Security Review Report
NM-0093 Ether Fi

NETHERMIND

(June 22, 2023)

Contents

1 Executive Summary

This document outlines the security review conducted by Nethermind for the ether.fi protocol. `ether.fi` is a decentralized, non-custodial delegated staking protocol with a Liquid Staking Derivative token. One of the distinguishing characteristics of `ether.fi` is that stakers control their keys. The `ether.fi` mechanism also allows for the creation of a node services marketplace where stakers and node operators can enroll nodes to provide infrastructure services

The audited code comprises 2894 lines of Solidity, achieving a code coverage of 81.46%. The `ether.fi` team provided a **README** file containing instructions for compiling and running tests. In addition, presentations explaining some of the mechanisms and official documentation were available, offering a good explanation of the system's functionality.

We have observed a noteworthy presence of **circular dependencies in the smart contract structure of the system**. These dependencies can significantly elevate the complexity of the code, making comprehension, modification, and bug-fixing more challenging. Circular dependencies typically occur when two or more modules depend on each other directly or indirectly.

After thoroughly examining the current implementation of the `ether.fi` protocol, we propose conducting comprehensive reviews and extensive testing before contemplating any deployment decisions. The meticulous testing process holds the utmost importance in guaranteeing the solidity and security of the system, particularly when implementing substantial changes or introducing new features.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. **Along this document, we report** 33 points of attention, where one is classified as `Critical`, five are classified as `High`, eight are classified as `Medium`, six are classified as `Low`, and fourteen 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 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.

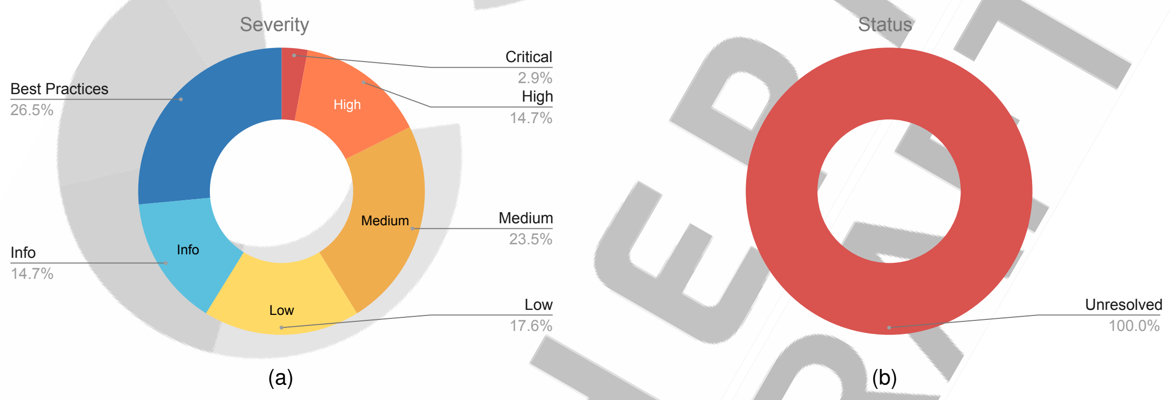

Fig 1: Distribution of issues: Critical (1), **High** (5), **Medium** (8), **Low** (6), **Undetermined** (0), **Informational** (5), **Best Practices** (9).
Distribution of status: Fixed (0), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (34)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jun 22, 2023
Response from Client	-
Final Report	-
Methods	Manual Review, Automated Analysis
Repository	ether.fi
Commit Hash (Initial Audit)	3a29e7462ecada4d134128d074f17d928809fc2c
Documentation	README.md, Official documentation, Membership program
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/EtherFiNode.sol	371	117	31.5%	54	542
2	src/BNFT.sol	39	25	64.1%	15	79
3	src/EtherFiNodesManager.sol	388	148	38.1%	81	617
4	src/TNFT.sol	31	24	77.4%	14	69
5	src/ProtocolRevenueManager.sol	100	39	39.0%	28	167
6	src/StakingManager.sol	255	80	31.4%	70	405
7	src/WeETH.sol	51	26	51.0%	16	93
8	src/RegulationsManager.sol	65	26	40.0%	23	114
9	src/NodeOperatorManager.sol	108	50	46.3%	33	191
10	src/UUPSProxy.sol	8	1	12.5%	2	11
11	src/AuctionManager.sol	219	71	32.4%	51	341
12	src/EETH.sol	83	5	6.0%	26	114
13	src/MeETH.sol	439	122	27.8%	127	688
14	src/Treasury.sol	15	6	40.0%	6	27
15	src/LiquidityPool.sol	162	43	26.5%	49	254
16	src/MembershipNFT.sol	118	26	22.0%	39	183
17	src/interfaces/IStakingManager.sol	19	1	5.3%	5	25
18	src/interfaces/ITreasury.sol	4	1	25.0%	1	6
19	src/interfaces/IProtocolRevenueManager.sol	27	1	3.7%	10	38
20	src/interfaces/IeETH.sol	15	1	6.7%	3	19
21	src/interfaces/IDepositContract.sol	18	16	88.9%	5	39
22	src/interfaces/IEtherFiNode.sol	78	5	6.4%	26	109
23	src/interfaces/IWETH.sol	6	0	0.0%	3	9
24	src/interfaces/ILiquidityPool.sol	23	1	4.3%	6	30
25	src/interfaces/IRegulationsManager.sol	9	1	11.1%	7	17
26	src/interfaces/IEtherFiNodesManager.sol	126	3	2.4%	36	165
27	src/interfaces/ImeETH.sol	55	4	7.3%	14	73
28	src/interfaces/INodeOperatorManager.sol	22	1	4.5%	6	29
29	src/interfaces/IBNFT.sol	6	1	16.7%	3	10
30	src/interfaces/ITNFT.sol	6	1	16.7%	3	10
31	src/interfaces/IAuctionManager.sol	28	1	3.6%	13	42
	Total	**2894**	**847**	**29.3%**	**775**	**4516**

3 Summary of Issues

	Finding	Severity	Update
1	The LiquidityPool.withdraw(...) function can by called for any address	Critical	Unresolved
2	Deposit may be front-run with arbitrary withdrawal credentials	High	Unresolved
3	Owner can withdraw users' assets as fees	High	Unresolved
4	Protocol rewards may be claimed after the node exit	High	Unresolved
5	Staked funds are frozen after slashing	High	Unresolved
6	Treasury and ProtocolRevenueManager owner can not withdraw eETH tokens	High	Unresolved
7	Centralization risks	Medium	Unresolved
8	Check in LiquidityPool.deposit(...) is incorrect	Medium	Unresolved
9	Incorrect calculation in `convertEapPoints(...)`	Medium	Unresolved
10	Marking the node as EXITED before receiving funds may lead to locked funds	Medium	Unresolved
11	Points diluting mechanism does not disincentivize users from `topUp` with huge amounts	Medium	Unresolved
12	The B-NFT holder is unfairly penalized when the T-NFT holder requests to exit the node	Medium	Unresolved
13	Users can get more value than what they should when withdrawing from the pool	Medium	Unresolved
14	_processNodeExit(...) never distributes the protocol reward to the validator's node	Medium	Unresolved
15	BurnFee never initialized or set	Low	Unresolved
16	Function markBeingSlahsed(...) does not check node phase	Low	Unresolved
17	Incorrect deposit address	Low	Unresolved
18	Partial withdrawal differs from batch partial withdrawal	Low	Unresolved
19	Potential price manipulation by the first depositor in `LiquidityPool`	Low	Unresolved
20	The operator is incentivized to delay exit	Low	Unresolved
21	Inconsistent access control in the `getFullWithdrawalPayouts(...)` function	Info	Unresolved
22	Check if `returnAmount` is not zero	Info	Unresolved
23	EETH approval race condition	Info	Unresolved
24	Misleading error message	Info	Unresolved
25	Usage of __gap	Info	Unresolved
26	Circular dependency and lack of cohesion in contracts	Best Practices	Unresolved
27	Lack of events for relevant operations	Best Practices	Unresolved
28	Local variable tokenData shadows the existing state variable	Best Practices	Unresolved
29	No explicit return value in _updateAllTimeHighDepositOf(...)	Best Practices	Unresolved
30	Place-holder in the struct for future usage	Best Practices	Unresolved
31	Some functions can be external	Best Practices	Unresolved
32	Unnecessary update of `prevPointsAccrualTimestamp`	Best Practices	Unresolved
33	Unused code	Best Practices	Unresolved
34	Use of 2-step ownership transition	Best Practices	Unresolved

4 System Overview

This audit encompasses 16 contracts and 15 interfaces. Fig. 2 illustrates a structural diagram of the core contracts.

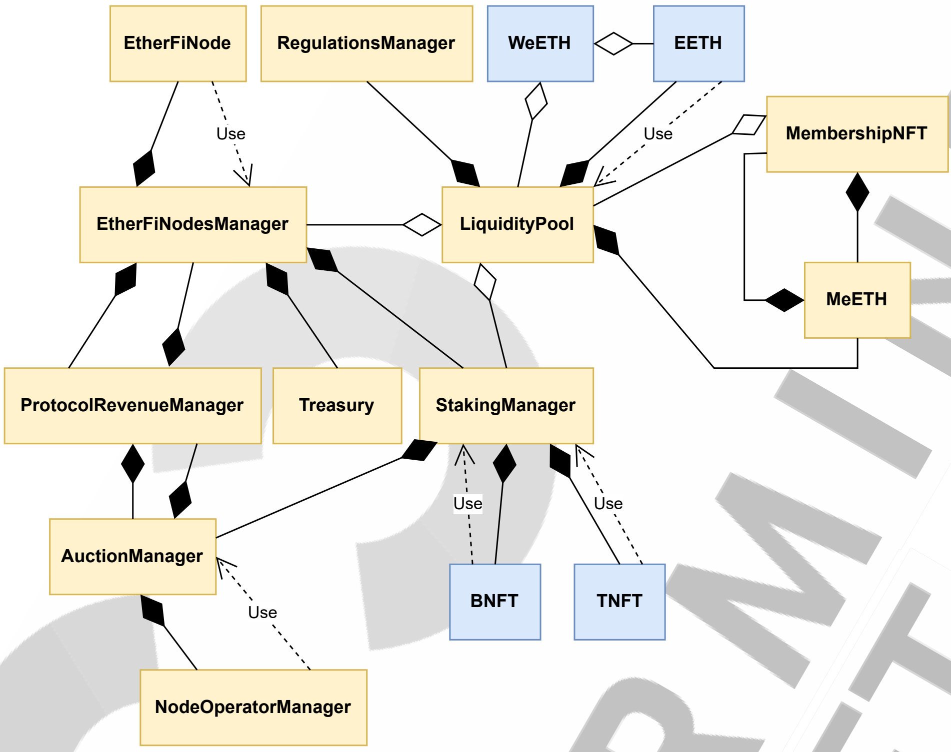

Fig. 2: Structure Diagram of EtherFi

4.1 EtherFiNode

The **EtherFiNode** contract implements the `IEtherFiNode` interface. The contract uses the struct `VALIDATOR_PHASE` to update the state of the validator:

```
enum VALIDATOR_PHASE { STAKE_DEPOSITED, LIVE, EXITED, CANCELLED}
```

The **EtherFiNode** contains several state-changing and view functions that allow only the **EtherFiNodesManager** contract calls. The main functions of the contract are:

- `setPhase(...)`: Updates the validator phase;
- `setIpfsHashForEncryptedValidatorKey(...)`: Sets the IPFS hash of the validator's encrypted private key;
- `setLocalRevenueIndex(...)`: Updates and resets the local revenue index;
- `setExitRequestTimestamp(...)`: T-NFT holders call this function to send exit requests to corresponding B-NFT holders;
- `markExited(...)`: When the protocol processes the node's exit, this function is called to mark it as EXITED;
- `markBeingSlahsed(...)` and `markFullyWithdrawn(...)`: These functions are quite similar to `markExited(...)`. It only updates the node's phase when it is being slashed and when the staker has fully withdrawn after the node is exited.
- `withdrawFunds(...)`: This function is called for different reasons: partial and full withdrawals and during the node's exit process.
- `getRewardsPayouts(...)`: Computes the payouts for staking and protocol rewards and also calculates vested auction fees;
- `getNonExitPenalty(...)`: Computes the non-exit penalty for the B-NFT holder based on the principal, daily penalty, and exit timestamp;
- `getFullWithdrawalPayouts(...)`: Computes the payouts to the node operator, T-NFT holder, B-NFT holder, and treasury based on the current balance of the EtherFiNode after its exit.
- `receiveVestedRewardsForStakers(...)`: Allows only the **Protocol Revenue Manager** calls to send vested rewards for stakers.

4.2 RegulationsManager

The **RegulationsManager** contract is an implementation of the `IRegulationsManager` interface and is inherited by `Initializable`, `OwnableUpgradeable`, `PausableUpgradeable`, and `UUPSUpgradeable` libraries.

The contract maintains and manages a whitelist of addresses not associated with blacklisted countries. The whitelist version is stored in `whitelistVersion`. The contract includes eight functions, and the main ones are described below:

- `confirmEligibility(...)`: Adds an address to the whitelist, confirming that the address is not associated with any blacklisted countries. If the hash provided in the function call matches the correct version hash, the user's eligibility is confirmed and added to the whitelist.
- `removeFromWhitelist(...)`: Removes an address from the whitelist. This function can only be called by the owner or the user themself.
- `initializeNewWhitelist(...)`: Initializes a new version of the whitelist by incrementing the `whitelistVersion`.

4.3 Treasury

The contract **Treasury** implements the interface `ITreasury` and inherits the `Ownable` contract. The contract allows ether to be sent and implements the function `withdraw(...)` for the owner to withdraw all funds in the contract.

4.4 EETH

The **EETH** contract represents a tokenized version of `ETH` backed by a liquidity pool. It implements the interfaces `IERC20Upgradeable` and `IeETH` and inherits the contracts `UUPSUpgradeable` and `OwnableUpgradeable`. The contract contains 12 external/public functions allowing users to mint and burn token shares. The pool contract functions can only call the mint and burn functions. Also, it has getter functions and others to transfer shares, such as: `transfer(...)`, `approve(...)`, and `transferFrom(...)` functions, etc.

4.5 BNFT and TNFT

The **BNFT** and **TNFT** contracts follow the ERC 721 standard. When creating a validator node, both `TNFT` and `BNFT` are minted. The `BNFT` contract is an implementation of the `ERC721Upgradeable`, `UUPSUpgradeable`, and `OwnableUpgradeable` contracts. B-NFTs are soul-bound to the staker, meaning they cannot be transferred to another address once minted. The function `mint(...)` can be called only by the `StakingManager` contract. The `TNFT` contract also inherits the `ERC721Upgradeable`, `UUPSUpgradeable`, and `OwnableUpgradeable` contracts. As opposed to the B-NFT tokens, T-NFT is transferable.

4.6 NodeOperatorManager

The **NodeOperatorManager** contract implements the `INodeOperatorManager` interface. It also inherits the contracts `Initializable`, `UUPSUpgradeable`, `PausableUpgradeable`, and `OwnableUpgradeable`. The contract has 12 public/external functions, including setter and getter functions. The main functions implemented in the contract are:

- `registerNodeOperator(...)`: This function registers a user as an operator to allow them to bid.
- `fetchNextKeyIndex(...)`: Fetches the next key available for a user. Only the `AuctionManager` contract is allowed to call this function.
- `addToWhitelist(...)`: Adds an address to the whitelist.
- `removeFromWhitelist(...)`: Removes an address from the whitelist.
- `setAuctionContractAddress(...)`: Sets the auction contract address for verification purposes.

4.7 StakingManager

The **StakingManager** manages the staking process for validators on the Beacon Chain. It implements the interfaces `IStakingManager` and `IBeaconUpgradeable`. The contract inherits from the following contracts:

- `Initializable`
- `PausableUpgradeable`
- `OwnableUpgradeable`
- `ReentrancyGuardUpgradeable`
- `UUPSUpgradeable`

This contract contains 21 public/external functions, including getter and setter functions. The setter functions allow only the owner to set the addresses for the contracts `LiquidityPool`, `BNFT`, `TNFT`, `EtherFiNode`, and the `Deposit`. The core functions in the contract are:

- `batchDepositWithBidIds`: Allows depositing multiple stakes at once.
- `batchRegisterValidators`: This function updates validator information (e.g., set status to LIVE), mints NFTs, and deposits into the beacon chain.
- `batchCancelDeposit`: Cancels users' deposits. The function unregisters the validators in the deposit phase, re-initiates the bid in the `AuctionManager` contract, and refunds the users.
- `disableWhitelist`: Allows only the owner to disable the bid whitelist.
- `enableWhitelist`: Allows only the owner to enable the bid whitelist.
- `verifyWhitelisted`: Verifies whether a user is whitelisted.

4.8 AuctionManager

The **AuctionManager** manages the bidding process for the right to run a validator node when ETH is deposited. The contract implements the interface `IAuctionManager`. The contract inherits from the following contracts:

- `Initializable`
- `PausableUpgradeable`
- `OwnableUpgradeable`
- `ReentrancyGuardUpgradeable`
- `UUPSUpgradeable`

The contract contains 19 public/external functions, including getter and setter functions. The setter functions allow only the owner to update the minimum bid price for non-whitelisted bidders, the minimum bid price for a whitelisted bidder, and the maximum bid price for both bidders. The core functions in the contract are:

- `createBid(...)`: This function creates bids for the right to run a validator node when ETH is deposited. Implements different checking steps, e.g., if the bid value is correct, if the whitelist is enabled, and if the user is whitelisted.
- `cancelBidBatch(...)`: Only the bidder (i.e., operator) can call this function to cancel multiple bids in a batch.
- `cancelBid(...)`: Only the bidder (i.e., operator) can call this function to cancel a specified bid by deactivating it.
- `updateSelectedBidInformation(...)`: Only the `StakingManager` contract can call this function to update the bid details used in a stake match.
- `reEnterAuction(...)`: Only the `StakingManager` contract can call this function to allow a bid that was matched to a canceled stake to re-enter the auction.
- `processAuctionFeeTransfer(...)`: Only the `StakingManager` contract can call this function to transfer the auction fee received from the node operator to the protocol revenue manager.
- `disableWhitelist(...)`: Only the owner can call it to disable the whitelisting phase of the bidding. It allows both regular users and whitelisted users to bid.
- `enableWhitelist(...)`: Only the owner can call it to enable the whitelisting phase of the bidding. Only users who are on a whitelist can bid.

4.9 ProtocolRevenueManager

The **ProtocolRevenueManager** manages the revenue distribution from an auction fee paid by a node operator for the corresponding validator. The contract implements the interface `IProtocolRevenueManager`. The contract inherits from the following contracts:

- `Initializable`
- `PausableUpgradeable`
- `OwnableUpgradeable`
- `ReentrancyGuardUpgradeable`
- `UUPSUpgradeable`

The contract contains 10 public/external functions, including getter and setter functions. The setter functions allow only the owner to instantiate the interfaces of the `EtherFiNodesManager` and `AuctionManager`. There are also setter functions to set the auction reward vesting period and the reward split for the stakers. The core functions in the contract are:

- `addAuctionRevenue(...)`: Adds the revenue from the auction fee paid by the node operator for the corresponding validator.
- `distributeAuctionRevenue(...)`: A function to distribute the accrued rewards to the validator.

4.10 EtherFiNodesManager

The **EtherFiNodesManager** manages the validator nodes for the EtherFi network. The contract implements the interface `IEtherFiNodesManager`. The contract inherits from the following contracts:

- `Initializable`
- `PausableUpgradeable`
- `OwnableUpgradeable`
- `ReentrancyGuardUpgradeable`
- `UUPSUpgradeable`

The contract contains 35 public/external functions, including getter and setter functions. Some setter functions allow only the owner to call them. They implement different functionality, e.g., set the staking rewards split, the protocol rewards split, and the non-exit penalty principal amount and daily rate amount. The contract also has setter functions that allow only the contract `StakingManager` calls, such as set the phase of the validator, set the ipfs hash of the validator's encrypted private key, and increment the number of validators. The `ProtocolRevenueManager` interacts with this contract as well to set the local revenue index for a specific node. The core functions in the contract are:

- `registerEtherFiNode(...)` and `unregisterEtherFiNode(...)`: The contract allows the `StakingManager` contract calls the functions to register the validator ID for the EtherFiNode contract.
- `unregisterEtherFiNode(...)`: The `StakingManager` also can call this function to unset the EtherFiNode contract for the validator ID.
- `sendExitRequest(...) and batchSendExitRequest(...)`: Only T-NFT holders can call these functions to send the request to exit the validator node.
- `processNodeExit(...)`: The protocol monitors the status of the validator nodes and marks them as `EXITED` by calling this function to process their exits. The function marks the node as `EXITED`, distributes the protocol rewards (auction), and stops sharing the protocol revenue.
- `partialWithdraw(...) and partialWithdrawBatch`: These functions allow for the partial withdrawal of rewards for a single validator or a list of validators, respectively. The partial withdrawal can only be processed if the EtherFi node's balance is lower than 8 ether and the node is not being slashed. The function retrieves the staking and protocol rewards and the vested auction fee revenue to distribute the payouts for the treasury, operator, and {T,B}-NFT holders.
- `partialWithdrawBatchGroupByOperator(...)`: This function allows operators to process the rewards skimming for the validator nodes that they possess. Similarly to the `partialWithdraw(...)`, the node's balance needs to be lower than 8 ether, the node can not be marked as `BEING_SLASHED`. The difference between them is that the rewards to be paid are moved to the `EtherFiNodesManager` contract before distributing the payouts. For each validator: a) the payment is applied to the respective {T,B}-NFT holders, and b) the operator and treasury receive the total rewards amount for all validator nodes in a single transaction.
- `fullWithdraw(...)` and `fullWithdrawBatch(...)`: These functions can only be called after the node is marked as `EXITED`. Then, the node is marked as `FULLY_WITHDRAWN` and distributes the payouts for the treasury, operator, and {T,B}-NFT holders.
- `markBeingSlahsed(...)`: This function is called by the owner to mark validators as being slashed.

4.11 MembershipNFT

The **MembershipNFT** contract is an `ERC1155Upgradeable` contract that mint and burn NFTs. The contract also inherits from the following contracts:

- `Initializable`
- `OwnableUpgradeable`
- `UUPSUpgradeable`

The contract contains 19 public/external functions, including getter and setter functions. The core functions in the contract are:

- `mint(...)` and `burn(...)`: Only the `MeETH` contract can mint and burn membership tokens.
- `loyaltyPointsOf(...)`: this function returns the total loyalty points of a particular token. It calculates the accrued loyalty points and sums with the current loyalty points.
- `tierPointsOf(...)`: This function is quite similar to the `loyaltyPointsOf(...)`. The difference relies on the tier points instead of the loyalty points.

4.12 WeETH

The **WeETH** contract is an ERC20 token that can be wrapped and unwrapped by users with eETH and weETH respectively. The contract also inherits from the following contracts:

- `ERC20Upgradeable`
- `ERC20PermitUpgradeable`
- `OwnableUpgradeable`
- `UUPSUpgradeable`

The contract contains six public/external functions. The core functions in the contract are `wrap(...)` and `unwrap(...)`. The function `wrap(...)` wraps an amount of eETH into WeETH, while `unwrap(...)` unwraps an amount of WeETH into eETH.

4.13 MeETH

The **MeETH** contract is an implementation of the ImeETH interface. The contract implements features for the wrapping and unwrapping users' ETH into membership NFTs. It also implements staking rewards so the users can sacrifice their rewards to earn more points. The contract also inherits from the following contracts:

- `Initializable`
- `OwnableUpgradeable`
- `UUPSUpgradeable`

The contract contains 28 public/external functions, including getter and setter functions. The core functions in the contract are:

- `wrapEthForEap(...)`: Allows EarlyAdopterPool users to re-deposit and mint membership NFT claiming their loyalty and tier points. The amount of ETH is then sent to the `LiquidityPool` on behalf of the `MeETH` contract.
- `wrapEth(...)`: It is a function that allows users to wrap their ETH into a membership NFT. Similarly to the `wrapEthForEap(...)`, the deposit amount of ETH is also sent to the `LiquidityPool` on behalf of the `MeETH` contract.
- `topUpDepositWithEth(...)`: Allows the token owner to increase their deposit tied to a specified token within a percentage limit. This function can only be called once per month for each NFT. The deposit amount of ETH is also sent to the `LiquidityPool` on behalf of the `MeETH` contract.
- `unwrapForEth(...)`: The token owner calls this function to unwrap their membership tokens and receive ETH in return. This function follows several steps, such as: it checks if the caller is the NFT owner, updates the token status by calculating the accrued membership loyalty and tier points, calculates the staking rewards for the token and update the token data and deposits information, and sends the amount of ether to the token's owner.
- `withdrawAndBurnForEth(...)`: The token's owner can withdraw the entire balance of the NFT and burn it.
- `stakeForPoints(...)` and `unstakeForPoints(...)`: Users can stake their staking rewards to earn membership points faster and unstake to undo the staking, respectively.
- `distributeStakingRewards(...)` Only the contract owner can call this function to distribute staking rewards to eligible NFTs based on their staked tokens and membership tiers.
- `withdrawFees(...)`: Only the contract owner can call this function to withdraw accumulated fees and send them to the `Treasury` and `ProtocolRevenueManager` contracts.
- `updateFeeRecipientsValues(...)`: Only the contract owner can update the fee recipient percentages, i.e., `ProtocolRevenueManager` and `Treasury` contracts.

4.14 LiquidityPool

The **LiquidityPool** contract manages the liquidity pool of eETH tokens and ETH. The contract allows for deposits of ETH, and will mint and send eETH to the sender in exchange. Withdrawals of eETH can be made, which burn the user's balance and send an equivalent amount of ETH back to the recipient. The contract also implements validators' management and operational tasks to manage liquidity. The contract inherits from the following contracts:

- `Initializable`
- `OwnableUpgradeable`
- `UUPSUpgradeable`

The contract contains 20 public/external functions, including getter and setter functions. The getter functions implement different features, e.g., convert the amount of shares into ETH and vice versa, and calculate the total ETH a user can claim. The core functions in the contract are:

- `deposit(...)`: A whitelisted user calls this function directly or through the `MeETH` contract to deposit ETH into the pool. Then, the function calculates the amount of shares to mint eETH. When this function is called from `MeETH`, this contract is the eETH owner.
- `withdraw(...)`: This function burns a specified eETH amount of the caller and sends an equivalent amount of ETH back to the recipient.
- `batchDepositWithBidIds(...)`: Only the owner can call this function. The owner deposits 2 ETH, which is combined with 30 ETH from the liquidity pool, and deposits 32 ETH into the `StakingManager` contract by calling the function `batchDepositWithBidIds(...)`.
- `batchRegisterValidators(...)`: Only the owner can call this function to register validators after `batchDepositWithBidIds(...)`.
- `processNodeExit(...)`: The owner removes exited nodes from the liquidity pool and calls the function `fullWithdrawBatch()` in `EtherFiNodesManager` to apply the full withdrawal process.
- `sendExitRequests(...)`: The owner calls this function to send exit requests as the T-NFT holder to the `sendExitRequest()` in `EtherFiNodesManager`.
- `rebase(...)`: The owner rebases the total value that is out of the liquidity pool by providing the total value locked in the liquidity pool.

4.15 Design Evaluation

Like any software program, smart contracts can suffer from design flaws that can lead to inefficiencies and security vulnerabilities. During the auditing, we detected several occurrences related to design, Circular Dependencies, and Lack of Cohesion in functions.

Circular dependencies occur when two or more smart contracts depend on each other in a way that creates a loop. This can increase the effort to debug and comprehend the code. In addition, when updating smart contracts, changes made to one contract can have unintended consequences on the others. We should carefully consider the dependencies between contracts and ensure no circular references. This can be achieved through modular design, where contracts are broken down into weakly coupled components that can be easily tested, comprehended, and updated.

Lack of Cohesion occurs when a smart contract contains functions that do not relate to the overall purpose of the contract. This can make contracts harder to understand and apply changes. It can also increase the risk of security vulnerabilities, as unnecessary functions may introduce unintended interactions with the contract's other functions. To maintain cohesion, we need to observe if each function within a contract has a clear and specific purpose and is related to the contract's overall purpose (e.g., to use state variables). This best practice is reported in **Issue 6.25** . Fig. 2 illustrates a structural diagram of the contracts audited in this report. The following circular dependencies can be visualized on the diagram:

1. `MembershipNFT` depends on `MeETH` and vice versa.
2. `EtherFiNodesManager` ⇒ `ProtocolRevenueManager` ⇒ `AuctionManager` ⇒ `ProtocolRevenueManager` ⇒ `EtherFiNodesManager`.
3. `LiquidityPool` ⇒ `MeETH` ⇒ `MembershipNFT` ⇒ `LiquidityPool`

It is worth noting that the dependence described in item **2)** also exists between the contracts directly in smaller loops. For example, `ProtocolRevenueManager` depends on `AuctionManager` and vice versa.

Remarks

We highlight the presence of circular dependencies, which substantially increase the code complexity (dependencies among the mentioned smart contracts above). The best solution to mitigate this is to remove these detected loops. If this is not possible for some reason, we recommend auditing the involved contracts after any applied change because vulnerabilities can emerge due to the high coupling of the contracts.

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] The LiquidityPool.withdraw(...) function can by called for any address

File(s): LiquidityPool.sol

Description: The LiquidityPool.withdraw(...) has parameter _recipient, which is not checked and can be any address. This allows malicious users to call withdraw(...) for any EETH holder. This is especially dangerous for MeETH holders since the malicious user may call withdraw(...) with the MeETH token address - this will result in burning EETH and sending ethers to the MeETH token contract. In effect, the funds would be locked in the MeETH contract, and the holder of the MeETH token won't be able to exchange their MeETH for EETH or ETH. We present the function below:

```
function withdraw(address _recipient, uint256 _amount) public whenLiquidStakingOpen {
    require(address(this).balance >= _amount, "Not enough ETH in the liquidity pool");
    require(eETH.balanceOf(_recipient) >= _amount, "Not enough eETH");

    uint256 share = sharesForAmount(_amount);
    eETH.burnShares(_recipient, share);

    (bool sent, ) = _recipient.call{value: _amount}("");
    require(sent, "Failed to send Ether");

    emit Withdraw(_recipient, _amount);
}
```

Recommendation(s): During direct calls to LiquidityPool.withdraw(...) check if msg.sender == _recipient.

Status: Unresolved

Update from the client:

6.2 [High] Deposit may be front-run with arbitrary withdrawal credentials

File(s): StakingManager.sol

Description: Delegated staking protocols may be exposed to a known vulnerability, where the malicious operator may front-run staker's deposit call to deposit_contract and provide different withdrawal credentials. This issue also exists in the EtherFi protocol. The registration of the validator is done with the function `_registerValidator(...)`, called by the staker. In this function call, the funds are deposited to the `deposit_contract` with the additional data, and the encrypted validator keys are shared with the operator. We present the function below:

```
function _registerValidator(
    uint256 _validatorId, address _bNftRecipient, address _tNftRecipient, DepositData calldata _depositData
) internal {
    require(nodesManager.phase(_validatorId) == IEtherFiNode.VALIDATOR_PHASE.STAKE_DEPOSITED, "Incorrect phase");
    require(bidIdToStaker[_validatorId] == msg.sender, "Not deposit owner");

    // Deposit to the Beacon Chain
    bytes memory withdrawalCredentials = nodesManager.getWithdrawalCredentials(_validatorId); // @audit Operator may decipher and front-run
    depositContractEth2.deposit{value: stakeAmount}(_depositData.publicKey, withdrawalCredentials, _depositData.signature, _depositData.depositDataRoot);

    nodesManager.incrementNumberOfValidators(1);
    nodesManager.setEtherFiNodePhase(_validatorId, IEtherFiNode.VALIDATOR_PHASE.LIVE);
    nodesManager.setEtherFiNodeIpfsHashForEncryptedValidatorKey(_validatorId, _depositData.ipfsHashForEncryptedValidatorKey);

    // Let valiadatorId = nftTokenId
    uint256 nftTokenId = _validatorId;
    TNFTInterfaceInstance.mint(_tNftRecipient, nftTokenId);
    BNFTInterfaceInstance.mint(_bNftRecipient, nftTokenId);

    auctionManager.processAuctionFeeTransfer(_validatorId);

    emit ValidatorRegistered(
        auctionManager.getBidOwner(_validatorId),
        _bNftRecipient,
        _tNftRecipient,
        _validatorId,
        _depositData.publicKey,
        _depositData.ipfsHashForEncryptedValidatorKey
    );
}
```

The operator, observing the `mempool`, may extract the validation key, create a new payload, and front-run staker by depositing `1 ETH` to `deposit_contract` with chosen withdrawal credentials. The stakers deposit would be successfully processed, but the withdrawal credentials provided by the operator would not be overwritten. Moreover, in the EtherFi protocol, the staker may act maliciously and provide other addresses in withdrawal credentials. There is an incentive for that since the staking funds (rewards and principal) are shared among the B/T-NFT holders and the operator. The staker may deposit `1 ETH` to `deposit_contract` and provide chosen withdrawal credentials before calling `_registerValidator(...)`. That way, the operator and treasury contract would not receive the rewards, and the T-NFT holder would not receive rewards and principal. Note that depositing staking funds may be secured from front-running by providing `_depositRoot` on the `registerValidator(...)` call. The `_depositRoot` can be fetched from the `deposit_contract`. The `verifyDepositState(...)` modifier compares provided `_depositRoot` with the current root fetched by calling `deposit_contract.get_deposit_root(...)`. This would mitigate the scenario described above since the operator would change the root by front-running their deposit. However, we still believe the issue exists since the check may be omitted by passing the value of `_depositRoot` as `0x0`.

Recommendation(s): The incentives for the staker may not be worth exploiting this issue since the operator only gets rewards after some time spent on maintaining the process, and the T-NFT holder can check if withdrawal credentials are correct. Credentials may be checked by fetching data as described here. This can be resolved on the frontend. If the credentials are wrong, T-NFT doesn't buy the token, and the operator stops maintaining the process. If we assume that the staker won't act maliciously because of the lack of a strong incentive, then this issue may be resolved by splitting the logic of the `_registerValidator(...)` function. Sharing validation key may be moved to a separate function and be called by the staker after the `_registerValidator(...)` function is called. This way, the operator would receive the key only after the funds are deposited to the `deposit_contract` and the proper withdrawal credentials are set. Consider also informing stakers that it is a best practice to call `registerValidator(...)` with the latest `_depositRoot` fetched from the `deposit_contract` to secure the deposit from front-running.

Status: Unresolved

Update from the client:

6.3 [High] Owner can withdraw users' assets as fees

File(s): `MeETH.sol`

Description: The function `withdrawFees()` allows the owner withdraws treasury and protocol revenue fees, as presented below:

```
    function withdrawFees() external onlyOwner {
        uint256 totalAccumulatedFeesBefore = totalFeesAccumulated;

        uint256 treasuryFees = totalAccumulatedFeesBefore * treasuryFeePercentage / 100;
        uint256 protocolRevenueFees = totalAccumulatedFeesBefore * protocolRevenueFeePercentage / 100;

        // @audit totalFeesAccumulated is never reset after owner withdraws
        totalAccumulatedFeesBefore = 0;

        eETH.transfer(treasury, treasuryFees);
        eETH.transfer(protocolRevenueManager, protocolRevenueFees);
    }
```

The state variable `totalFeesAccumulated` is always incremented when a new meETH NFT is minted (mintFee) or burned (burnFee). When the owner calls `withdrawFees()`, the `totalFeesAccumulated` should be set to `0`. As we can see in the function above, the local variable `totalAccumulatedFeesBefore` is set to zero instead. In effect, the owner will withdraw users' assets.

Recommendation(s): Set `totalFeesAccumulated` to `0`.

Status: Unresolved

Update from the client:

6.4 [High] Protocol rewards may be claimed after the node exit

File(s): `ProtocolRevenueManager.sol`

Description: Protocol participants (`B-NFT`/ `T-NFT` holders, operator) may collect protocol fees by calling `EtherFiNodesManager.partialWithdraw(...)`. This function checks if the balance of the `EtherFiNode` is less than `8 ETH` and if the `EtherFiNode` is not in the `BEING_SLASHED` phase. Next, the fee is computed, and the local node index is updated in the ProtocolRevenueManager with the function `distributeAuctionRevenue(...)`. The local index of a node is updated if the node is active and the fee amount is returned. Otherwise, the amount `0` should be returned if the node exited. We present the `distributeAuctionRevenue(...)` function below:

```
function distributeAuctionRevenue(uint256 _validatorId) external onlyEtherFiNodesManager nonReentrant returns (uint256)
 ↪ {
    if (etherFiNodesManager.isExited(_validatorId)) {
        return 0;
    }
    uint256 amount = getAccruedAuctionRevenueRewards(_validatorId);
    etherFiNodesManager.setEtherFiNodeLocalRevenueIndex{value: amount}(_validatorId, globalRevenueIndex);
    return amount;
}
```

However, with the current implementation, after the node is exited and the funds are withdrawn, the node enters the `FULLY_WITHDRAWN` phase. This allows protocol participants to claim protocol rewards after they stop participating in the staking process, therefore stealing rewards from other users. Note that the first reward after exiting will be `0` since the local index is set to `0` while exiting the node, but each next reward would be a valid amount.

Recommendation(s): Consider checking in the `ProtocolRevenueManager.distributeAuctionRevenue(...)` if the node is in the `FULLY_WITHDRAWN` phase.

Status: Unresolved

Update from the client:

6.5 [High] Staked funds are frozen after slashing

File(s): EtherFiNodesManager.sol

Description: The validator may be slashed during staking. In EtherFi protocol, slashing is marked in the EtherFiNode contract instance by setting the phase to BEING_SLASHED. We present the function responsible for this below:

```
function markBeingSlahsed(uint256[] calldata _validatorIds) external whenNotPaused onlyOwner {
    for (uint256 i = 0; i < _validatorIds.length; i++) {
        address etherfiNode = etherfiNodeAddress[_validatorIds[i]];
        IEtherFiNode(etherfiNode).markBeingSlahsed();
    }
}
```

After the penalty is processed on the Beacon chain, staked funds are withdrawn to the EtherFiNode, where users may perform a full withdrawal. However, to fully withdraw funds from the EtherFiNode contract, it must be in the EXITED phase, as presented below:

```
function fullWithdraw(uint256 _validatorId) public nonReentrant whenNotPaused{
    address etherfiNode = etherfiNodeAddress[_validatorId];
    require(
        IEtherFiNode(etherfiNode).phase() == IEtherFiNode.VALIDATOR_PHASE.EXITED,
        "validator node is not exited"
    );

    (uint256 toOperator, uint256 toTnft, uint256 toBnft, uint256 toTreasury)
        = getFullWithdrawalPayouts(_validatorId);
    IEtherFiNode(etherfiNode).processVestedAuctionFeeWithdrawal();
    IEtherFiNode(etherfiNode).markFullyWithdrawn();

    _distributePayouts(_validatorId, toTreasury, toOperator, toTnft, toBnft);
}
```

The EXITED phase can only be triggered by the function _processNodeExit(...), which requires the node to be in the LIVE phase. Since the node is marked as BEING_SLASHED in the occurrence of slashing, the node can't enter the EXITED phase. In effect, the funds are locked in EtherFiNode and can't be distributed to the protocol participants.

Recommendation(s): Consider introducing function that transits node's phase from BEING_SLASHED to EXITED.

Status: Unresolved

Update from the client:

6.6 [High] Treasury and ProtocolRevenueManager owner can not withdraw eETH tokens

File(s): MeETH.sol, Treasury.sol, ProtocolRevenueManager.sol

Description: The function withdrawFees() calculates the treasuryFees to transfer the amount of eETH to the treasury contract. The function is presented below:

```
function withdrawFees() external onlyOwner {
    uint256 totalAccumulatedFeesBefore = totalFeesAccumulated;

    uint256 treasuryFees = totalAccumulatedFeesBefore * treasuryFeePercentage / 100;
    uint256 protocolRevenueFees = totalAccumulatedFeesBefore * protocolRevenueFeePercentage / 100;

    totalAccumulatedFeesBefore = 0;

    // @audit treasury receives the treasuryFees amount
    eETH.transfer(treasury, treasuryFees);
    eETH.transfer(protocolRevenueManager, protocolRevenueFees);
}
```

However, the Treasury contract does not implement a function to call the withdraw(...) in the LiquidityPool for receiving an equal amount of ETH or another mechanism, e.g., transfer to another address. As we can observe in the LiquidityPool.withdraw(...), the msg.sender should be the Treasury. In addition, the same issue occurs for ProtocolRevenueManager.

```
function withdraw(address _recipient, uint256 _amount) public whenLiquidStakingOpen {
    require(address(this).balance >= _amount, "Not enough ETH in the liquidity pool");
    require(eETH.balanceOf(msg.sender) >= _amount, "Not enough eETH");

    uint256 share = sharesForAmount(_amount);
    eETH.burnShares(msg.sender, share);
    ...
}
```

Recommendation(s): Ensure `Treasury` and `ProtocolRevenueManager` owners can withdraw `eETH` tokens.

Status: Unresolved

Update from the client:

6.7 [Medium] Centralization risks

File(s): `src/`

Description: The EtherFi protocol's current architecture leans toward centralization, with crucial procedures depending on the owner's accurate function execution. In some instances, the owner can set arbitrary values, like points in membership contracts. While this structure can streamline operations, it also introduces a dependency on the owner's actions and decisions. If owner integration is incorrect by mistake or the power is used maliciously due to ownership being taken by a malicious agent, users can be harmed. Below we list examples of functionalities controlled by the owner that could heavily affect the correct behavior of the protocol:

- `processNodeExit(...)`: Its execution timing is vital to maintaining user funds' security;
- The rebasing mechanism in `LiquidityPool`;
- Determining reward split values and non-exit penalty principals in `EtherFiNodesManager`;
- Setting fee amounts in MeETH;
- Modifying contract addresses post-initialization;
- Upgrading the contracts;

Recommendation(s): To diversify the decision-making process and reduce potential risks, it might be worth considering using a multi-signature wallet with a timelock implementation for `onlyOwner` function calls. This could add a consensus layer to significant actions and allow users to react against certain changes. Adding input checks for acceptable value ranges could be a beneficial safeguard for functions that set specific values. Alternatively, if certain functions are not essential, reviewing them for potential removal or adjustment might be prudent.

Status: Unresolved

Update from the client:

6.8 [Medium] Check in LiquidityPool.deposit(...) is incorrect

File(s): `LiquidityPool.sol`

Description: The `deposit(...)` function in LiquidityPool does incorrect check of the `_user` parameter. We present the function below.

```
function deposit(address _user, address _recipient, bytes32[] calldata _merkleProof) public payable
 ↪ whenLiquidStakingOpen {
    stakingManager.verifyWhitelisted(_user, _merkleProof);
    require(regulationsManager.isEligible(regulationsManager.whitelistVersion(), _user), "User is not whitelisted");
    require(_recipient == msg.sender || _recipient == address(meETH), "Wrong Recipient");

    uint256 share = _sharesForDepositAmount(msg.value);
    if (share == 0) {
        share = msg.value;
    }
    eETH.mintShares(_recipient, share);

    emit Deposit(_recipient, msg.value);
}
```

The `_user` parameter is checked if it's whitelisted and eligible. The function `deposit(...)` may be called from the MeETH contract, where `_user` is defined as a `msg.sender`. However, if the function `deposit(...)` is called directly, the `_user` address may be any address that is already whitelisted. Therefore, the checks on the `_user` address in the second case are ineffective since the `_user` parameter is not used in the function for any other purpose than verification. Any non-whitelisted user may deposit funds to the `LiquidityPool`.

Recommendation(s): Consider checking if `msg.sender` is whitelisted when the `deposit(...)` function is called directly.

Status: Unresolved

Update from the client:

6.9 [Medium] Incorrect calculation in `convertEapPoints(...)`

File(s): `MeETH.sol`

Description: The function `convertEapPoints(...)` is used for migration funds and points from the early adopter pool to MeETH. However, converting EAP points to loyalty points differs between documentation and implementation. Provided documentation states that conversion is:

```
LoyaltyPoints = EapScores * 1e5 / 1000 / 1 days
```

Calculation in `convertEapPoints(...)` is:

```
uint256 loyaltyPoints = _min(1e5 * _eapPoints / 1 days , type(uint40).max);
```

Implemented calculation lacks division by `1000`. Therefore users migrated from EAP would have 1000x more loyalty points than defined in the documentation.

Recommendation(s): Consider unifying documentation and implementation of converting EAP points to loyalty points.

Status: Unresolved

Update from the client:

6.10 [Medium] Marking the node as EXITED before receiving funds may lead to locked funds

File(s): `EtherFiNodesManager.sol`

Description: The EtherFiNode is marked `EXITED` with the `processNodeExit(...)` function. This function can be called only by the EtherFi protocol team. However, the timing of calling this function is crucial. The natspec documentation of the function says, "Once the node's exit is observed, the protocol calls this function to process their exits.". That means the node is marked `EXITED` when the node's exit is observed, but the funds are not yet transferred to the EtherFiNode, since there is a substantial amount of time between staking exit and transferring funds from the beacon chain. This creates an attack vector. A malicious user may call `fullWithdraw(...)` right after the node is marked `EXITED`, which would mark node `FULLY_WITHDRAWN`. Later, when the funds staked funds reach the EtherFiNode, they won't be withdrawable since the node is in the `FULLY_WITHDRAWN` phase.

Recommendation(s): Make sure to call `processNodeExit(...)` only after the staked funds reach EtherFiNode contract.

Status: Unresolved

Update from the client:

6.11 [Medium] Points diluting mechanism does not disincentivize users from `topUp` with huge amounts

File(s): `MeETH.sol`

Description: The `topUpDepositWithEth(...)` function allows users to increase the staked `ETH` amount in a `MembershipNFT` token. The function contains a mechanism to penalize users and decrease their tier points if they add a large amount of `ETH`, a preventive measure to discourage users from gaming the system by initially staking a low amount of `ETH`, climbing the tier ladder, and then topping up with a larger amount. This mechanism can be seen in the next snippet of code.

```
function _topUpDeposit(uint256 _tokenId, uint128 _amount, uint128 _amountForPoints) internal {
        ...
        uint256 maxDepositWithoutPenalty = (totalDeposit * maxDepositTopUpPercent) / 100;

        _deposit(_tokenId, _amount, _amountForPoints);
        token.prevTopUpTimestamp = uint32(block.timestamp);

        // proportionally dilute tier points if over deposit threshold & update the tier
        if (msg.value > maxDepositWithoutPenalty) {
            // @audit - The new amount of points does not decrease by increasing `msg.value`
            uint256 dilutedPoints = (msg.value * token.baseTierPoints) / (msg.value + totalDeposit);
            token.baseTierPoints = uint40(dilutedPoints);
            _claimTier(_tokenId);
        }
    }
```

However, the dilution formula used seems to be ineffective. When the deposit surpasses the defined threshold, the resulting tier points after dilution might not decrease as expected. We can rewrite the used expression as

$$\frac{msg.value}{msg.value + totalDeposit} * baseTierPoints$$

when `msg.value` is much bigger than `totalDeposit`, the left factor is close to one, causing the result to be close to `baseTierPoints`.

This cause that for larger deposits, the new tier points could be close to the original amount, effectively incentivizing users to top up with larger amounts if they decide to bypass the threshold.

Recommendation(s): Consider revising the tier point dilution formula to ensure it functions as intended. The new formula should effectively penalize users who top up substantially after surpassing the threshold.

Status: Unresolved

Update from the client:

6.12 [Medium] The B-NFT holder is unfairly penalized when the T-NFT holder requests to exit the node

File(s): `EtherFiNode.sol`

Description: The node exit process may be requested by T-NFT holder, which signals B-NFT holder to exit the node. During the time between the exit request and the exit execution, the penalty for B-NFT holder is increasing. However, with the current setup, the B-NFT holder cannot avoid a penalty even if exits the validator immediately. The function `processNodeExit(...)`, which marks the exit timestamp, will be called only after the staked ETH is transferred to EtherfFiNode, which is at least 27-36 hrs after the validator exits.

Recommendation(s): Consider passing to the `processNodeExit(...)` the timestamp of the exit request by B-NFT holder/operator on the beacon chain.

Status: Unresolved

Update from the client:

6.13 [Medium] Users can get more value than what they should when withdrawing from the pool

File(s): `LiquidityPool.sol`

Description: The `withdraw(...)` function in the `LiquidityPool` contract allows users to withdraw their `ETH` by specifying the amount they wish to receive. The function then calculates the equivalent number of shares that should be burned.

```
function withdraw(address _recipient, uint256 _amount) public whenLiquidStakingOpen {
    require(address(this).balance >= _amount, "Not enough ETH in the liquidity pool");
    require(eETH.balanceOf(msg.sender) >= _amount, "Not enough eETH");

    // @audit - This computation rounds down the number of shares
    uint256 share = sharesForAmount(_amount);
    eETH.burnShares(msg.sender, share);

    (bool sent, ) = _recipient.call{value: _amount}("");
    require(sent, "Failed to send Ether");

    emit Withdraw(msg.sender, _recipient, _amount);
}
```

However, users may burn less than they should. The function `sharesForAmount(...)` computes the share value and rounds down the result. Consequently, a user can withdraw more `ETH` than the value of the shares they are burning. The discrepancy can range from negligible to substantial amounts depending on the pool conditions, which can adversely affect the remaining value for other users in the pool.

Recommendation(s): Consider rounding up the value when calculating the number of shares to be burned. Alternatively, the function could be modified to require users to specify the number of shares they wish to burn rather than the amount of `ETH` they want to receive.

Status: Unresolved

Update from the client:

6.14 [Medium] _processNodeExit(...) never distributes the protocol reward to the validator's node

File(s): `EtherFiNodesManager.sol`

Description: The function `_processNodeExit(...)` is called by the protocol to mark the node as `EXITED`, distribute the protocol revenue, and stop sharing the protocol revenue. The function is described below:

```
function _processNodeExit(uint256 _validatorId, uint32 _exitTimestamp) internal {
  address etherfiNode = etherfiNodeAddress[_validatorId];

  // @audit Making EXITED before calling distributeAuctionRevenue results in amount == 0
  // Mark EXITED
  IEtherFiNode(etherfiNode).markExited(_exitTimestamp);

  // distribute the protocol reward from the ProtocolRevenueMgr contract to the validator's etherfi node contract
  uint256 amount = protocolRevenueManager.distributeAuctionRevenue(_validatorId);

  // Reset its local revenue index to 0, which indicates that no accrued protocol revenue exists
  IEtherFiNode(etherfiNode).setLocalRevenueIndex(0);
  ...
}
```

However, the node is marked as exited before distributing the accrued rewards to the validator. As shown below in `distributeAuctionRevenue(...)`, the function will return `0`.

```
function distributeAuctionRevenue(uint256 _validatorId) external onlyEtherFiNodesManager nonReentrant returns (uint256)
↪ {
    // @audit The node is set as EXITED and then the amount is 0
    if (etherFiNodesManager.isExited(_validatorId)) {
        return 0;
    }
    uint256 amount = getAccruedAuctionRevenueRewards(_validatorId);
    etherFiNodesManager.setEtherFiNodeLocalRevenueIndex{value: amount}(_validatorId, globalRevenueIndex);
    return amount;
}
```

Recommendation(s): Ensure the node is marked `EXITED` only after calculating the protocol reward amount. For example:

```
-IEtherFiNode(etherfiNode).markExited(_exitTimestamp);
-uint256 amount = protocolRevenueManager.distributeAuctionRevenue(_validatorId);
+uint256 amount = protocolRevenueManager.distributeAuctionRevenue(_validatorId);
+IEtherFiNode(etherfiNode).markExited(_exitTimestamp);
```

Status: Unresolved

Update from the client:

6.15 [Low] BurnFee never initialized or set

File(s): `MeETH.sol`

Description: `burnFee` state variable is never initialized or there is no setter function for it. The value will always be equal to the default value. `WithdrawAndBurnForEth()` adds burnFee to the `totalFeesAccumulated` and withdraws the `totalBalance - burnFee`.

Recommendation(s): Initialize burnFee or set it in `setFeeAmounts()`. Update the event `UpdatedFees()` and add updated burnFee value.

Status: Unresolved

Update from the client:

6.16 [Low] Function markBeingSlahsed(...) does not check node phase

File(s): `EtherFiNodesManager.sol`

Description: The function `markBeingSlahsed(...)` is called by the owner of the EtherFiNodesManager contract and is responsible for transitioning the node into the `BEING_SLASHED` state. Even though this function can only be called by the owner, the phase of a node should be checked to avoid possible mistakes. Transitioning to `BEING_SLASHED` in the phase before `STAKE_DEPOSITED` may lead to unwanted behavior, like a lock of funds.

Recommendation(s): Consider checking if a node is in the `LIVE` state in function `markBeingSlahsed(...)`.

Status: Unresolved

Update from the client:

6.17 [Low] Incorrect deposit address

File(s): `StakingManager.sol`

Description: The address under `depositContractEth2`, `0xff50ed3d0ec03aC01D4C79aAd74928BFF48a7b2b` is not the address of a `deposit_contract`.

Recommendation(s): After discussion with the EtherFi team, the contract will be initialized with the incorrect address, but then the correct address will be set with a separate transaction. We still strongly advise initializing the contract with the correct address of `deposit_contract`: `0x00000000219ab540356cBB839Cbe05303d7705Fa`.

Status: Unresolved

Update from the client:

6.18 [Low] Partial withdrawal differs from batch partial withdrawal

File(s): `EtherFiNodesManager.sol`

Description: Users may withdraw the rewards from EtherFiNode by executing a single partial withdrawal or batch partial withdrawal. However, the batch partial withdrawal differs from the single one. In a single partial withdrawal, the mechanism in `EtherFiNode.withdrawFunds(...)` to send funds is secured from DOS - if the recipient does not accept the ETH, the funds are sent to the treasury. In the batch partial withdrawal, executed in `EtherfFiNodesManager.partialWithdrawBatchGroupByOperator(...)`, the participants can refuse to accept funds making the function revert. We present the part of `partialWithdrawBatchGroupByOperator(...)` function below:

```
    function partialWithdrawBatchGroupByOperator(
        address _operator,
        uint256[] memory _validatorIds,
        bool _stakingRewards,
        bool _protocolRewards,
        bool _vestedAuctionFee
    ) external nonReentrant whenNotPaused{
        /////////////////////////////////////////
        // Omitted part of the code
        /////////////////////////////////////////

        bool sent;
        tnftHolder = tnft.ownerOf(_validatorId);
        bnftHolder = bnft.ownerOf(_validatorId);
        if (tnftHolder == bnftHolder) {
            (sent, ) = payable(tnftHolder).call{value: toTnft + toBnft}("");
            require(sent, "Failed to send Ether");
        } else {
            (sent, ) = payable(tnftHolder).call{value: toTnft}("");
            require(sent, "Failed to send Ether");
            (sent, ) = payable(bnftHolder).call{value: toBnft}("");
            require(sent, "Failed to send Ether");
        }
        totalOperatorAmount += toOperator;
        totalTreasuryAmount += toTreasury;
        }
        (bool sent, ) = payable(_operator).call{value: totalOperatorAmount}("");
        require(sent, "Failed to send Ether");
        (sent, ) = payable(treasuryContract).call{value: totalTreasuryAmount}("");
        require(sent, "Failed to send Ether");
}
```

Recommendation(s): Consider unifying the batch and single partial withdrawal mechanism.

Status: Unresolved

Update from the client:

6.19 [Low] Potential price manipulation by the first depositor in `LiquidityPool`

File(s): `LiquidityPool.sol`

Description: The contract's design allows for the possibility of price manipulation by the first depositor in the `LiquidityPool`. By making a small initial deposit and subsequently inflating the `ETH` balance of the contract, the first depositor can manipulate the price of shares. Although direct `ETH` transfers to the contract are prevented by the `receive(...)` function, it remains possible to inflate the contract's balance through alternative methods, such as `selfdestruct` or block rewards. This vulnerability could be used to attack legitimate first depositors or establish an artificially high share price that could enable precision error-related attacks.

Recommendation(s): To mitigate this vulnerability, consider implementing a minimum amount of shares that can be minted when the total shares are zero. This can be achieved by setting a minimum initial deposit amount or by minting a fixed number of shares for the initial deposit.

Status: Unresolved

Update from the client:

6.20 [Low] The operator is incentivized to delay exit

File(s): `EtherFiNode.sol`

Description: The exit can be requested by the TNFT holder. Both the BNFT holder and operator are incentivized to process exit. The BNFT holder loses their stake proportionally to the exit delay. Conversely, the operator is incentivized by accruing the funds taken from BNFT as a penalty. Below we present part of the `getFullWithdrawalPayouts(...)` function, which is responsible for calculating the penalty:

```
function getFullWithdrawalPayouts(
    IEtherFiNodesManager.RewardsSplit memory _splits,
    uint256 _scale,
    uint128 _principal,
    uint64 _dailyPenalty
)
    external
    view
    returns (
        uint256 toNodeOperator,
        uint256 toTnft,
        uint256 toBnft,
        uint256 toTreasury
    )
{
   ////////////////
   // HIDDEN CODE //
   ////////////////

    // Deduct the NonExitPenalty from the payout to the B-NFT
    uint256 bnftNonExitPenalty = getNonExitPenalty(
        _principal,
        _dailyPenalty,
        exitTimestamp
    );

    if (payouts[2] > bnftNonExitPenalty) {
        payouts[2] -= bnftNonExitPenalty;

        // While the NonExitPenalty keeps growing till 1 ether,
        //  the incentive to the node operator stops growing at 0.5 ether
        //  the rest goes to the treasury
        if (bnftNonExitPenalty > 0.5 ether) {
            payouts[0] += 0.5 ether;
            payouts[3] += bnftNonExitPenalty - 0.5 ether;
        } else {
            payouts[0] += bnftNonExitPenalty;
        }
    } else {
        // If the B-NFT lost the whole principal, incentivize the node operator to exit the node.
        payouts[0] += payouts[2];
        payouts[2] = 0;
    }

   ////////////////
   // HIDDEN CODE //
   ////////////////

}
```

Additionally, to presented logic, the penalty for the operator is applied in the `getStakingRewardsPayouts(...)`. Below we present the part responsible only for removing staking rewards from the operator:

```
if (daysPassedSinceExitRequest >= 14) {
    treasury += operator;
    operator = 0;
}
```

Both mechanisms may incentivize the operator to delay exit or even act maliciously under certain conditions. While those conditions are very unlikely, they are still possible. Below, we list the cases where the operator may delay exit for their profit. All the cases assume the BNFT holder doesn't exit for a long period (which is very unlikely), so the operator may exit with delay.

- 1: The operator delays exit for <14 days. This allows for getting part of the rewards of BNFT and protects the operator from losing staking rewards. If the operator didn't exit for 13 days, they would earn `0.3269729098`. We can predict that operator will always delay exit for 13 days since it is a great profit;
- 2: The operator delays for the amount of days until the penalty for BNFT reach `0.5 ETH`. This amount is accrued after 23 days of delay. The operator would lose their staking rewards since the delay reached 14 days. However, the operator may still be profitable since the difference between 13 days of delay (`0.3269729098`) and 23 days of delay (`0.5036935857`) is `0.1767206759`, which is still more than rewards for the operator from 2 years of staking (assuming 5% APY). Therefore if the operator stakes for 2 years or less, they could delay exit for up to 23 days. Note that a longer delay does not make sense since everything above `0.5 ETH` goes to the treasury;
- 3: If the penalty for the BNFT holder is greater than the payout, all penalty is paid to the operator. However, the penalty may be at maximum `1 ETH` after 1 year of delay. That means that the penalty may only be greater from payout if the validator gets slashed `>16 ETH` (+ rewards). Therefore, the operator may benefit from delaying exit for a 1 year and then act maliciously to be slashed more than `16 ETH`. The benefit for the operator may be lucrative since if the payout for BNFT would be around `0.8 ETH`, that is equivalent to staking rewards for the operator after 10 years of work (assuming 5% APY);

Recommendation(s): Presented scenarios, even if very unlikely, are still coded and may possibly happen. We think that incentivization BNFT holder by penalties is a good approach. Since the operator doesn't put any funds upfront, it is a good idea to incentive them with additional rewards. However, consider capping the profit of the operator. Case `3`, described above, incentivizes the operator to delay exit for 1 year and act maliciously. We believe that such an incentive is unnecessary and possibly dangerous. Also, the maximum incentive for the operator could be lower than `0.5 ETH` since it is equivalent to 6 years of staking (5% APY).

Status: Unresolved

Update from the client:

6.21 [Info] Inconsistent access control in the `getFullWithdrawalPayouts(...)` function

File(s): `EtherFiNode.sol`

Description: The `EtherFiNode` contract implements the external function `getFullWithdrawalPayouts(...)` without using the `onlyEtherFiNodeManagerContract` modifier, i.e., it should accept anyone calling it. However, this function calls `getRewardsPayouts(...)` that implements `onlyEtherFiNodeManagerContract`. If this function is invoked by an address different from the `EtherFiNodeManager` contract, it reverts. The functions are presented below:

```
contract EtherFiNode is IEtherFiNode {
    ...
    function getFullWithdrawalPayouts(
        IEtherFiNodesManager.RewardsSplit memory _splits,
        uint256 _scale,
        uint128 _principal,
        uint64 _dailyPenalty
    )
        /////////////////////////////////////////////////////////////////////////////////
        // @audit This function does not implement the onlyEtherFiNodeManagerContract
        //        modifier and calls the function getRewardsPayouts
        /////////////////////////////////////////////////////////////////////////////////
        external
        view
        returns (uint256 toNodeOperator, uint256 toTnft, uint256 toBnft, uint256 toTreasury)
    {
        ...
        if (balance > 32 ether) {
            (
                payouts[0],
                payouts[1],
                payouts[2],
                payouts[3]
            ) = getRewardsPayouts(
                true,
                false,
                true,
                _splits,
                _scale,
                _splits,
                _scale
            );
        ...
}

function getRewardsPayouts(
        bool _stakingRewards,
        bool _protocolRewards,
        bool _vestedAuctionFee,
        IEtherFiNodesManager.RewardsSplit memory _SRsplits,
        uint256 _SRscale,
        IEtherFiNodesManager.RewardsSplit memory _PRsplits,
        uint256 _PRscale
    )
        public
        view
        onlyEtherFiNodeManagerContract
        returns (uint256, uint256, uint256, uint256)
    {
    ...
    }
}
```

Recommendation(s): In case only `EtherFiNodeManager` contract should be able to call `getFullWithdrawalPayouts(...)` function, consider adding the `onlyEtherFiNodeManagerContract(...)` modifier. Otherwise, an extract method refactoring should be applied on `getRewardsPayouts(...)` to allow both functions to work properly as designed.

Status: Unresolved

Update from the client:

6.22 [Info] Check if `returnAmount` is not zero

File(s): `LiquidityPool.sol`

Description: The function `batchDepositWithBidIds(...)` calculates `returnAmount` to update the state variable `totalValueOutOfLp` and send back to the `msg.sender` the non-used amount. When `returnAmount` is equal to `zero`, the writing operation and the call to send `0 ether` will still cost gas.

```
function batchDepositWithBidIds(
    uint256 _numDeposits,
    uint256[] calldata _candidateBidIds,
    bytes32[] calldata _merkleProof
    ) payable public onlyOwner returns (uint256[] memory) {
    ...
    uint256 returnAmount = 2 ether * (_numDeposits - newValidators.length);
    // @audit check if returnAmount > 0
    totalValueOutOfLp += returnAmount;
    (bool sent, ) = address(msg.sender).call{value: returnAmount}("");
    require(sent, "Failed to send Ether");

    return newValidators;
}
```

Recommendation(s): Consider checking if `returnAmount` is not `zero`.

Status: Unresolved

Update from the client:

6.23 [Info] EETH approval race condition

File(s): `EETH.sol`

Description: The EETH token contract allows for shares approval. However, the `approve(...)` function is vulnerable to a known attack vector - the user may use both the old and the new allowance by unfortunate transaction ordering. This is possible because `approve(...)` sets new approval. Therefore the user may observe the mempool, front-run approve transaction, and spend old approval. The user may spend this new approval as well after the new approval is set.

Recommendation(s): Consider mitigating this issue by introducing functions `increaseAllowance(...)` and `decreaseAllowance(...)`, which subtract/add allowance from the current one instead of setting a new one.

Status: Unresolved

Update from the client:

6.24 [Info] Misleading error message

File(s): `WeETH.sol`

Description: In the `unwrap()` function, the error message states: "Cannot wrap a zero amount".

Recommendation(s): Consider changing "wrap" to "unwrap" in the message to make it more precise.

Status: Unresolved

Update from the client:

6.25 [Info] Usage of __gap

File(s): src/

Description: The __gap variable is used in many contracts in the EtherFi protocol. This variable is used for upgradable contracts, which are meant to be inherited by other contracts. This is useful to avoid shifting down storage in the inheritance chain. The __gap allows for reserving space for state variables in inherited contracts and avoiding storage layout shifts. Usually, the size of a __gap array is a number that sums with a number of state variable slots to a round number (50 is used by OpenZeppelin). E.g., if a contract has two state variables that take two slots, the gap is defined as __gap[48]. This is a best practice to keep the size of a gap in all upgradable inherited contracts the same, as it prevents mistakes while upgrading those contracts.

Recommendation(s): Consider introducing a gap only to the upgradable contracts that are designed to be inherited. Also, consider keeping the size of a __gap array to sum with the reserved slots by state variables to a round number (e.g., 50). Keeping the round and consistent number of slots in memory helps with the upgrading process. More information can be found in the OpenZeppelin docs. Additionally, make sure that the name of the array is exactly __gap to keep it compatible with tools like a hardhat, which helps with the upgrading process. An example of a different name is in MembershipNFT, which is gap.

Status: Unresolved

Update from the client:

6.26 [Best Practice] Circular dependency and lack of cohesion in contracts

File(s): MeETH.sol, MembershipNFT.sol

Description: Circular dependencies and lack of cohesion are two issues that can lead to inefficiencies and security vulnerabilities. Developers can avoid these issues by carefully considering the dependencies between contracts and ensuring that each function within a contract has a clear purpose and is related to the overall purpose of the contract, e.g., interact with its state variables. The contracts MembershipNFT and MeETH contain both issues that can be significantly minimized by applying **move method** and **move state variables** tasks.

Recommendation(s): As a best practice, we suggest applying some refactoring steps to tackle these issues. Some recommendations are listed below:

1 - **Responsibilities:** The first step would be to define what are the responsibilities that which contract should implement. In this context, MembershipNFT would implement all functions related to the membership itself while MeETH would handle tasks related to stake rewards and deposit. To achieve this goal, tokenData and tierData should be moved to MembershipNFT.

2 - **Moving Functions:** Considering the previous recommendations, the second step would be to apply the move functions. Below we list the functions that could be moved between the contracts to improve cohesion and tackle circular dependencies.

Functions to be moved from MembershipNFT to MeETH:

```
function isWithdrawable(uint256 _tokenId, uint256 _withdrawalAmount) public view returns (bool) {...}
function allTimeHighDepositOf(uint256 _tokenId) public view returns (uint256){...}
```

The function accruedTierPointsOf(...) could be refactored after those changes, e.g., breaking into two functions.

Functions to be moved from MeETH to MembershipNFT:

```
function claimPoints(uint256 _tokenId) public {...}
function setPoints(uint256 _tokenId, uint40 _loyaltyPoints, uint40 _tierPoints) external onlyOwner {...}
function tierForPoints(uint40 _tierPoints) public view returns (uint8) {...}
```

The function _mintMembershipNFT(...) now can be refactored by applying an extraction function: A new function in MembershipNFT to implement the mint and update of the tokenData. Then _mintMembershipNFT(...) would call this new function before the internal function _deposit(...). Similarly, this would happen to the function _applyUnwrapPenalty(...).

Remarks: In the end, some small changes would be needed to adapt the new scenario of the state variables moved to MembershipNFT contract. For example, the function below updates tierData and tokenData in the MeETH.

```
function _claimTier(uint256 _tokenId, uint8 _curTier, uint8 _newTier) internal {...}
```

Status: Unresolved

Update from the client:

6.27 [Best Practice] Lack of events for relevant operations

File(s): src/

Description: Several relevant operations in the contracts do not emit events, making it difficult to monitor and review the contracts' behavior once deployed. Emitting events for the relevant operations will enable users and blockchain monitoring systems to easily detect suspicious behaviors and ensure the correct functioning of the contracts.

Recommendation(s): Add events for all relevant operations to facilitate contract monitoring and detect suspicious behavior more effectively.

Status: Unresolved

Update from the client:

6.28 [Best Practice] Local variable tokenData shadows the existing state variable

File(s): MeETH.sol

Description: The local variable `tokenData` declaration shadows the existing state variable `tokenData`.

```
function claimStakingRewards(uint256 _tokenId) public {
    // @audit The local variable tokenData shadows the existing mapping tokenData
    TokenData storage tokenData = tokenData[_tokenId];
    uint256 tier = tokenData.tier;
    uint256 amount = (tierData[tier].rewardsGlobalIndex - tokenData.rewardsLocalIndex) *
    tokenDeposits[_tokenId].amounts / 1 ether;
    _incrementTokenDeposit(_tokenId, amount, 0);
    tokenData.rewardsLocalIndex = tierData[tier].rewardsGlobalIndex;
}
```

Recommendation(s): Consider renaming the local variable `tokenData`.

Status: Unresolved

Update from the client:

6.29 [Best Practice] No explicit return value in _updateAllTimeHighDepositOf(...)

File(s): MeETH.sol

Description: The function `_updateAllTimeHighDepositOf(...)` does not explicitly return with value. The function is listed below:

```
function _updateAllTimeHighDepositOf(uint256 _tokenId) internal returns (uint256) {
    // @audit There is no explicit return value
    allTimeHighDepositAmount[_tokenId] = allTimeHighDepositOf(_tokenId);
}
```

Recommendation(s): Consider adding an explicit return or removing the return statement.

Status: Unresolved

Update from the client:

6.30 [Best Practice] Place-holder in the struct for future usage

File(s): ImeETH.sol

Description: The TokenData struct contains place-holder member _dummy. We present the struct below:

```
struct TokenData {
    uint96 rewardsLocalIndex;
    uint40 baseLoyaltyPoints;
    uint40 baseTierPoints;
    uint32 prevPointsAccrualTimestamp;
    uint32 prevTopUpTimestamp;
    uint8  tier;
    ////////////////////////////////////////////////////////////////////////////
    // @audit _dummy can be replaced with __gap fixed-sized array
    ////////////////////////////////////////////////////////////////////////////
    uint8  _dummy; // a place-holder for future usage
}
```

This last member is kept to maintain the possibility of increasing the number of fields in the TokenData struct, which is possible since the MeETH contract is upgradeable. However, a standard solution would be to use __gap instead of _dummy. It would allow reserving more place-holder space for future possible updates of the structs and more clearly indicate the purpose of the additional variable.

Recommendation(s): Consider using __gap instead of _dummy.

Status: Unresolved

Update from the client:

6.31 [Best Practice] Some functions can be external

File(s): AuctionManager.sol, LiquidityPool.sol

Description: In the contract AuctionManager the following function can be external:

- function updateSelectedBidInformation(uint256 _bidId) public onlyStakingManagerContract ...;

Similarly, in the contract LiquidityPool:

- function deposit(address _user, bytes32[] calldata _merkleProof) public payable ...;
- function withdraw(address _recipient, uint256 _amount) public whenLiquidStakingOpen ...;
- function batchDepositWithBidIds(uint256 _numDeposits, uint256[] calldata _candidateBidIds, bytes32[] calldata _merkleProof) payable public onlyOwner returns (uint256[] memory) ...;
- function batchRegisterValidators(bytes32 _depositRoot, uint256[] calldata _validatorIds, IStakingManager.DepositData[] calldata _depositData) public onlyOwner ... ;
- function processNodeExit(uint256[] calldata _validatorIds, uint256[] calldata _slashingPenalties) public onlyOwner ...;
- function sendExitRequests(uint256[] calldata _validatorIds) public onlyOwner...;

Recommendation(s): Consider setting these functions as external.

Status: Unresolved

Update from the client:

6.32 [Best Practice] Unnecessary update of `prevPointsAccrualTimestamp`

File(s): `MeETH.sol`

Description: The function `_applyUnwrapPenalty(...)` updates the `prevPointsAccrualTimestamp` for the unwrapped token. The issue lies in the redundancy of this update, as the `claimPoints(...)` function, which is called before `_applyUnwrapPenalty(...)`, already updates this value. Here's the relevant snippet:

```
function _applyUnwrapPenalty(...) internal {
        ...

        // point deduction if scaled proportional to withdrawal amount
        uint256 ratio = (10000 * _withdrawalAmount) / _prevAmount;
        uint40 scaledTierPointsPenalty = uint40((ratio * curTierPoints) / 10000);

        uint40 penalty = uint40(_max(degradeTierPenalty, scaledTierPointsPenalty));

        token.baseTierPoints -= penalty;
        // @audit - This line is unnecessary.
        token.prevPointsAccrualTimestamp = uint32(block.timestamp);
        _claimTier(_tokenId);
    }

```

Recommendation(s): Consider removing the line of code that unnecessarily updates the `prevPointsAccrualTimestamp` in the `_applyUnwrapPenalty(...)` function.

Status: Unresolved

Update from the client:

6.33 [Best Practice] Unused code

File(s): `LiquidityPool.sol`, `MeETH.sol`

Description: The presence of unused code is generally considered a practice to be avoided, as it may lead to potential issues such as:

- Increased computational costs (i.e., unnecessary gas consumption);
- Reduced code readability; and;
- The possibility of bugs (e.g., when an explicit return statement is forgotten and return values are never assigned);

Below we list unused code:

- in the `LiquidityPool.processNodeExit(...)` , the parameter `_slashingPenalties` is never used;

```
// @audit unused parameter
function processNodeExit(uint256[] calldata _validatorIds, uint256[] calldata _slashingPenalties) public onlyOwner {
    numValidators -= _validatorIds.length;
    nodesManager.fullWithdrawBatch(_validatorIds);
}
```

- in the function `MeETH.canTopUp(...)`, to local variable `deposit` is never used;

```
function canTopUp(uint256 _tokenId, uint256 _totalAmount, uint128 _amount, uint128 _amountForPoints) public view
↪ returns (bool) {
        uint32 prevTopUpTimestamp = tokenData[_tokenId].prevTopUpTimestamp;
        // @audit unused variable deposit
        TokenDeposit memory deposit = tokenDeposits[_tokenId];
        uint256 monthInSeconds = 28 days;
        if (block.timestamp - uint256(prevTopUpTimestamp) < monthInSeconds) revert OncePerMonth();
        if (_totalAmount != _amount + _amountForPoints) revert InvalidAllocation();

        return true;
    }
```

- modifier `onlyNodeOperatorManagerContract()` in AuctionManager.sol;
- import `@openzeppelin/contracts/utils/cryptography/MerkleProof.sol` in NodeOperatorManager.sol;
- event declaration `MerkleUpdated(bytes32 oldMerkle, bytes32 indexed newMerkle)` in NodeOperatorManager.sol;

Recommendation(s): We recommend removing all unused code from the codebase to enhance code quality and minimize the risk of unexpected errors or inefficiencies.

Status: Unresolved

Update from the client:

6.34 [Best Practice] Use of 2-step ownership transition

File(s): `src/`

Description: Many contracts in the protocol utilize the OwnableUpgradeable contract. It is recommended to use Ownable2StepUpgradeable since it provides a more secure two-step transfer of the ownership.

Recommendation(s): Consider using Ownable2StepUpgradeable.

Status: Unresolved

Update from the client:

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 plays a critical role in software development, enabling effective communication between developers, testers, users, and other stakeholders involved in the software development process. 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 `Ether.fi` team has provided extensive `documentation` on their official website, including an overview of the protocol and technical insights into its components. They have also shared spreadsheets outlining rewards and punishments for participants in different scenarios and PDF document which describes the membership program. Moreover, the team conducted a comprehensive code walkthrough and maintained open communication to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Contracts Compilation Output

```
> forge build
[] Compiling...
[] Compiling 109 files with 0.8.13
[] Solc 0.8.13 finished in 16.33s
Compiler run successful (with warnings)
```

8.2 Tests Output

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

Running 5 tests for test/RegulationsManager.t.sol:RegulationsManagerTest
[PASS] test_ConfirmEligibilityWorks() (gas: 86038)
[PASS] test_RemoveFromWhitelistWorks() (gas: 150613)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
[PASS] test_initializeNewWhitelistWorks() (gas: 121052)
Test result: ok. 5 passed; 0 failed; finished in 22.33ms

Running 8 tests for test/NodeOperatorManager.t.sol:NodeOperatorManagerTest
[PASS] test_CanAddAddressToWhitelist() (gas: 178655)
[PASS] test_CanOnlySetAddressesOnce() (gas: 20227)
[PASS] test_CanRemoveAddressFromWhitelist() (gas: 153040)
[PASS] test_EventOperatorRegistered() (gas: 143287)
[PASS] test_FetchNextKeyIndex() (gas: 401775)
[PASS] test_RegisterNodeOperator() (gas: 178112)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 8 passed; 0 failed; finished in 20.52ms

Running 12 tests for test/EETH.t.sol:EETHTest
[PASS] test_ApproveWithAmount() (gas: 52835)
[PASS] test_ApproveWithZero() (gas: 28216)
[PASS] test_BurnShares() (gas: 75109)
[PASS] test_EETHInitializedCorrectly() (gas: 67484)
[PASS] test_EEthRebase() (gas: 395000)
[PASS] test_MintShares() (gas: 86445)
[PASS] test_TransferFromWithAmount() (gas: 270720)
[PASS] test_TransferFromWithZero() (gas: 265786)
[PASS] test_TransferWithAmount() (gas: 268853)
[PASS] test_TransferWithZero() (gas: 232972)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
Test result: ok. 12 passed; 0 failed; finished in 31.14ms

Running 6 tests for test/BNFT.t.sol:BNFTTest
[PASS] test_BNFTCannotBeTransferred() (gas: 1185596)
[PASS] test_BNFTMintsFailsIfNotCorrectCaller() (gas: 18237)
[PASS] test_DisableInitializer() (gas: 15594)
[PASS] test_Mint() (gas: 1188210)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 6 passed; 0 failed; finished in 29.68ms

Running 14 tests for test/LiquidityPool.t.sol:LiquidityPoolTest
[PASS] test_LiquidStakingAccessControl() (gas: 224487)
[PASS] test_LiquidityPoolBatchDepositWithBidIds() (gas: 1033421)
[PASS] test_LiquidityPoolBatchRegisterValidators() (gas: 2230855)
[PASS] test_ProcessNodeExit() (gas: 2457850)
[PASS] test_SettersFailOnZeroAddress() (gas: 35491)
[PASS] test_StakingManagerFailsNotInitializedToken() (gas: 2499494)
```

```
[PASS] test_StakingManagerLiquidityFails() (gas: 102683)
[PASS] test_StakingManagerLiquidityPool() (gas: 335866)
[PASS] test_WithdrawFailsNotInitializedToken() (gas: 18786)
[PASS] test_WithdrawLiquidityPoolAccrueStakingRewardsWithoutPartialWithdrawal() (gas: 448405)
[PASS] test_WithdrawLiquidityPoolFails() (gas: 33677)
[PASS] test_WithdrawLiquidityPoolSuccess() (gas: 414186)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
Test result: ok. 14 passed; 0 failed; finished in 27.69ms

Running 29 tests for test/AuctionManager.t.sol:AuctionManagerTest
[PASS] test_AuctionManagerContractInstantiatedCorrectly() (gas: 30254)
[PASS] test_CanOnlySetAddressesOnce() (gas: 30833)
[PASS] test_CancelBidFailsWhenBidAlreadyInactive() (gas: 198711)
[PASS] test_CancelBidFailsWhenNotBidOwnerCalling() (gas: 208225)
[PASS] test_CancelBidFailsWhenNotExistingBid() (gas: 20359)
[PASS] test_CancelBidWorksIfBidIsNotCurrentHighest() (gas: 502450)
[PASS] test_CreateBidMinMaxAmounts() (gas: 349953)

[PASS] test_CreateBidPauseable() (gas: 231727)
[PASS] test_DisableInitializer() (gas: 15726)
[PASS] test_DisableWhitelist() (gas: 32743)
[PASS] test_EnableWhitelist() (gas: 32386)
[PASS] test_EventBidCancelled() (gas: 198084)
[PASS] test_EventBidPlaced() (gas: 210328)
[PASS] test_EventBidReEnteredAuction() (gas: 821624)
[PASS] test_PausableCancelBid() (gas: 445915)
[PASS] test_ProcessAuctionFeeTransfer() (gas: 1297477)
[PASS] test_ReEnterAuctionManagerFailsIfBidAlreadyActive() (gas: 902800)
[PASS] test_ReEnterAuctionManagerFailsIfNotCorrectCaller() (gas: 816062)
[PASS] test_ReEnterAuctionManagerWorks() (gas: 890258)
[PASS] test_SetMaxBidAmount() (gas: 47732)
[PASS] test_SetMinBidAmount() (gas: 47740)
[PASS] test_SetWhitelistBidAmount() (gas: 58452)
[PASS] test_createBidBatch() (gas: 991405)
[PASS] test_createBidBatchFailsWithIncorrectValue() (gas: 124887)
[PASS] test_createBidFailsIfBidSizeIsLargerThanKeysRemaining() (gas: 339246)
[PASS] test_createBidFailsIfIPFSIndexMoreThanTotalKeys() (gas: 239771)
[PASS] test_createBidWorks() (gas: 869846)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10899)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10878)
Test result: ok. 29 passed; 0 failed; finished in 29.59ms

Running 18 tests for test/EtherFiNodeManager.t.sol:EtherFiNodesManagerTest
[PASS] test_CreateEtherFiNode() (gas: 827536)
[PASS] test_PausableModifierWorks() (gas: 95238)
[PASS] test_RegisterEtherFiNode() (gas: 827517)
[PASS] test_RegisterEtherFiNodeRevertsIfAlreadyRegistered() (gas: 26838)
[PASS] test_RegisterEtherFiNodeRevertsOnIncorrectCaller() (gas: 24698)
[PASS] test_SendExitRequestWorksCorrectly() (gas: 172149)
[PASS] test_SetEtherFiNodePhaseRevertsOnIncorrectCaller() (gas: 22577)
[PASS] test_SetNonExitPenaltyDailyRate() (gas: 36531)
[PASS] test_SetNonExitPenaltyPrincipal() (gas: 36414)
[PASS] test_SetProtocolRewardsSplit() (gas: 52658)
[PASS] test_SetStakingRewardsSplit() (gas: 52794)
[PASS] test_UnregisterEtherFiNode() (gas: 26051)
[PASS] test_UnregisterEtherFiNodeRevertsIfAlreadyUnregistered() (gas: 25247)
[PASS] test_UnregisterEtherFiNodeRevertsOnIncorrectCaller() (gas: 22458)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
[PASS] test_setEtherFiNodeIpfsHashForEncryptedValidatorKeyRevertsOnIncorrectCaller() (gas: 22717)
[PASS] test_setEtherFiNodeLocalRevenueIndexRevertsOnIncorrectCaller() (gas: 22470)
Test result: ok. 18 passed; 0 failed; finished in 29.77ms

Running 3 tests for test/LargeScenario.t.sol:LargeScenariosTest
[PASS] test_LargeScenarioOne() (gas: 15459703)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 3 passed; 0 failed; finished in 32.10ms

Running 4 tests for test/SmallScenerios.t.sol:SmallScenariosTest
[PASS] test_AuctionToStakerFlow() (gas: 5940045)
[PASS] test_EEthWeTHLpScenarios() (gas: 2540643)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
```

```
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
Test result: ok. 4 passed; 0 failed; finished in 41.60ms

Running 6 tests for test/Treasury.t.sol:TreasuryTest
[PASS] test_TreasuryCanReceiveFunds() (gas: 15706)
[PASS] test_WithdrawFailsIfNotOwner() (gas: 21064)
[PASS] test_WithdrawPartialWorks() (gas: 28745)
[PASS] test_WithdrawWorks() (gas: 32734)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 6 passed; 0 failed; finished in 42.72ms

Running 22 tests for test/EarlyAdopterPool.t.sol:EarlyAdopterPoolTest
[PASS] test_ClaimFailsIfClaimingIsComplete() (gas: 239932)
[PASS] test_ClaimFailsIfClaimingNotOpen() (gas: 192554)
[PASS] test_ClaimFailsIfClaimingReceiverNotSet() (gas: 236783)
[PASS] test_ClaimWorks() (gas: 402000)
[PASS] test_DepositAndClaimWithERC20AndETH() (gas: 532140)
[PASS] test_DepositERC20IntoEarlyAdopterPool()
[PASS] test_DepositETHIntoEarlyAdopterPool() (gas: 101321)
[PASS] test_EventERC20TVLUpdated() (gas: 502205)
[PASS] test_EventEthTVLUpdated() (gas: 163713)
[PASS] test_EventTVLUpdatedOnERC20AndEthDeposit() (gas: 508996)
[PASS] test_GetTVL() (gas: 335757)
[PASS] test_GetUserTVL() (gas: 410893)
[PASS] test_PointsCalculatorWorksCorrectly() (gas: 757136)
[PASS] test_RewardsPoolMaxDeposit() (gas: 13772)
[PASS] test_RewardsPoolMinDeposit() (gas: 13727)
[PASS] test_SetClaimableStatusFailsIfNotOwner() (gas: 13187)
[PASS] test_SetClaimableStatusTrue() (gas: 82226)
[PASS] test_SetReceiverAddress() (gas: 40339)
[PASS] test_SetReceiverFailsIfAddressZero() (gas: 13327)
[PASS] test_SetReceiverFailsIfNotOwner() (gas: 13373)
[PASS] test_SetUp() (gas: 25409)
[PASS] test_WithdrawWorksCorrectly() (gas: 399849)
Test result: ok. 22 passed; 0 failed; finished in 5.60ms

Running 7 tests for test/ProtocolRevenueManager.t.sol:ProtocolRevenueManagerTest
[PASS] test_AddAuctionRevenueWorksAndFailsCorrectly() (gas: 996606)
[PASS] test_GetAccruedAuctionRevenueRewards() (gas: 1645201)
[PASS] test_Receive() (gas: 1723652)
[PASS] test_changeAuctionRewardParams() (gas: 46361)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
[PASS] test_modifiers() (gas: 43669)
Test result: ok. 7 passed; 0 failed; finished in 23.43ms

Running 2 tests for test/TestSetup.sol:TestSetup
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 2 passed; 0 failed; finished in 779.67µs

Running 5 tests for test/RewardsSkimmingTest.t.sol:RewardsSkimmingTest
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
[PASS] test_partialWithdrawBatchForTNftInLiquidityPool() (gas: 327557)
[PASS] test_partialWithdrawBatchGroupByOperator() (gas: 1272080)
[PASS] test_partialWithdraw_batch_base() (gas: 1405411)
Test result: ok. 5 passed; 0 failed; finished in 62.12ms

Running 5 tests for test/TNFT.t.sol:TnftTest
[PASS] test_DisableInitializer() (gas: 15610)
[PASS] test_Mint() (gas: 1188128)
[PASS] test_TNFTMintsFailsIfNotCorrectCaller() (gas: 18238)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10875)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 5 passed; 0 failed; finished in 16.69ms

Running 19 tests for test/MeETH.t.sol:MeETHTest
[PASS] test_EapMigrationFails() (gas: 243209)
[PASS] test_EapMigrationWorks() (gas: 565882)
[PASS] test_HowPointsGrow() (gas: 572740)
[PASS] test_MaximumPoints() (gas: 349059)
[PASS] test_MembershipTier() (gas: 593963)
```

```
[PASS] test_MixedDeposits() (gas: 657425)
[PASS] test_OwnerPermissions() (gas: 44931)
[PASS] test_SacrificeRewardsForPoints() (gas: 1033852)
[PASS] test_StakingRewards() (gas: 1080155)
[PASS] test_UpdatingPointsGrowthRate() (gas: 328419)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10898)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
[PASS] test_setPoints() (gas: 415199)
[PASS] test_topUpDepositWithEth() (gas: 573836)
[PASS] test_topUpDilution() (gas: 727878)
[PASS] test_trade() (gas: 418991)
[PASS] test_unwrapForEth() (gas: 535410)
[PASS] test_withdrawalPenalty() (gas: 868985)
[PASS] test_wrapEth() (gas: 570481)
Test result: ok. 19 passed; 0 failed; finished in 22.83ms

Running 10 tests for test/Upgradable.t.sol:UpgradeTest
[PASS] test_CanUpgradeAuctionManager() (gas: 2717348)
[PASS] test_CanUpgradeBNFT() (gas: 2192671)
[PASS] test_CanUpgradeEtherFiNodesManager() (gas: 4350130)
[PASS] test_CanUpgradeNodeOperatorManager() (gas: 1667997)
[PASS] test_CanUpgradeProtocolRevenueManager() (gas: 1929109)
[PASS] test_CanUpgradeStakingManager() (gas: 4558380)
[PASS] test_CanUpgradeTNFT() (gas: 2169789)
[PASS] test_canUpgradeEtherFiNode() (gas: 3570384)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
Test result: ok. 10 passed; 0 failed; finished in 20.28ms

Running 4 tests for test/MembershipNFT.t.sol:MembershipNFTTest
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10854)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10877)
[PASS] test_metadata() (gas: 119155)
[PASS] test_permissions() (gas: 93291)
Test result: ok. 4 passed; 0 failed; finished in 16.18ms

Running 33 tests for test/StakingManager.t.sol:StakingManagerTest
[PASS] test_BatchDepositWithBidIdsFailsIFInvalidDepositAmount() (gas: 634637)
[PASS] test_BatchDepositWithBidIdsFailsIfNoIdsProvided() (gas: 1681636)
[PASS] test_BatchDepositWithBidIdsFailsIfNotEnoughActiveBids() (gas: 347092)
[PASS] test_BatchDepositWithBidIdsFailsIfPaused() (gas: 1701333)
[PASS] test_BatchDepositWithIdsComplexWorksCorrectly() (gas: 2462922)
[PASS] test_BatchDepositWithIdsSimpleWorksCorrectly() (gas: 2092402)
[PASS] test_BatchRegisterValidatorFailsIfArrayLengthAreNotEqual() (gas: 2266053)
[PASS] test_BatchRegisterValidatorFailsIfIncorrectPhase() (gas: 4442688)
[PASS] test_BatchRegisterValidatorFailsIfMoreThan16Registers() (gas: 2697887)
[PASS] test_BatchRegisterValidatorWorksCorrectly() (gas: 4372250)
[PASS] test_CanOnlySetAddressesOnce() (gas: 62570)
[PASS] test_CorrectValidatorAttatchedToNft() (gas: 2052307)
[PASS] test_DepositOneWorksCorrectly() (gas: 1229922)
[PASS] test_DisableInitializer() (gas: 15740)
[PASS] test_EnablingAndDisablingWhitelistingWorks() (gas: 27015)
[PASS] test_EventDepositCancelled() (gas: 819887)
[PASS] test_EventValidatorRegistered() (gas: 1196909)
[PASS] test_GenerateWithdrawalCredentialsCorrectly() (gas: 11830)
[PASS] test_MaxBatchBidGasFee() (gas: 2045428)
[PASS] test_RegisterValidatorFailsIfContractPaused() (gas: 845770)
[PASS] test_RegisterValidatorFailsIfIncorrectCaller() (gas: 827481)
[PASS] test_RegisterValidatorFailsIfIncorrectPhase() (gas: 1191596)
[PASS] test_RegisterValidatorWorksCorrectly() (gas: 1198980)
[PASS] test_SetMaxDeposit() (gas: 29727)
[PASS] test_StakingManagerContractInstantiatedCorrectly() (gas: 18141)
[PASS] test_cancelDepositFailsIfDepositDoesNotExist() (gas: 820739)
[PASS] test_cancelDepositFailsIfIncorrectPhase() (gas: 1197504)
[PASS] test_cancelDepositFailsIfNotStakeOwner() (gas: 825673)
[PASS] test_cancelDepositWorksCorrectly() (gas: 965409)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10854)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10878)
[PASS] test_fake() (gas: 112431)
[PASS] test_reproduceBugFromSimulator() (gas: 37371)
Test result: ok. 33 passed; 0 failed; finished in 79.29ms
```

```
Running 8 tests for test/WeETH.t.sol:WeETHTest
[PASS] test_MultipleDepositsAndFunctionalityWorksCorrectly() (gas: 644570)
[PASS] test_UnWrapEETHFailsIfZeroAmount() (gas: 13455)
[PASS] test_UnWrapWorksCorrectly() (gas: 304348)
[PASS] test_UnwrappingWithRewards() (gas: 422407)
[PASS] test_WrapEETHFailsIfZeroAmount() (gas: 13466)
[PASS] test_WrapWorksCorrectly() (gas: 343950)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10854)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
Test result: ok. 8 passed; 0 failed; finished in 18.81ms

Running 29 tests for test/EtherFiNode.t.sol:EtherFiNodeTest
[PASS] test_EtherFiNodeMultipleSafesWorkCorrectly() (gas: 2067453)
[PASS] test_ExitRequestAfterExitFails() (gas: 197837)
[PASS] test_ExitTimestampBeforeExitRequestLeadsToZeroNonExitPenalty() (gas: 203079)
[PASS] test_ImplementationContract() (gas: 18772)
[PASS] test_SetExitRequestTimestampFailsOnIncorrectCaller() (gas: 23227)
[PASS] test_SetExitRequestTimestampRevertsOnIcorrectCaller() (gas: 23259)
[PASS] test_SetIpfsHashForEncryptedValidatorKeyRevertsOnIcorrectCaller() (gas: 23476)
[PASS] test_SetLocalRevenueIndexRevertsOnIcorrectCaller() (gas: 23253)
[PASS] test_SetPhaseRevertsOnIcorrectCaller() (gas: 23318)
[PASS] test_data():(bytes,bytes,bytes32,string) (gas: 10853)
[PASS] test_data_2():(bytes,bytes,bytes32,string) (gas: 10855)
[PASS] test_getFullWithdrawBeforeVestingPeriodAndPartialWithdrawAfterVestingPeriod() (gas: 396632)
[PASS] test_getFullWithdrawalPayoutsFails() (gas: 48675)
[PASS] test_getFullWithdrawalPayoutsWorksCorrectly1() (gas: 388893)
[PASS] test_getFullWithdrawalPayoutsWorksCorrectlyAfterVestingPeriod() (gas: 253029)
[PASS] test_getFullWithdrawalPayoutsWorksWithNonExitPenaltyCorrectly1() (gas: 239725)
[PASS] test_getFullWithdrawalPayoutsWorksWithNonExitPenaltyCorrectly2() (gas: 241968)
[PASS] test_getFullWithdrawalPayoutsWorksWithNonExitPenaltyCorrectly3() (gas: 246962)
[PASS] test_getFullWithdrawalPayoutsWorksWithNonExitPenaltyCorrectly4() (gas: 244802)
[PASS] test_getFullWithrdawalPayoutsAuditFix1() (gas: 243103)
[PASS] test_getFullWithrdawalPayoutsAuditFix2() (gas: 243102)
[PASS] test_getFullWithrdawalPayoutsAuditFix3() (gas: 245945)
[PASS] test_markExitedFails() (gas: 27057)
[PASS] test_markExitedWorksCorrectly() (gas: 220612)
[PASS] test_partialWithdraw() (gas: 327369)
[PASS] test_partialWithdrawAfterExitRequest() (gas: 377558)
[PASS] test_partialWithdrawFails() (gas: 70719)
[PASS] test_processNodeDistributeProtocolRevenueCorrectly() (gas: 204910)
[PASS] test_sendEthToEtherFiNodeContractSucceeds() (gas: 42681)
Test result: ok. 29 passed; 0 failed; finished in 22.23ms
```

8.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

```
Analysing contracts...
Running tests...
| File                                  | % Lines             | % Statements         | % Branches           | % Funcs            |
|---------------------------------------|---------------------|----------------------|----------------------|--------------------|
| src/AuctionManager.sol                | 79.22% (61/77)      | 81.40% (70/86)       | 75.00% (33/44)       | 95.45% (21/22)     |
| src/BNFT.sol                          | 37.50% (3/8)        | 37.50% (3/8)         | 50.00% (2/4)         | 100.00% (5/5)      |
| src/EETH.sol                          | 88.89% (32/36)      | 89.19% (33/37)       | 85.71% (12/14)       | 88.24% (15/17)     |
| src/EtherFiNode.sol                   | 99.26% (134/135)    | 99.35% (153/154)     | 76.67% (46/60)       | 90.91% (20/22)     |
| src/EtherFiNodesManager.sol           | 77.12% (118/153)    | 76.69% (125/163)     | 45.31% (29/64)       | 89.74% (35/39)     |
| src/LiquidityPool.sol                 | 81.16% (56/69)      | 84.15% (69/82)       | 63.16% (24/38)       | 81.82% (18/22)     |
| src/MeETH.sol                         | 85.04% (216/254)    | 82.52% (269/326)     | 54.55% (36/66)       | 71.43% (35/49)     |
| src/MembershipNFT.sol                 | 84.31% (43/51)      | 89.19% (66/74)       | 50.00% (1/2)         | 63.64% (14/22)     |
| src/NodeOperatorManager.sol           | 78.57% (22/28)      | 79.31% (23/29)       | 50.00% (4/8)         | 92.31% (12/13)     |
| src/ProtocolRevenueManager.sol        | 58.97% (23/39)      | 65.22% (30/46)       | 44.44% (8/18)        | 83.33% (10/12)     |
| src/RegulationsManager.sol            | 76.47% (13/17)      | 76.47% (13/17)       | 100.00% (6/6)        | 62.50% (5/8)       |
| src/StakingManager.sol                | 77.27% (85/110)     | 80.00% (100/125)     | 58.33% (35/60)       | 88.89% (24/27)     |
| src/TNFT.sol                          | 28.57% (2/7)        | 28.57% (2/7)         | 0.00% (0/2)          | 100.00% (4/4)      |
| src/Treasury.sol                      | 100.00% (4/4)       | 100.00% (5/5)        | 66.67% (4/6)         | 100.00% (1/1)      |
| src/WeETH.sol                         | 47.62% (10/21)      | 52.17% (12/23)       | 50.00% (4/8)         | 28.57% (2/7)       |
| Total                                 | 81.46% (822/1009)   | 82.31% (973/1182)    | 61.00% (244/400)     | 81.85% (221/270)   |
```

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.

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.