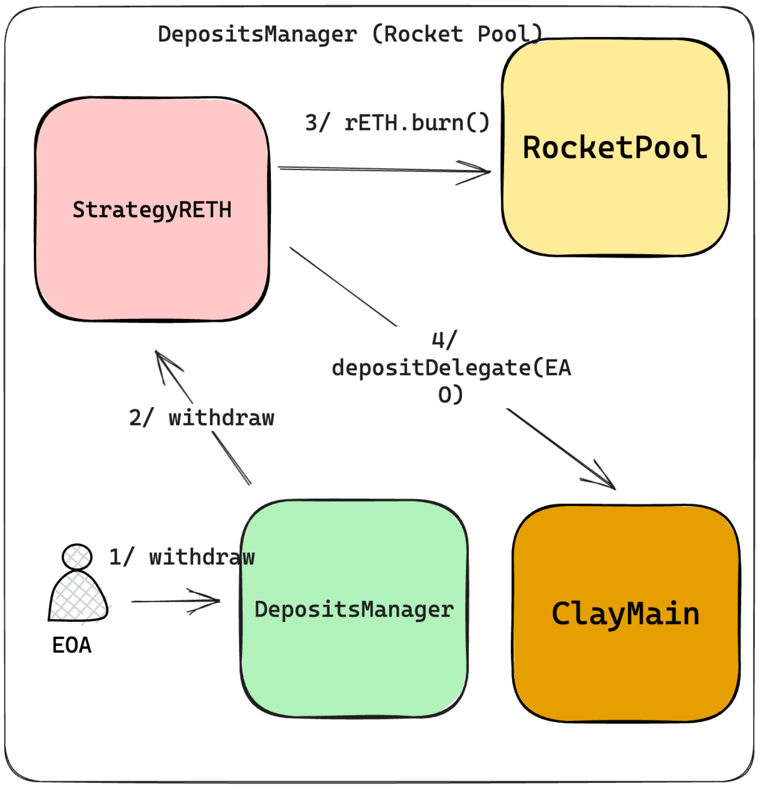


Fig. 4: RETH Strategy Flow

For the Lido protocol, the migration is done in two steps. Users must first start the withdrawal process by calling the `withdraw()` function and then wait for a period of time. When withdrawal is ready, users can call the function `claim()` to claim `ETH` and deposit it to `ClayMain`.

Fig. 5 shows the flow of how the migration is done for the StETH (Lido).

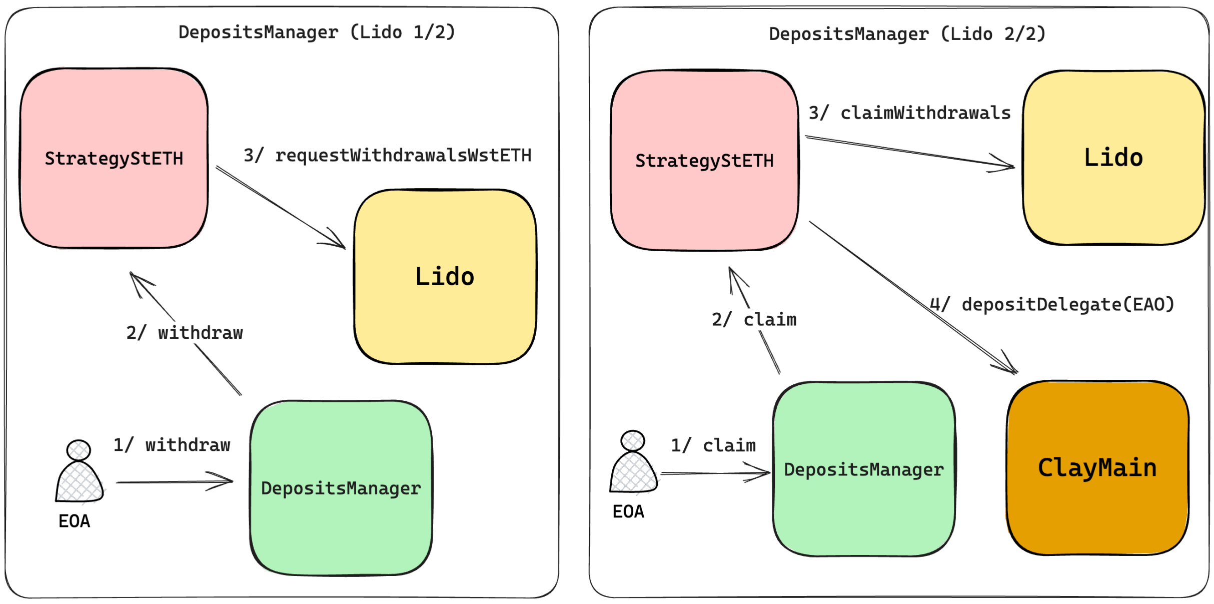


Fig. 5: StETH Strategy Flow

5 Risk Rating Methodology

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

Likelihood is a measure of 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 the finding were to be exploited by an attacker. 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 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 formally pass to the client;

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 [Critical] Breaking the invariant of ClayMain contract leads to DOS in all functions

File(s): ClayMain.sol

Description: The ClayMain contract contains an essential invariant that is checked after each interaction, within the _balanced(...) function. This invariant is designed to ensure that the total user flow equals the staking flow. However, the current implementation calculates the stakingFlow by directly using address(this).balance. This practice is risky, as it opens a vulnerability where an attacker can manipulate the balance by self-destructing other contracts and sending ETH to ClayMain. Since self-destruct does not execute any code, the receive() function won't be triggered either. The problematic section of the code is provided below.

```
// @dev Assures contract state remains balanced by comparing user flows deposits vs staking flows.
function _balanced() internal {
    uint256 currentDeposit = funds.currentDeposit;
    require(currentDeposit > 0, "Current deposits cannot go to zero");
    uint256 userFlow = (funds.currentDeposit + funds.withdrawQueue) / 1e16;
    if (userFlow != 0) {
        ////////////////////////////////////////////////////////////////////
        // @audit address(this).balance can be manipulated by self-destruct
        ////////////////////////////////////////////////////////////////////
        uint256 stakingFlow = (address(this).balance +
            funds.stakedDeposit +
            funds.unstakeInternal +
            funds.unstakeExternal) / 1e16;
        require(stakingFlow <= userFlow + 1 && userFlow <= stakingFlow + 1, "Unbalanced contract");
    }

    _exchangeRateProtection();
}
```

Recommendation(s): Consider maintaining a state variable to track the contract balance instead of relying on address(this).balance.

Status: Fixed

Update from the client: Fixed in commit 4dba840

6.2 [Critical] Function claim(...) in StrategyStETH has no access control, allowing attackers to steal all funds in the withdrawal requests

File(s): deposits/strategies/StrategyStETH.sol

Description: The claim(...) function in the StrategyStETH contract lacks access control, which allows attackers to call it with arbitrary _requestId and _owner, thereby stealing funds from withdrawal requests. This function should only be called by DepositsManager to complete the withdrawal process of stETH.

```
// @audit No access control, can be called with arbitrary _owner param
function claim(uint256 _requestID, address _owner, bytes memory _data) external nonReentrant returns (uint256) {
    require(_requestID != 0, "Invalid requestID");
    require(_owner != address(0), "Invalid owner");
    ...

    // Mint csETH
    uint256 claimedAmount = address(this).balance - balanceBefore;
    uint256 csEthMinted = clayMain.depositDelegate{value: claimedAmount}(_owner);
    return csEthMinted;
}
```

Recommendation(s): Consider adding access control to only allow DepositManager to call the claim(...) function.

Status: Fixed

Update from the client: Fixed in commit e54b6a25

6.3 [High] A Denial Of Service attack could be carried out on the `oracleReport()` function

File(s): `NodeManager.sol`

Description: The `oracleReport()` function is used to provide information from the consensus layer to the protocol. This information can be reported by anyone but must be signed by whitelisted oracles.

The function starts by checking that the provided `_reportBlock` parameter is strictly above the `oracleReportBlock` state variable. This check ensures that the provided information is newer than the last provided Oracle report.

```
function oracleReport(
    uint256 _reportBlock,
    uint256 _validatorsCount,
    uint256 _validatorsBalance,
    uint256 _validatorsExited,
    bool _withdrawalsDisabled,
    bytes[] calldata _signatures
) external whenNotPaused nonReentrant {
    //@audit _reportBlock parameter must be higher than oracleReportBlock
    require(_reportBlock > oracleReportBlock, "Invalid report block");
[...]
```

However, in the `NodeManager` contract, the `oracleReportBlock` state variable is updated to the latest `block.number` each time the `receive()` function is invoked.

```
receive() external payable {
    // next report must be after this to confirm a post-balance
    oracleReportBlock = block.number;
}
```

The `receive()` function is invoked when a transaction with empty `calldata` and `msg.value` (i.e with ether) is sent to the contract.

An attacker could take advantage of this feature to deny any update of data from the consensus layer. He would have to proceed this way:

– watch the mempool for any transaction calling the `oracleReport()` function;
– front-run this transaction, sending as low as `1 wei` to the `NodeManager` contract to update the `oracleReportBlock` state variable;
– the legitimate `oracleReport()` transaction reverts;

Recommendation(s): Consider updating the `oracleReportBlock` state variable exclusively when an Oracle report has been validated.

Status: Fixed

Update from the client: Fixed in commit b2a2fa8

6.4 [Medium] Incorrect calculation of `amountNewRewards` from Oracles's reported data

File(s): `NodeManager.sol`

Description: The `oracleReport()` function calculates the amount of new rewards since the last report. The calculated `amountNewRewards` is then used to send fees to the `nodeOperator`.

However, the `amountNewRewards` are exclusively calculated based on the liquid rewards provided to the contract. If there are no `amountLiquidRewards`, there won't be any accounting for the rewards to be sent to the `nodeOperator`.

```
uint256 amountRewards = _validatorsBalance > amountPrincipal ? _validatorsBalance - amountPrincipal : 0;
[...]
uint256 previousRewardsAccrued = rewardsAccrued;
[...]
rewardsAccrued = amountRewards;
[...]
uint256 amountLiquid = address(this).balance;
if (amountLiquid != 0) {
    uint256 amountExited = _min(amountLiquid, newExits * DEPOSIT_SIZE);
    uint256 amountLiquidRewards = amountLiquid - amountExited;
    uint256 totalRewards = amountRewards + amountLiquidRewards;
    uint256 amountNewRewards = totalRewards > previousRewardsAccrued ? totalRewards - amountRewards : 0;
```

There are cases where the rewards are partially accounted for, leading to wrong accounting of the `getBalance()` function from the `NodeManager` contract. This could lead to the wrong exchange rate between CS and ETH tokens.

Recommendation(s): Account for the accrued fees allocated to the `nodeOperators` regardless of the liquidity.

Status: Fixed.

Update from the client: There is indeed a missing accrued rewards on non-liquid rewards that must be deducted updated (e.g. on penalties). See replicated unit test with the provided example and an additional where there is both partial payment additional not accounted for. The report 3 suggested wasn't too clear if you can expand, we can replicate.

There are two cases, where new rewards are 100% or potentially partially already accounted for by a previous oracle report. Thus a comparison is missing. This slightly changes the logic.

After our second meeting, clarified the liquid rewards are paid as they come and `nodeOperatorAccruedFees` is used to ensure the exchange rate reflects any fees due.

Fixed 12ef32e.

6.5 [Medium] `_closeBatches()` always calls `_deleteBatchFromQueue()` with `index = 0` incorrectly

File(s): `NodeManager.sol`

Description: In the function `_closeBatches(...)`, after calculating the number of fulfilled batches, it loops through the queue and calls the function `_deleteBatchFromQueue(...)` with `index = 0` to delete each element from the queue. The function `_deleteBatchFromQueue(...)` swaps the deleting element with the last element and pops it from the queue. However, this rearranges the order of elements in the queue and makes calling `_deleteBatchFromQueue(index = 0)` incorrect, resulting in the wrong element being deleted.

```
function _closeBatches(uint256 _newExits) internal {
    ...

    // create output
    uint256[] memory batches = new uint256[](fulfilledBatches);
    for (uint256 i = 0; i < fulfilledBatches; i++) {
        batches[i] = batchExitQueue[0];
        ////////////////////////////////////////////
        // @audit Always delete index = 0 is incorrect
        ////////////////////////////////////////////
        _deleteBatchFromQueue(0);
    }
    ...
}

function _deleteBatchFromQueue(uint256 _indexToDelete) internal {
    require(_indexToDelete < batchExitQueue.length, "Invalid index");

    // Move the last element to the index to be deleted
    batchExitQueue[_indexToDelete] = batchExitQueue[batchExitQueue.length - 1];

    // Remove the last element
    batchExitQueue.pop();
}
```

Recommendation(s): Consider rewriting the process to ensure the order of the elements in the queue remains the same after deleting.

Status: Fixed

Update from the client: Fixed in commit f6c5609

6.6 [Low] Incorrect calculation of `totalPages` in getter functions

File(s): `ClayMain.sol`, `NodeManager.sol`, `deposits/DepositsManager.sol`

Description: In various contract getter functions that support pagination, there is an issue with the computation of the `totalPages` variable. This leads to an incorrect value, exceeding the expected count by 1 in certain scenarios.

Let's examine one such case. In the provided code snippet, where `pageSize` is set to 10, the intended behaviour is that if the number of user orders (`length`) falls within the range of 1 to 10, the `totalPages` should have the same value. However, the current calculation results in different outcomes: when `length = 5`, `totalPages = 0`, and when `length = 10`, `totalPages = 1`.

```
uint256 pageSize = 10;
uint256 length = userWithdrawIds[_user].length;
uint256 totalPages = length / pageSize;
```

Recommendation(s): Consider correcting the calculation of `totalPages` in these getter functions.

Status: Fixed

Update from the client: Fixed in commit 89dc5e

6.7 [Low] Skipping invariant check when `userFlow == 0` breaks its meaning

File(s): `ClayMain.sol`

Description: The function `_balanced(...)` includes an invariant check that must hold in all cases. However, when `userFlow == 0`, the check is skipped, without taking into account the value of `stakingFlow`. This behaviour breaks the meaning of the invariant and could potentially enable attackers to cause `userFlow` to be nearly 0 while `stakingFlow > 0`.

```
// @audit In case userFlow = 0, invariant check is skipped
if (userFlow != 0) {
    uint256 stakingFlow = (address(this).balance +
        funds.stakedDeposit +
        funds.unstakeInternal +
        funds.unstakeExternal) / 1e16;
    require(stakingFlow <= userFlow + 1 && userFlow <= stakingFlow + 1, "Unbalanced contract");
}

```

Recommendation(s): Consider modifying the function `_balanced(...)` to ensure that the invariant check holds in all cases.

Status: Fixed

Update from the client: Fixed in commit eaef200

6.8 [Low] The `_closeBatches()` function returns a different output from the `fulfilledBatches`

File(s): `NodeManager.sol`

Description: The `_closeBatches()` function closes batches that were in the exit queue. Each exit request is associated with a batch id and each batch id is related to a request count, which is the number of exits contained in the request.

If the request contains too many exit orders, the function goes to the next batch request to fulfill it.

```
for (uint256 i = 0; i < batchExitQueue.length; i++) {
    uint256 batchId = batchExitQueue[i];
    uint256 requestCount = batchExitRequests[batchId];
    //@audit: if batch request has too many exit orders
    //we will try to fulfill the next batch
    if (requestCount <= exitsTotal) {
        fulfilledBatches++;
        exitsTotal -= requestCount;
        fulfilledCount += requestCount;
    }
    if (exitsTotal <= 0) break;
}
```

The `_closeBatches()` function then creates an output report that will be provided to the `closeBatches()` function from the `clayMain` contract.

This output takes batches in the exit queue sequentially without taking into account the batches that were avoided because of their `requestCount` value.

```
[...]
// create output
uint256[] memory batches = new uint256[](fulfilledBatches);
for (uint256 i = 0; i < fulfilledBatches; i++) {
    //@audit: output sequentially pops the first element of the exit queue
    //based on fulfillBatches value regardless the number of requestCount
    batches[i] = batchExitQueue[0];
    _deleteBatchFromQueue(0);
}

clayMain.closeBatches(batches);
[...]
```

Recommendation(s): Modify the function to send effectively fulfilled batches to the `closeBatches()` function of `ClayMain` contract.

Status: Fixed

Update from the client: Suggested change simplifies logic updating partial fulfilled batches and exiting loop. Added unit test from your example too

Commit: 4880df2

6.9 [Low] Update of `claimablePool` Amount Prior to ETH Transfer

File(s): `ClayMain.sol`

Description: In both `claim(...)` and `instantWithdraw(...)` functions, `_updateClaimablePool()` is called to update the `claimablePool` amount, the call is followed by an ETH transfer.

```
function claim(uint256[] calldata _orderIds) external override whenNotPaused nonReentrant returns (bool) {
    [...]
    _updateClaimablePool();

    // @audit ETH transfer after updating the `claimablePool` amount

    (bool success, ) = msg.sender.call{value: amountsToUser}("");
    [...]
}
```

```
function instantWithdraw(uint256 _amountCs) external whenNotPaused nonReentrant returns (bool) {
    [...]
    _updateClaimablePool();
    [...]
    // @audit ETH transfer after updating the `claimablePool` amount
    (bool success, ) = msg.sender.call{value: amountTokenWithdrawToUser}("");
    [...]
}
```

The `_updateClaimablePool()` function uses the contract balance to adjust its amount, which might result in inaccurate accounting.

```
function _updateClaimablePool() internal {
    [...]
    funds.claimablePool = _max(
        _min(funds.claimablePool, funds.withdrawQueue),
        _min(funds.withdrawQueue - unstakeExternal, address(this).balance)
    );
}
```

Recommendation(s): Consider moving `_updateClaimablePool()` call after the ETH transfer in both functions.

Status: Fixed

Update from the client: Fixed in commit d2d0a0

6.10 [Low] `ClayStack` protocol does not have any protection against deposit front-running

File(s): `NodeManager.sol`

Description: Once enough amount of ETH is available (at least 32 ETH), the `autobalance()` function from the `NodeManager` contract is in charge of depositing the ETH to the Beacon Chain Deposit contract.

This function takes the following parameters:

- The public key of the validator key;
- The hash of the withdrawal public key;
- Signature for the data above signed with the validator's private key;

The root cause of the vulnerability comes from the fact that Beacon Chain permits multiple deposits for a single validator public key. It does not verify that all deposits have the same withdrawal credentials. Moreover, it will assume that the withdrawal credential is in the first valid deposit.

The risk is that a malicious validator front-run the deposit to be made by the `NodeManager` contract specifying a controlled withdrawal credential instead of the `NodeManager` address. The rogue validator would be able to steal the funds brought by the `NodeManager`.

The severity of this finding is rated Low because validators are trusted entities that are not likely to try to attack the protocol.

However, this problem should be addressed in the future when validators will be external (and untrusted) entities.

Ref: https://github.com/rocket-pool/rocketpool-research/blob/master/Reports/withdrawal-creds-exploit.md

Recommendation(s): Implement a protection that checks that the deposit has not been front-runned. Ref: https://research.lido.fi/t/mitigations-for-deposit-front-running-vulnerability/1239

Status: Fixed

Update from the client: Fixed in commit ecd3e2e

6.11 [Low] `getOrder` does not consider all possible Strategy statuses

File(s): `deposits/DepositsManager.sol`

Description: The `DepositsManager` contract currently interacts with `StrategyStETH` and `StrategyRETH` strategies that primarily use three order statuses: `Unbonding`, `Claimable`, and `Claimed`. However, the contract may potentially interact with other future strategies that could introduce additional order statuses such as `Expired`. As of now, the `getOrder` logic is heavily tied to the existing strategies and doesn't cover all potential cases that may arise in the future, which could lead to incorrect status determination for certain strategies.

```
enum OrderStatus {
    Unbonding,
    Claimable,
    Expired,
    Claimed
}
```

The current logic considers only `Unbonding` status and handles the rest as `Claimable`, the `Claimed` status returned by the strategy is never taken into consideration. Moreover, `Expired` status is currently handled as `Claimable`.

```
IStrategy.OrderStatus statusStrategy = strategy.getStatus(order.id);
if (statusStrategy == IStrategy.OrderStatus.Unbonding) {
    status = Status.Unbonding;
} else {
    //@audit All other statuses are narrowed to `Claimable`.
    status = Status.Claimable;
}
```

Recommendation(s): Consider revisiting the current behaviour of the function and take into consideration all possible strategy statuses that may emerge in the future. This would involve extending the conditional checks to handle `Claimed` and `Expired` statuses.

Status: Fixed

Update from the client: Fixed in commit af3e1d

6.12 [Low] `previousExits` updated with the total unfulfilled exits requests

File(s): `NodeManager.sol`

Description: In the `_closeBatches(...)` function, `previousExits` variable is summed up with the newly reported exits to represent the total not used exits. This total is used to fulfill and close batches in the queue.

```
uint256 exitsTotal = _newExits + previousExits;
```

However, when updating this variable, it is set to the total of unfulfilled exit requests.

```
previousExits = _min(exitsTotal, _positiveSub(exitRequestsCount, fulfilledCount));
```

Recommendation(s): Consider not using the total unfulfilled exit requests when updating `previousExits`.

Status: Fixed

Update from the client:

The suggested changes simplifying the approach will avoid keeping track of partial batches in a variable: Commit: 4880df2

6.13 [Low] `setClayMain()` function initializer is not restricted

File(s): `CsToken.sol`

Description: The `setClayMain()` function does not have any access control. A malicious user could front-run initialization of the `CsToken` contract with a controlled `ClayMain` contract address.

```
/// @notice Sets `_clayMain` address.
/// @dev ClayMain can be set only once.
/// @param _clayMain : Address of new ClayMain contract.
function setClayMain(address _clayMain) external {
    require(_clayMain != address(0), "Invalid ClayMain");
    require(clayMain == address(0), "ClayMain Already Set");
    clayMain = _clayMain;
}
```

Recommendation(s): Consider making the `clayMain` variable immutable and initialize it within the `CsToken` contract `constructor()`.

Status: Acknowledged

Update from the client: Noted. This is concious by design. Setting it on constructor creates an extra complexity step as address needs to be pre-computed given the dependency as well on ClayMain, while making the testing more complex. There is no risk of front running as at initial deployment time the addressed will be verified before the contracts are then considered as the protocol's contract.

6.14 [Low] `withdrawQueueNew` always resets to `0` when swapping internal and external unstake

File(s): `ClayMain.sol`

Description: In the `ClayMain` contract, there are two variables that account for the unstaking amount. The state variable `funds.unstakeExternal` is the amount of ETH that has already been unstaked from validators. While the state variable `funds.unstakeInternal` is the amount of ETH waiting to be unstaked from validators.

Also, `funds.withdrawQueueNew` accounts for the amount of ETH in the current batch of withdrawals that has not yet been accounted for in auto balance. This variable is checked to decide whether to close the current batch or not. However, its value is always reset to 0 in the function `_updateClaimablePool(...)` after swapping internal and external unstake. As a result, the batch of withdrawals is ignored and not closed.

```
function _updateClaimablePool() internal {
    uint256 unstakeExternal = funds.unstakeExternal;

    // Unstake update swap given new withdrawals
    if (funds.unstakeInternal > 0) {
        uint256 swaps = _min(
            funds.withdrawQueue - funds.unstakeExternal - funds.claimablePool,
            funds.unstakeInternal
        );
        unstakeExternal += swaps;
        // @audit funds.withdrawQueueNew is always reset to 0
        funds.withdrawQueueNew = 0;
        funds.unstakeExternal = unstakeExternal;
        funds.unstakeInternal = funds.unstakeInternal - swaps;
    }
    ...
}
```

Recommendation(s): Consider not resetting `withdrawQueueNew` to `0` when swapping internal and external unstake.

Status: Fixed

Update from the client: Fixed in commit a1967e0

6.15 [Low] `withdrawQueueNew` is not updated after closing a batch

File(s): `ClayMain.sol`

Description: In the `ClayMain` contract, the state variable `funds.withdrawQueueNew` keeps track of the amount of ETH in the current batch of withdrawals that has not yet been accounted for in autobalance. This variable is checked to decide whether to close the current batch or not. However, its value is not reset after closing a batch, which could cause extra batches to close.

```
// Close batch if staked or if withdrawals within batch to unlock them
// @audit funds.withdrawQueueNew is not reset after closing batch
if (validatorCount != 0 || funds.withdrawQueueNew != 0) {
    _closeBatch(currentBatchId, minClaimDays);
    batchId++;
}
```

Recommendation(s): Consider updating value of `withdrawQueueNew` when closing a batch.

Status: Fixed

Update from the client: Fixed in commit d486bea

6.16 [Best Practices] Confusing parameter naming (`minClaimDays`)

File(s): `ClayMain.sol`

Description: The `minClaimDays` state variable is used to allow withdrawal orders to be processed after a minimum amount of time has passed. However, this state variable is a `uint256`, which refers to elapsed seconds.

```
/// @notice Sets the minimum time for a claim tokens when netted locally.
/// @param _seconds claim time
function setMinClaimTime(uint256 _seconds) external onlyRole(TIMELOCK_ROLE) {
    require(_seconds <= MAX_UNBONDING, "Minimum claim time too high");
    minClaimDays = _seconds;
}
```

Recommendation(s): Consider renaming the `minClaimDays` state variable to reflect its unit (seconds).

Status: Fixed

Update from the client: Fix ee2514

6.17 [Best Practices] Consider making `withdrawQueueNew` a boolean instead of uint256

File(s): `ClayMain.sol`

Description: In the `ClayMain` contract, `funds.withdrawQueueNew` is only incremented, checked against zero, and then reset to zero. Additionally, no other contracts appear to use this variable. Given these observations, `funds.withdrawQueueNew` could be modified to be a boolean to save gas.

```
...
funds.withdrawQueueNew += amountTokenWithdraw;
...
if (validatorCount != 0 || funds.withdrawQueueNew != 0) {
...
funds.withdrawQueueNew = 0;
```

Recommendation(s): Consider modifying `funds.withdrawQueueNew` to be a boolean instead of a uint256.

Status: Fixed

Update from Client: Changed in 9f843c4

6.18 [Best Practices] Emit events on important updates

File(s): `ClayMain.sol` `NodeManager.sol`

Description: Some functions in both `ClayMain.sol` and `NodeManager.sol` contracts, are updating important storage variables and not emitting any event. It is a good practice to emit an event in that case to enhance traceability and facilitate monitoring of the product.

Some functions that are updating important parameters and could emit an event are:

- `NodeManager.setClayMain(...);`
- `NodeManager.setNodeOperator(...);`
- `ClayMain.setMinClaimTime(...);`
- `ClayMain.setWithdrawalsDisabled(...);`
- `ClayMain.setExchangeDeviation(...);`

Recommendation(s): Consider emitting an event where relevant.

Status: Fixed

Update from the client: Added 71c680

6.19 [Best Practices] Function `getValidators()` in the `NodeManager` could break earlier to save gas

File(s): `NodeManager.sol`

Description: In the `NodeManager` contract, the `validators[]` list starts with `id = 1`, making `validators[0]` non-existent. However, the function `getValidators(...)` still accepts `index = 0` as a valid input, and this can lead to unexpected behaviour and cost more gas. The relevant code snippet is shown below:

```solidity
function getValidators(uint256 _page) external view returns (ValidatorView[] memory, uint256) {
    ValidatorView[] memory info = new ValidatorView[](10);
    uint256 pageSize = 10;
    uint256 length = validatorNonce;
    uint256 totalPages = length / pageSize;
    if (_page <= totalPages && length != 0) {
        for (uint256 i = 0; i < pageSize; i++) {
            uint256 index = length - _page * pageSize - i;
            Validator memory validator = validators[index];
            info[i] = ValidatorView({
                id: index,
                publicKey: validator.publicKey,
                validatorType: validator.validatorType
            });
            // @audit validators index 0 is non-existent
            if (index == 0) break;
        }
    }
    return (info, totalPages);
}
```

Recommendation(s): Consider breaking the loop when `index` has a value of 1 instead of 0.

Status: Fixed

Update from the client: As discussed, this is not an issue but the function returns as expected. Added the following unit test to aid the discussion: Commit c0df599

Furthermore as part of the discussion, gas can be saved by breaking the loop one iteration earlier. Implemented in this commit 3fecb6b

6.20 [Best Practices] Lack of existence check for Strategy access

File(s): `DepositsManager.sol`

Description: In multiple functions within the `DepositsManager.sol` contract, the strategy address is retrieved from storage without prior existence validation. This could lead to failures in calls to the strategy without clear error messages.

In both `claim(...)` and `getOrder(...)` functions, the `orderId` is provided as an input and used to access the associated strategy address. However, the case where the order does not exist is not handled.

```solidity
function claim(
    uint256[] calldata _orderIds,
    bytes[] memory _data
) external whenNotPaused nonReentrant returns (bool) {
require(_orderIds.length == _data.length, "Param length mismatch");
for (uint256 i = 0; i < _orderIds.length; i++) {
    uint256 id = _orderIds[i];
    WithdrawOrder memory order = withdrawOrders[id];
    //@audit order could be non-existent.
    require(!order.isClaimed, "Already claimed");

    // Claim from strategy
    IStrategy strategy = IStrategy(strategies[order.token].strategy);

    //@audit A call to address(0) will occur if `_orderIds[i]` does not correspond to an existing order.
    uint256 amountCsETH = strategy.claim(order.id, order.owner, _data[i]);
    [...]
}
```

```solidity
function getOrder(uint256 _orderId) public view returns (UserWithdrawOrderInfo memory) {
    //@audit order could be non-existent.
    WithdrawOrder memory order = withdrawOrders[_orderId];

    [...]
    //@audit Similarly, if `_orderId` does not correspond to an existing order, a call to `address(0)` will occur.
    IStrategy.OrderStatus statusStrategy = strategy.getStatus(order.id);
    [...]
}
```

The function `updateStrategy` is used to update the strategy status of a given token to active/inactive. However, it lacks a check to determine whether a strategy is registered for the given token.

```
function updateStrategy(address _token, bool _isActive) external nonReentrant onlyRole(TIMELOCK_ROLE) {
    require(_token != address(0), "Invalid token in strategy");
    // @audit A check should be included to ensure that a strategy exists for the provided token (strategies[_token] !=
↪   address(0))
    strategies[_token].isActive = _isActive;
}
```

Recommendation(s): Consider implementing existence checks before accessing strategies and orders. This can be achieved by adding a conditional check such as:

```
require(strategies[order.token].strategy != address(0), "Strategy does not exist for this order token")
```

in both the `claim(...)` and `getOrder(...)` functions.

Additionally, in the `updateStrategy(...)` function, include a check like the following to ensure a strategy is registered for the provided token before updating its status.

```
require(strategies[_token].strategy != address(0), "Strategy does not exist for this token");
```

Implementing these checks provides users with clear error messages in case of non-existent strategies or orders.

Status: Fixed

Update from the client: Fixed in commit 834af0

6.21 [Best Practices] Missing balance check in claim function

File(s): `ClayMain.sol`

Description: The `claim(...)` function in the `DepositsManager` contract allows users to input a list of order IDs to be claimed. Within the function, the code iterates through these orders and calls the internal function `_claim(...)` for each order. This internal function is responsible for verifying if both the `ClaimablePool` and the contract's balance are sufficient to cover the order amount.

In scenarios where the cumulative amounts of the orders exceed the available `ClaimablePool` amount, the function appropriately handles this situation as it updates the `ClaimablePool` during the iteration. However, the contract balance is individually checked for each order and not updated.

```
for (uint256 i = 0; i < _orderIds.length; i++) {
    (uint256 _amountsToUser, uint256 _orderFee) = _claim(_orderIds[i], msg.sender);
    amountsToUser += _amountsToUser;
    ordersFees += _orderFee;
}
[...]
//@audit There is no guarantee that contract balance is sufficient.
(bool success, ) = msg.sender.call{value: amountsToUser}("");
```

The variable `amountsToUser` stores the total amount of ETH to be sent to the user, `_claim` function does not guarantee that the contract balance covers this value. As a result, if the contract's balance is insufficient, the ETH transfer will fail without providing a clear error message.

Recommendation(s): In order to enhance code clarity and prevent unexpected failures, consider adding a balance check before performing the ETH transfer.

Status: Fixed

Update from the client: Given the update due to the self-destruct fix, the contract will update the **ethBalance** variable and when balance is below it should revert.

6.22 [Best Practices] The `getValidators()` function's documentation is incorrect

File(s): `NodeManager.sol`

Description: The `getValidators()` function's documentation mentions that this function returns a list of nodes for a user. However, this function does not return the list of nodes depending on a specific user.

```
/// @notice Returns validators and status
/// @notice Returns list of nodes for a user.
/// @dev Max 10 results starting at page 0
/// @param _page : Page to query.
/// @return info Array for struct of nodes view
/// @return totalPages supported for given user.
function getValidators(uint256 _page) external view returns (ValidatorView[] memory, uint256) {
[...]
```

Recommendation(s): Modify the documentation to reflect what the function does.

Status: Fixed

Update from the client: Changed in 3037af

6.23 [Best Practices] Token strategy is not checked for existence before adding in the function `addStrategy(...)`

File(s): `deposits/DepositsManager.sol`

Description: In the `DepositsManager` contract, the timelock can call the function `addStrategy(...)` to add a new liquid strategy contract for a token or call the function `updateStrategy(...)` to update the strategy contract. However, the function `addStrategy(...)` does not check if the strategy for the token already exists or not. This can result in the existing strategy being modified to a new address, which can cause funds to be locked in the previous strategy.

```
// @audit Not check if strategy for `_token` existed
function addStrategy(address _strategy, address _token) external nonReentrant onlyRole(TIMELOCK_ROLE) {
    require(_strategy != address(0), "Invalid strategy");
    IStrategy strategy = IStrategy(_strategy);
    require(strategy.isValidToken(_token), "Invalid token in strategy");
    strategies[_token] = IStrategy.Strategy({strategy: _strategy, isActive: true});
}

```

Recommendation(s): We recommend that you add a check in the `addStrategy(...)` function to ensure that the strategy for the token does not already exist before modifying it.

Status: Fixed

Update from Client: Fixed in commit 053208

6.24 [Best Practices] Unused import of `PausableUpgradeable` contract

File(s): `deposits/strategies/StrategyStETH.sol`, `deposits/strategies/StrategyRETH.sol`

Description: Both `StrategyStETH` and `StrategyRETH` contracts import `PausableUpgradeable` but do not use it in their logic.

Recommendation(s): For better code readability, consider removing unused imports.

Status: Fixed

Update from the client: Fixed in commit 562b9d

6.25 [Best Practices] `updateBalances()` function visibility can be reduced to external

File(s): `ClayMain.sol`

Description: The current `updateBalances()` function is declared with a `public` visibility. But, it is exclusively invoked from outside the contract.

Recommendation(s): Consider setting the visibility to `external`.

Status: Fixed

Update from the client: Fixed in commit d37162

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.

The ClayStack contracts' documentation is presented through inline comments within the code, providing explanations for the objectives and operations of various functions and formulas. Additionally, the README file, accessible at this link, outlines the deposit and withdrawal processes and delineates different roles within the system. Furthermore, the `ClayStack` team remained available to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Contracts Compilation

```
> forge compile
[] Compiling...
[] Compiling 123 files with 0.8.18
[] Solc 0.8.18 finished in 16.60s
Compiler run successful with warnings.
```

8.2 Tests Output

```
> forge test
[] Compiling...
No files changed, compilation skipped

Running 10 tests for test/staking/RoleManager.t.sol:TestRoleManager
Test result: ok. 10 passed; 0 failed; finished in 10.86ms

Running 17 tests for test/deposits/rETH.t.sol:rETHTest
Test result: ok. 17 passed; 0 failed; finished in 19.78ms

Running 3 tests for test/staking/Setters.t.sol:TestSetters
Test result: ok. 3 passed; 0 failed; finished in 39.09ms

Running 2 tests for test/staking/UserClaimsView.t.sol:TestUserClaimsView
Test result: ok. 2 passed; 0 failed; finished in 78.10ms

Running 6 tests for test/staking/InstantWithdrawal.t.sol:TestInstantWithdraw
Test result: ok. 6 passed; 0 failed; finished in 116.87ms

Running 14 tests for test/staking/AutoBalance.t.sol:AutoBalance
Test result: ok. 14 passed; 0 failed; finished in 50.87ms

Running 8 tests for test/staking/CsToken.t.sol:TestCsToken
Test result: ok. 8 passed; 0 failed; finished in 175.32ms

Running 4 tests for test/staking/TimeLock.t.sol:TestTimeLock
Test result: ok. 4 passed; 0 failed; finished in 6.94ms

Running 3 tests for test/staking/RegisterValidator.t.sol:TestRegisterValidator
Test result: ok. 3 passed; 0 failed; finished in 8.39ms

Running 17 tests for test/deposits/stETH.t.sol:stETHTest
Test result: ok. 17 passed; 0 failed; finished in 24.23ms

Running 5 tests for test/staking/NodeManager.t.sol:TestNodeManager
Test result: ok. 5 passed; 0 failed; finished in 13.01ms

Running 13 tests for test/staking/Oracle.t.sol:Oracle
Test result: ok. 13 passed; 0 failed; finished in 55.62ms

Running 18 tests for test/staking/OracleNetwork.t.sol:OracleNetwork
Test result: ok. 18 passed; 0 failed; finished in 58.74ms

Running 31 tests for test/deposits/wstETH.t.sol:wstETH
Test result: ok. 31 passed; 0 failed; finished in 61.45ms

Running 6 tests for test/staking/RateProtection.t.sol:RateProtection
Test result: ok. 6 passed; 0 failed; finished in 4.35s

Running 4 tests for test/staking/Donations.test.sol:TestDonations
Test result: ok. 4 passed; 0 failed; finished in 142.55ms

Running 8 tests for test/staking/Penalties.t.sol:Penalties
Test result: ok. 8 passed; 0 failed; finished in 17.60ms
```

```
Running 10 tests for test/staking/Withdraw.t.sol:TestWithdraw
Test result: ok. 10 passed; 0 failed; finished in 495.80ms

Running 8 tests for test/staking/Deposit.t.sol:TestDeposit
Test result: ok. 8 passed; 0 failed; finished in 447.15ms

Running 8 tests for test/staking/SetFee.t.sol:TestSetFee
Test result: ok. 8 passed; 0 failed; finished in 41.33s

Running 16 tests for test/staking/Claim.t.sol:TestClaim
Test result: ok. 16 passed; 0 failed; finished in 41.33s
```

8.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

```
| File                                     | % Lines            | % Statements       | % Branches         | % Funcs           |
|------------------------------------------|--------------------|--------------------|--------------------|-------------------|
| ClayMain.sol                             | 94.86% (314/331)   | 95.56% (366/383)   | 82.35% (140/170)   | 97.62% (41/42)    |
| CsToken.sol                              | 85.71% (6/7)       | 85.71% (6/7)       | 75.00% (3/4)       | 100.00% (3/3)     |
| NodeManager.sol                          | 93.53% (130/139)   | 94.89% (167/176)   | 78.21% (61/78)     | 95.65% (22/23)    |
| RoleManager.sol                          | 37.50% (6/16)      | 33.33% (6/18)      | 66.67% (4/6)       | 57.14% (4/7)      |
| deposits/DepositsManager.sol             | 91.14% (72/79)     | 92.93% (92/99)     | 83.33% (35/42)     | 93.33% (14/15)    |
| deposits/strategies/StrategyRETH.sol     | 62.96% (17/27)     | 65.52% (19/29)     | 50.00% (8/16)      | 90.91% (10/11)    |
| deposits/strategies/StrategyStETH.sol    | 75.81% (47/62)     | 80.00% (60/75)     | 56.25% (18/32)     | 91.67% (11/12)    |
| Total                                    | 89.56% (592/661)   | 90.98% (716/787)   | 77.30% (269/348)   | 92.92% (105/113)  |
```

8.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

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