Security Assessment Report

ether.fi – ETHFI Core Audit Report

December 2025 - January 2026

Prepared for ether.fi

certora

ether.fi

Table of contents

Project Summary

Project Scope

Project Name	Review Commit	Fixes Initial Commit	Fixes Final Commit	Platform	Start Date	End Date
ETHFI smart-contracts	733b5f5	67588052	77381e3f2	EVM	10/12/2025	13/01/2026

Project Overview

This document describes the manual review of the live contract code for the **ETHFI smart-contracts** core repository.

The work was a 30 day effort undertaken between **10/12/2025** and **13/01/2026**

The following contract list is included in our scope:

- src/AssetRecovery.sol
- src/BucketRateLimiter.sol
- src/DepositAdapter.sol
- src/EETH.sol
- src/EtherFiAdmin.sol
- src/EtherFiNode.sol
- src/EtherFiNodesManager.sol
- src/EtherFiOracle.sol
- src/EtherFiRateLimiter.sol
- src/EtherFiRedemptionManager.sol
- src/EtherFiRestaker.sol
- src/EtherFiRewardsRouter.sol
- src/EtherFiTimelock.sol
- src/LiquidityPool.sol
- src/Liquifier.sol
- src/MembershipManager.sol

- src/MembershipNFT.sol
- src/RoleRegistry.sol
- src/StakingManager.sol
- src/TVLOracle.sol
- src/UUPSProxy.sol
- src/WeETH.sol
- src/WithdrawRequestNFT.sol
- src/helpers/AddressProvider.sol
- src/helpers/EtherFiOperationParameters.sol
- src/helpers/EtherFiViewer.sol
- src/helpers/WeETHWithdrawAdapter.sol
- src/libraries/DepositDataRootGenerator.sol
- src/libraries/GlobalIndexLibrary.sol

The team performed a manual audit of the deployed Solidity smart contracts. During the manual audit, the Certora team discovered bugs in the Solidity smart contracts code, as listed on the following pages.

Threat Model

Assets

- **Native ETH:** Held in Liquidity Pool, Restaker contracts, Staking Manager.
- **Liquid Staking Tokens (eETH):** Represents user shares of the underlying ETH pool.
- **External Assets (stETH, cbETH, wBETH):** Held by Restaker contracts for yield generation.
- **Membership NFTs:** ERC1155 tokens representing user tiers and loyalty status.
- **Abstract Value:** Loyalty Points, Tier Points, and accrued staking rewards.
- **Validator Stakes:** 32+ ETH deposits locked on the Beacon Chain.

Actors

- **EtherFi Admin Roles:** Privileged roles capable of upgrades, parameter setting, pausing and handling a wide range of permissioned flows related to managing pooled assets
- **Oracle:** Trusted entity responsible for reporting portfolio performance (rewards/slashing in EigenLayer strategies, Lido, etc.).
- **User/Staker:** Participants depositing ETH and holding eETH or Membership NFTs.
- **Node Operator:** Entities managing the validator infrastructure.
- **External Protocols:** Lido and EigenLayer (dependencies for withdrawals and restaking).

Trust Assumptions

- **Admin Competence & Honesty:** Assumes admins will not misconfigure critical parameters.
- **Oracle Reliability:** Relies on the Oracle to accurately report rewards and slashing without desynchronizing from the Admin contract.
- **Event completeness:** Off-chain systems assume emitted events fully describe on-chain state
- **Bounded data structures:** Assumes admin managed arrays will not grow unbounded
- **External Protocol Availability:** Assumes Lido and EigenLayer protocols function within expected constraints (e.g., minimum withdrawal amounts).

Attack Vectors

- **Denial of Service (DoS):**
  - **Donation Attacks:** Sending dust ETH to contracts to force accounting mismatches and revert withdrawals.
  - **Logic Bombs:** Exploiting array length mismatches or integer overflows (e.g., during negative rebases) to permanently halt reward distribution.
- **Yield Leakage & Theft:**
  - **Slashing Evasion:** Migrating legacy positions during a loss event to reset principal value.
  - **Rounding Exploits:** Manipulating withdrawal amounts to bypass rate limits or extract dust value due to floor rounding.
  - **Griefing:** Resetting reward accrual timestamps frequently to prevent other users from earning points.
- **State Corruption:**
  - **Protocol Desynchronization:** Modifying Oracle parameters without updating the Admin, breaking the reporting pipeline.
  - **Logic Mixing:** Conflating legacy (V0) and new (V1) storage layouts, leading to incorrect share calculations.
- **Safety Failure:**
  - **Broken Pausing:** Critical functions lacking the whenNotPaused modifier, rendering emergency stops ineffective.
  - **Reentrancy:** Exploiting NFT onReceived callbacks to manipulate state during minting.
- **Accounting & Reporting Failures:**
  - **TVL Corruption:** Underreporting protocol value by ignoring specific assets (e.g., cbETH) or failing to account for slashing penalties in pending redemptions, leading to incorrect share pricing.
  - **Casting Underflows:** Corrupting global TVL variables during negative rebases (slashing events) by unsafely casting negative signed integers to unsigned integers.
- **Lifecycle & Initialization Risks:**
  - **Re-initialization:** Lack of idempotency in upgrade initialization functions, allowing admins to accidentally execute them multiple times and overwrite critical system addresses.

Protocol Overview

Ether.fi is a non-custodial liquid staking and restaking protocol where users deposit ETH to mint eETH, a rebasing ERC20 token that represents their share of the underlying staked assets and accrued rewards. The core architecture relies on a LiquidityPool contract that coordinates with a StakingManager to provision Ethereum validators, while an EtherFiOracle committee submits consensus layer reports to trigger rebasing and update the protocol's total value locked. A key differentiator is its deep integration with EigenLayer; the EtherFiNodesManager and EtherFiNode contracts facilitate native restaking by creating EigenPods, allowing validators to secure Actively Validated Services (AVS) for additional yield. The protocol also supports the intake and management of external Liquid Staking Tokens (LSTs) like stETH, which are managed and delegated via the EtherFiRestaker. Additionally, the system includes a loyalty layer managed by a MembershipManager, which tracks user engagement and distributes tier-based rewards via NFTs. Finally, the entire system is built on upgradeable UUPS proxy contracts, allowing for modular improvements to its staking, oracle, and node management logic.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	8	8	2
Low	14	14	4
Informational	19	19	6
Total	41	41	12

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
M-01	Incorrect TVL Reporting in Liquifier & Restaker	Medium	Fixed
M-02	Unbacked Shares Minted due to stETH Transfer Rounding	Medium	Fixed
M-03	Critical State Corruption in setPoints due to V0/V1 Logic Conflict	Medium	Acknowledged
M-04	Rewards Distribution for New Tiers(V1) could brake	Medium	Acknowledged
M-05	Incorrect V0 Rewards Calculation during Loss	Medium	Acknowledged
M-06	Rounding Issue in _withdraw allows value extraction	Medium	Acknowledged
M-07	Denial of Service in rebase due to Array Length Mismatch	Medium	Acknowledged
M-08	Griefing Attack on Tier Points Accrual	Medium	Acknowledged
L-01	Denial of Service in withdrawEther via Donation	Low	Fixed
L-02	Missing Slashing Accounting in TVL Calculation	Low	Fixed

L-03	Inaccurate TVL for Pending Redemptions due to Slashing	Low	Fixed
L-04	Inaccurate TVL for Pending Redemptions due to Slashing	Low	Acknowledged
L-05	Loss Avoidance during V0 to V1 Migration	Low	Acknowledged
L-06	Re-initialization Risk in initializeOnUpgrade	Low	Acknowledged
L-07	Zombie State Creation in claim	Low	Acknowledged
L-08	rewardsGlobalIndex unnecessarily stored as vaultShares	Low	Acknowledged
L-09	Broken EAP Calculation for Bronze Tier	Low	Acknowledged
L-10	Reward index calculation does not factor losses for V0 positions	Low	Acknowledged
L-11	Hypothetical underflow risk in rebase during slashing events	Low	Acknowledged
L-12	Protocol Desynchronization via setReportStartSlot	Low	Acknowledged
L-13	Broken Pausing Mechanism	Low	Acknowledged
L-14	Treasury receives less than intended fee amount due to share value inflation after burning staker fee shares	Low	Fixed

Medium Severity Issues

M-01 Incorrect TVL Reporting in Liquifier & Restaker

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Liquifier.sol	Status: Fixed	

Description: The Liquifier contract contains view functions intended to report the Total Value Locked (TVL) and the distribution of assets (restaked, holding, pending withdrawals). Specifically, the getTotalPooledEtherSplits function relies on tokenInfos[_token].strategyShare and tokenInfos[_token].ethAmountPendingForWithdrawals to calculate the value of assets currently restaked in EigenLayer or pending withdrawal.

However, these fields (strategyShare and ethAmountPendingForWithdrawals) are never updated in the contract logic. The depositWithERC20 function transfers assets to the EtherFiRestaker contract, but there is no callback or mechanism to update the Liquifier's state when those assets are deposited into a strategy or queued for withdrawal.

As a result, getTotalPooledEther will consistently report incorrect amounts for restaked tokens. If this function is used by off-chain oracles or front-ends to determine the protocol's exchange rate or health, it will lead to incorrect data, potentially causing share dilution or incorrect rebase rewards.

The same issue exists on the EtherFiRestaker contract, where the getTotalPooledEther() function (without arguments) calculates the total value held by the EtherFiRestaker. Currently, it only sums the ETH balance and the value of Lido assets (stETH). However, the Liquifier contract deposits other tokens (like cbETH, wBETH) into EtherFiRestaker. These assets are completely ignored in this calculation. If this function is used to calculate the protocol's Total Value Locked (TVL) or exchange rates, it will underreport the value, leading to incorrect share pricing.

Recommendations: Implement a mechanism to synchronize the state between EtherFiRestaker and Liquifier, or refactor Liquifier to query EtherFiRestaker (or the underlying EigenLayer contracts) directly for the current state of restaked assets and pending withdrawals.

Inside EtherFiRestaker iterates through all supported tokens (transferred from Liquifier or a local list) and sums their values, similar to how Liquifier.getTotalPooledEther() attempts to do.

Customer's response: Fixed in commit 6423174

Fix Review: Fixed

M-02 Unbacked Shares Minted due to stETH Transfer Rounding

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Liquifier.sol	Status: Fixed	

Description: When depositing stETH via depositWithERC20, the Liquifier contract transfers the specified _amount from the user to the EtherFiRestaker (or itself for L2). It then transfers the minted eETH shares based on this _amount.

However, stETH is known to have a 1-2 wei rounding issue where the amount actually received by the recipient can be slightly less than the amount requested in transferFrom. Because the Liquifier does not verify the actual amount received (by checking balances before and after the transfer), the protocol credits the user for the full _amount while receiving _amount - (1-2) wei.

Over time, this discrepancy creates a deficit in the protocol, leading to the minting of unbacked eETH shares.

Recommendations: Implement a "balance before" and "balance after" check to determine the actual amount of tokens received. Use this actual amount for the subsequent share calculation and minting logic.

Customer's response: Fixed in commit f27ba44

Fix Review: Fixed

M-03 Critical State Corruption in setPoints due to V0/V1 Logic Conflict

Severity: **Medium**	Impact: **High**	Likelihood: **Medium**
Files: MembershipManager.sol	Status: Acknowledged	

Description: The setPoints function allows an admin to update points for a token. However, it inadvertently mixes V0 and V1 logic in a way that corrupts the protocol's accounting.

1. It calls _claimStakingRewards(_tokenId). For V0 tokens, this function updates tokenData[_tokenId].vaultShare (slot 0) with rewardsGlobalIndex. This is legacy behavior where the first slot of TokenData was used for the rewards index.
2. It then calls _claimTier(_tokenId). This function assumes V1 logic. It reads tokenData[_tokenId].vaultShare expecting it to be a "Vault Share" (amount of shares in the tier vault).
3. Instead, it reads the rewardsGlobalIndex. It then performs calculations using eEthShareForVaultShare and vaultShareForEEthShare based on this incorrect value.
4. Finally, it updates the global tierVaults accounting (_decrementTierVaultV1 / _incrementTierVaultV1) using these bogus values.

This sequence corrupts the tierVaults state systemwide, permanently desynchronizing the total vault shares from the actual backing assets. This can lead to incorrect exchange rates for all users in those tiers.

Recommendations: Ensure the token is fully migrated to V1 before performing any V1-specific operations like _claimTier. Call _migrateFromV0ToV1(_tokenId) explicitly in setPoints after claiming rewards and before claiming tier.

Customer's response: Acknowledged

Fix Review: Acknowledged

M-04 Rewards Distribution for New Tiers(V1) could brake

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: MembershipManager.sol	Status: Acknowledged	

Description: The _distributeStakingRewardsV1 function iterates through tiers to update their share of rewards. However, the loop condition uses tierDeposits.length.

```javascript
for (uint256 i = 0; i < tierDeposits.length; i++) {
    tierVaults[i].totalPooledEEthShares = vaultTotalPooledEEthShares[i];
}
```

tierDeposits is a legacy V0 array. When new tiers are added via addNewTier, they are added to tierData and tierVaults (V1), but not to tierDeposits. Consequently, tierDeposits.length will be smaller than tierVaults.length. The loop will terminate early, and users in the newly added tiers will not receive any staking rewards updates.

Recommendations: Update the loop in _distributeStakingRewardsV1 to iterate up to tierVaults.length or tierData.length instead of tierDeposits.length.

Customer's response: Acknowledged

Fix Review: Acknowledged

M-05 Incorrect V0 Rewards Calculation during Loss

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: MembershipManager.sol	Status: Acknowledged	

Description: In _distributeStakingRewardsV0, the contract updates the shares associated with V0 deposits:

```javascript
tierDeposits[i].shares = uint128(liquidityPool.sharesForAmount(tierDeposits[i].amounts));
```

It uses sharesForAmount on the fixed ETH principal (amounts). If the pool suffers a loss (exchange rate drops), sharesForAmount increases. The contract then assumes it holds more shares than it actually does (since the actual shares held by MembershipManager in the LP did not increase).

This inflated share count is used to calculate the globalIndex. This can lead to incorrect reward distribution or accounting errors where the contract attempts to distribute rewards it does not possess.

Recommendations: The V0 accounting logic relies on the assumption that sharesForAmount is stable or decreasing (appreciation). It is fundamentally incompatible with slashing/loss scenarios. Consider forcing migration to V1 on all positions to make sure all reward distribution happens on the V1 branch.

Customer's response: Acknowledged

Fix Review: Acknowledged

M-06 Rounding Issue in _withdraw allows value extraction

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: MembershipManager.sol	Status: Acknowledged	

Description: In the _withdraw function, the amount of vault shares to burn is calculated using vaultShareForEthAmount, which rounds down.

```
JavaScript
uint256 vaultShare = vaultShareForEthAmount(tier, _amount);
// ...
_decrementTokenVaultShareV1(_tokenId, vaultShare);
```

A user withdrawing _amount of ETH value will burn slightly fewer vault shares than mathematically required. They retain the "dust" shares in their vaultShare balance. Since vaultShare represents a claim on the pool, retaining these shares allows the user to potentially withdraw slightly more than they deposited over many transactions, or keeps the NFT "active" with a non-zero balance even after a full withdrawal.

Recommendations: Use a rounding-up mechanism (ceiling) when calculating the shares to burn for a given withdrawal amount to favor the protocol and prevent dust accumulation.

Customer's response: Acknowledged

Fix Review: Acknowledged

M-07 Denial of Service in rebase due to Array Length Mismatch

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: GlobalIndexLibrary.sol	Status: Acknowledged	

Description: The calculateRewardsPerTierV0 function iterates through all tiers to calculate rewards. The loop limit is determined by membershipManager.numberOfTiers(), which returns the length of the tierData array (V1 storage).

However, inside the loop, the function accesses membershipManager.tierDeposits(i). tierDeposits is a legacy V0 array. When new tiers are added via MembershipManager.addNewTier, the function updates tierData and tierVaults but does not extend tierDeposits.

As a result, if any new tier is added, numberOfTiers will exceed the length of tierDeposits. When calculateRewardsPerTierV0 attempts to access the index corresponding to the new tier in tierDeposits, the transaction will revert due to an array out-of-bounds error. Since this function is a critical component of the rebase workflow, adding a new tier will permanently break the protocol's ability to distribute rewards.

```JavaScript
for (uint256 i = 0; i < numberOfTiers; i++) {
            //@audit-issue - if new tier is added, deposits array length will not grow and
cause a DOS on rebasing
            (uint128 amounts, uint128 shares) = membershipManager.tierDeposits(i);
```

Recommendations: Modify the loop in calculateRewardsPerTierV0 to iterate only up to the length of tierDeposits (or a separate V0 tier count), rather than numberOfTiers which reflects the

V1 state. Alternatively, ensure tierDeposits is resized when new tiers are added, even if unused by V1.

Customer's response: Acknowledged

Fix Review: Acknowledged

M-08 Griefing Attack on Tier Points Accrual

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: MembershipNFT.sol	Status: Acknowledged	

Description: The accruedTierPointsOf function calculates the points earned since the last accrual timestamp. The formula divides the elapsed time by 1 days (86400 seconds) and multiplies by tierPointsPerDay (24).

```
JavaScript
uint256 earnedPoints = ((uint32(block.timestamp) - prevPointsAccrualTimestamp) *
tierPointsPerDay) / 1 days;
```

This simplifies to elapsed / 3600. If the elapsed time is less than 3600 seconds (1 hour), the integer division results in 0.

The MembershipManager.claim(uint256 _tokenId) function is public and calls _claimPoints, which updates the prevPointsAccrualTimestamp to the current block timestamp.
A malicious actor can exploit this by calling claim on a victim's token ID every 59 minutes. This will repeatedly reset the prevPointsAccrualTimestamp while earnedPoints remains 0 due to rounding. As a result, the victim will not accrue any tier points, effectively denying them the benefits of their membership duration.

Recommendations: Update the prevPointsAccrualTimestamp only when at least 1 point is accrued. Alternatively, restrict the claim function so it can only be called by the token owner or approved operators, preventing third parties from resetting the accrual timer.

Customer's response: Acknowledged

Fix Review: Acknowledged

Low Severity Issues

L-01 Denial of Service in withdrawEther via Donation		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Liquifier.sol	Status: Fixed	

Description: The withdrawEther function allows an admin to sweep ETH held by the Liquifier contract back to the LiquidityPool. The function sends the entire balance of the Liquifier contract (address(this).balance) to the LiquidityPool.

In the LiquidityPool.receive() function, the received amount is subtracted from totalValueOutOfLp.

```JavaScript
receive() external payable {
        if (msg.value > type(uint128).max) revert InvalidAmount();
        totalValueOutOfLp -= uint128(msg.value);
        totalValueInLp += uint128(msg.value);
    }
```

totalValueOutOfLp tracks the amount of ETH that has left the pool (e.g., sent to Liquifier or validators). If the Liquifier contract holds more ETH than the LiquidityPool expects (i.e., address(this).balance > totalValueOutOfLp), the subtraction in LiquidityPool will underflow and revert.

This state might be reached if a malicious actor donates a small amount of ETH (dust) to the Liquifier contract, or if the Liquifier accumulates ETH from sources not accounted for in

totalValueOutOfLp. This effectively locks the ETH in the Liquifier contract as withdrawEther will always revert.

The issue also applies:

- to the withdrawEther() function inside EtherFiRestaker
- to the withdrawToLiquidityPool() function inside EtherFiRewardsRouter
- to the completeQueuedETHWithdrawals() function inside EtherFiNode

Recommendations: Modify withdrawEther to send the minimum of the contract's balance and the LiquidityPool's totalValueOutOfLp.

Customer's response: Fixed in commit 08aa248

Fix Review: Fixed – "Excess funds after totalValueOutOfLp may remain locked in EtherfiNode, EtherfiRewardRouter, EtherFiRestaker, and Liquifier and should be handled appropriately."

L-02 Missing Slashing Accounting in TVL Calculation

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Liquifier.sol	Status: Fixed	

Description: The getTotalPooledEtherSplits function calculates the value of restaked assets by directly converting the stored strategyShare to underlying tokens using info.strategy.sharesToUnderlyingView.

This approach is flawed for two reasons:

It relies on a static strategyShare variable stored in Liquifier, which does not reflect the actual state in EigenLayer.

Crucially, even if strategyShare were updated, this calculation fails to account for slashing. If the validator is slashed, the number of shares valid for withdrawal decreases.

The correct method to determine the value of restaked assets is to query the EigenLayer DelegationManager for getWithdrawableShares. This function returns the number of shares available to the staker, accounting for any slashing penalties that have occurred. Relying on the raw share count overestimates the protocol's TVL in the event of slashing.

Recommendations: Update the TVL calculation logic to query DelegationManager.getWithdrawableShares(address(this), strategies) to obtain the slashed-adjusted share amount before converting it to the underlying token amount.

Customer's response: Fixed in commit 6423174

Fix Review: Fixed

L-03 Incorrect Inclusion of Transferred Assets in TVL

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: Liquifier.sol	Status: Fixed	

Description: The getTotalPooledEther function in Liquifier.sol attempts to calculate the total value of the protocol by summing up balances and "restaked" amounts calculated via getTotalPooledEtherSplits.

However, the depositWithERC20 function transfers non-L2 tokens directly to the EtherFiRestaker contract:

Since the Liquifier contract does not hold these tokens and does not receive the strategy shares (which are minted to EtherFiRestaker), it is logically incorrect for Liquifier to attempt to report these as part of its own pooled Ether. The Liquifier has effectively become a pass-through for these assets, yet the view functions still attempt to perform accounting as if the assets were retained or tracked locally. This leads to confusion and potentially incorrect data if off-chain systems rely on Liquifier.getTotalPooledEther.

Recommendations: Remove the logic in Liquifier that attempts to calculate the value of assets held by EtherFiRestaker in case the Liquifier is not planned to hold restaked assets as well (which is not the case currently)

Customer's response: Fixed in commit 77381e3f

Fix Review: Fixed

L-04 Inaccurate TVL for Pending Redemptions due to Slashing

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: EtherFiRestaker.sol	Status: Acknowledged	

Description: The getAmountPendingForRedemption function calculates the value of stETH currently in the Lido withdrawal queue by summing the amountOfStETH of each request.

amountOfStETH represents the amount of tokens locked at the time of the request. It does not account for slashing penalties that may occur before finalization. If Lido validators are slashed, the actual ETH claimable for those shares will be lower than the original amountOfStETH. Using the face value of the locked tokens overestimates the protocol's assets in slashing scenarios.

Recommendations: Fixing this issue requires complicating the getTotalPooledEther() function to ensure proper reporting.

- For **unfinalized** requests, calculate the value based on the underlying shares and the current Lido exchange rate through getPooledEthByShares() (which reflects the latest slashing index).
- For **finalized** requests, use the claimable ETH amount via Lido getClaimableEther(), which requires providing a hint as input. The mental model is as follows:

```
JavaScript
1. if isFinalized -> getClaimableEther(reqId) -> this requires the precise hint to be inputed
-> after finalization amounts are not subject to slashing so getPooledEthByShares() would not
work
2. if NOT isFinalized -> lido.getPooledEthByShares(req.amountOfShares) -> latest slashing
```

Customer's response: Acknowledged - *"It's a very unlikely scenario that Lido's stETH gets slashed."*

Fix Review: Acknowledged

L-05 Loss Avoidance during V0 to V1 Migration

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MembershipManager.sol	Status: Acknowledged	

Description: The _migrateFromV0ToV1 function calculates the new V1 position based on the raw ETH principal amount recorded in V0 (tokenDeposits[_tokenId].amounts). It does not account for potential losses (slashing) that may have occurred in the underlying Liquidity Pool.

```javascript
uint128 amount = tokenDeposits[_tokenId].amounts;
// ...
uint256 eEthShare = liquidityPool.sharesForAmount(amount);
```

If the Liquidity Pool has suffered a loss, sharesForAmount(amount) will return more shares than the user effectively holds. By minting these new shares in the V1 system, the user is credited with the full original ETH value, effectively escaping the slashing loss. This loss is transferred to the remaining stakers in the protocol.

Recommendations: The whole V0 accounting follows this pattern of using the amounts, instead of the shares, so changing the logic here might be a very complicated effort, requiring changes throughout the whole contract.
The team should be aware of the behaviour in case of loss and potentially execute migration of all existing V0 positions, to ensure there is no exposure to the above edge case.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-06 Re-initialization Risk in initializeOnUpgrade		
Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MembershipManager.sol	Status: Acknowledged	

Description: The initializeOnUpgrade function sets important protocol parameters and addresses. While it is restricted to onlyOwner, it lacks a mechanism (like initializer or a boolean flag) to prevent it from being called multiple times. Accidental re-execution could overwrite the etherFiAdmin address or reset fanBoostThreshold and burnFeeWaiverPeriodInDays unexpectedly.

The same consideration applies to initializeOnUpgrade inside the MembershipNFT contract

Recommendations: Check that etherFiAdmin is not 0 to prevent secondary calls to the function.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-07 Zombie State Creation in claim

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MembershipManager.sol	Status: Acknowledged	

Description: The claim function does not verify if the tokenId exists or has been burned. If called on a burned/non-minted token (where tokenData is 0), it will execute _claimPoints, which sets token.prevPointsAccrualTimestamp to the current block timestamp and also set version to 1.

This effectively resurrects a "zombie" state for a non-existent token. While MembershipNFT tracks ownership, this dirty state in MembershipManager could lead to confusion or logic errors if other functions rely on token data

Recommendations: Add a check require(membershipNFT.exists(_tokenId)) or similar validation at the beginning of the claim function.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-08 rewardsGlobalIndex unnecessarily stored as vaultShares

Severity: **Low**	Impact: **High**	Likelihood: **High**
Files: MembershipManager.sol	Status: Acknowledged	

Description: Upon migration the token.vaultShare variable is set to rewardsGlobalIndex, which immediately after that is overwritten with the V1 vault shares. Currently it makes sense to completely remove this legacy update to vault, since it never gets used (since v0 always migrates to v1 after that). This also reduces the attack surface of potential accounting discrepancies

```
JavaScript
token.vaultShare = tierData[tier].rewardsGlobalIndex;
```

Recommendations: Remove the line that updates vaultShares to rewardsGlobalIndex

Customer's response: Acknowledged

Fix Review: Acknowledged

L-09 Broken EAP Calculation for Bronze Tier

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: MembershipNFT.sol	Status: Acknowledged	

Description: The computeTierPointsForEap function calculates points for Early Adopter Pool (EAP) users. It uses an array lastBlockNumbers where lastBlockNumbers[0] is explicitly set to 0.

If a user falls into the Bronze tier (tierId = 0), the code attempts to calculate the projection:

```
JavaScript
tierPoints += ((next - current) * (lastBlockNumbers[tierId] - _eapDepositBlockNumber)) / ...
```

Substituting tierId = 0:

```
JavaScript
(lastBlockNumbers[0] - _eapDepositBlockNumber) => (0 - _eapDepositBlockNumber)
```

Since _eapDepositBlockNumber is positive, this subtraction causes an integer underflow and reverts. This makes it impossible for any Bronze tier EAP user to migrate their points. Additionally, the denominator lastBlockNumbers[tierId] - lastBlockNumbers[nextTierId] would be 0 - 17664247, which also underflows.

Recommendations: Add a check to skip this calculation if tierId == 0, or adjust the logic to handle the Bronze tier correctly (likely by not projecting points backwards from block 0).

Customer's response: Acknowledged

Fix Review: Acknowledged

L-10 Reward index calculation does not factor losses for V0 positions

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: MembershipNFT.sol	Status: Acknowledged	

Description: The _V0_accruedStakingRewardsOf function does not handle scenarios where the protocol suffers a loss (negative rebase).

While MembershipNFT handles the decrease by returning 0 rewards (if (rewardsGlobalIndex > rewardsLocalIndex)), in GlobalIndexLibrary.calculateGlobalIndex. when a loss occurs, the library subtracts from the global index:

```
JavaScript
if (isLoss) {
    globalIndex[i] -= uint96(delta);
}
```

This however does not get reflected in the V0 position - for example when _claimStakingRewards is called _incrementTokenDeposit & _incrementTierDeposit will always retain the original amount and only increase it and when there are losses they will not get impacted. At the same time V1 positions will get their assets reduced, decreasing the amount of pooled shares.

Recommendations: The whole V0 accounting follows this pattern of using the amounts, instead of the shares, so changing the logic here might be a very complicated effort, requiring changes throughout the whole contract.

The team should be aware of the behaviour in case of loss and potentially execute migration of all existing V0 positions, to ensure there is no exposure to the above edge case.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-11 Hypothetical underflow risk in rebase during slashing events

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LiquidityPool.sol	Status: Acknowledged	

Description: The rebase function in LiquidityPool adjusts the protocol's Total Value Locked (TVL) by adding _accruedRewards to totalValueOutOfLp. This _accruedRewards value can be negative in the case of a slashing event. The current implementation performs the calculation int128(totalValueOutOfLp) + _accruedRewards and casts the result back to uint128.

If a slashing event occurs where the absolute value of the negative _accruedRewards is greater than the current totalValueOutOfLp, the result of the addition will be a negative number. Casting this negative number to an unsigned integer type (uint128) will cause it to underflow and wrap around to a very large positive value. This would permanently corrupt the totalValueOutOfLp accounting, breaking the protocol's TVL tracking and share price calculation.

Given the immense amount of funds accounted for in totalValueOutOfLp (in the billions) and the very low likelihood of slashing happening, the risk is very close to none. Still the impact of this edge case would be significant hence it makes sense to add a sanity check that would revert in case of underflow.

Recommendations: Make sure to check that _accruedRewards to never > totalValueOutOfLp and revert, which would give the team time to think how to handle the situation.

Customer's response: Acknowledged - *"It's very unlikely as mentioned, but important. So will include this check when moving the rebase logic from MembershipManager to LiquidityPool rather than making an upgrade just for the check."*

Fix Review: Acknowledged

L-12 Protocol Desynchronization via setReportStartSlot

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: EtherFiOracle.sol	Status: Acknowledged	

Description: The EtherFiOracle contract allows the admin to update the reportStartSlot via setReportStartSlot. This variable is used in blockStampForNextReport to determine the expected slotFrom for the next report if no reports have been published yet (lastPublishedReportRefSlot == 0).

However, the EtherFiAdmin contract calculates the expected start slot for processing (slotForNextReportToProcess) based solely on its own lastHandledReportRefSlot.

If setReportStartSlot is called to update the start slot (e.g., to skip slots or restart reporting), EtherFiOracle will generate reports with a refSlotFrom equal to the new reportStartSlot. EtherFiAdmin, expecting lastHandledReportRefSlot + 1, will reject these reports in executeTasks because the refSlotFrom does not match. This breaks the reporting pipeline, causing a Denial of Service.

Recommendations: Consider removing the setReportStartSlot() function from EtherFiOracle, since it makes sense to set the value of startSlot only once.

Customer's response: Acknowledged - *"That function is access controlled and will most likely never be called. Acknowledging the issue and can implement it in the next contract upgrade."*

Fix Review: Acknowledged

L-13 Broken Pausing Mechanism		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: StakingManager.sol	Status: Acknowledged	

Description: The StakingManager contract implements pauseContract and unPauseContract functions and inherits from PausableUpgradeable. However, the whenNotPaused modifier is not applied to any function within the contract.

As result the pausing()/unpausing() in EtherFiAdmin will not pause anything in that contract.

Recommendations: Double check if any functions should be pausable inside the StakingManager

Customer's response: Acknowledged - *"All functions are only called by LiquidityPool which has the whenNotPaused modifier. So, pausing LiquidityPool, pauses the functions. Other functions are access controlled by multisig so won't be an issue."*

Fix Review: Acknowledged

L-14 Treasury receives less than intended fee amount due to share value inflation after burning staker fee shares

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: EtherFiRedemptionManager.sol	Status: Fixed	

Description: The _calcRedemption function in EtherFiRedemptionManager calculates the treasury fee amount(eEthFeeAmountToTreasury) before burning staker fee shares. When feeShareToStakers is burned, the total share supply decreases while total assets remain constant, causing the value per share to increase. This means feeShareToTreasury shares have more value after the burn than the pre calculated eEthFeeAmountToTreasury. However, the contract only transfers the pre calculated eEthFeeAmountToTreasury to the treasury, leaving the additional value stuck in the contract. This results in:
1. Treasury receiving less than their entitled fee amount
2. Value accumulating in the contract that should have been sent to the treasury
3. Potential loss of protocol revenue over time

Recommendation: There are two solutions:

1. Burn the shares after sending eEthFeeAmountToTreasury to the treasury.
2. Recalculate eEthFeeAmountToTreasury based on the updated ratio and feeShareToTreasury.

Customer's response: Fixed in commit 6758805

Fix Review: Fixed

Informational Issues

I-01. Token Registration Bypass via updateWhitelistedToken in Liquifier

Description: The depositWithERC20 function in Liquifier checks if a token is whitelisted using isTokenWhitelisted. However, it does not explicitly check if the token has been properly initialized via registerToken.

The updateWhitelistedToken function allows an admin to whitelist any address. If a supported token (like lido stETH) is whitelisted via this function without being registered first, the tokenInfos struct for that token will remain uninitialized (default values).

While deposits will succeed (as quoteByMarketValue handles hardcoded token addresses), critical configuration parameters set during registration—such as the EigenLayer strategy address—will be missing.

Recommendation: Ensure that updateWhitelistedToken can only be called on tokens that have been previously registered, or add a check in depositWithERC20 to ensure the token's configuration (e.g., strategy address) is valid.

Customer's response: Acknowledged - *"We don't have any deposit tokens. It is intended to be used only for L2ETH. Hence, having a direct whitelisting is helpful."*

Fix Review: Acknowledged

I-02. Missing Events for State Changing Functions in Liquifier

Description: Most state-changing administrative functions in Liquifier.sol do not emit events. This includes updateTimeBoundCapRefreshInterval, updateDiscountInBasisPoints, updateWhitelistedToken, updateDepositCap, registerToken etc.. The lack of events makes it difficult for off-chain monitoring tools and indexers to track critical changes to the protocol configuration and asset movements.

Recommendation: Define and emit events for all state-changing functions.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-03. Redundant and Broken Token Pause Mechanism

Description: The pauseDeposits() in Liquifier function allows an admin to "pause" deposits for a specific token by setting its deposit caps to zero. This is redundant because the depositWithERC20 function is already protected by the whenNotPaused modifier, which enforces a global protocol pause.

Recommendation: Consider removing the redundant function

Customer's response: Fixed in commit 0b9f275

Fix Review: Fixed

I-04. Revert in stEthRequestWithdrawal due to Small Remainder

Description: The stEthRequestWithdrawal function splits large stETH withdrawal requests into smaller chunks to respect Lido's MAX_STETH_WITHDRAWAL_AMOUNT. However, the logic does not ensure that the final chunk respects Lido's MIN_STETH_WITHDRAWAL_AMOUNT.

If the requested _amount is slightly larger than a multiple of MAX_STETH_WITHDRAWAL_AMOUNT (e.g., MAX + 1 wei), the loop will create a final request for that small remainder (e.g., 1 wei). Lido's requestWithdrawals function reverts if any request amount is less than the minimum (typically small, but non-zero). This causes valid large withdrawal requests to fail.

Recommendation: Ensure the remainder is at least MIN_STETH_WITHDRAWAL_AMOUNT. If the remainder is too small, reduce the penultimate request slightly to increase the final request, or ensure _amount allows for a valid split.

Customer's response: Fixed in commit fedf838

Fix Review: Fixed

I-05. Pausing not used inside Restaker

Description: The EtherFiRestaker contract implements pauseContract and unPauseContract and inherits PausableUpgradeable. However, no functions in the contract utilize the whenNotPaused modifier. This means that calling pauseContract has no effect on the contract's operations, potentially giving admins a false sense of security during an emergency.

Recommendation: Apply the whenNotPaused modifier to critical state-changing functions.

Customer's response: Acknowledged - *"All functions are access controlled. Only transferStETH function is called by RedemptionManager which has a pausable modifier used."*

Fix Review: Acknowledged

I-06. Unused code

Description: Throughout the codebase there are lines of code that are redundant, because they are not used anywhere:

- EtherFiRestaker - Inside the stEthClaimWithdrawals, the first line that reads the ETH balance - link
- MembershipManager - Inside migrateFromV0ToV1 the final call to _migrateFromV0ToV1 is a redundant, since it was already called in claim() before that - link

Recommendation: Remove the redundant lines of code since they are not used anywhere.

Customer's response: Fixed in commit 328be4d

Fix Review: Fixed - *"The issue is only partially fixed, as the fix has not been applied in the Membership Manager."*

I-07. Discrepancy in totalRedeemableAmount() Calculation

Description: The totalRedeemableAmount function is intended to return the maximum amount of ETH (or stETH) that can be redeemed at the current moment. This calculation depends on two factors: the rate limit (bucket capacity) and the available liquidity in the pool.
The contract enforces a "low watermark" for liquidity—a minimum buffer that must be preserved. The canRedeem function correctly enforces this by checking if the available liquidity above the watermark is sufficient:

```
JavaScript
uint256 availableAmount = liquidEthAmount - lowWatermark;
if (availableAmount < amount) {
    return false;
}
```

However, the totalRedeemableAmount function calculates the limit as:

```
JavaScript
return Math.min(consumableAmount, liquidEthAmount);
```

It fails to subtract the lowWatermark from liquidEthAmount. As a result, it may report a redeemable amount that is higher than what is actually allowed by canRedeem. If a user or frontend relies on this value to maximize their redemption, the transaction will revert.

Recommendation: Update totalRedeemableAmount to subtract the low watermark from the liquid amount, ensuring consistency with canRedeem().

Customer's response: Fixed in commit 9f9a6ce

Fix Review: Fixed

I-08. Reentrancy in wrapEthForEap can manipulate emitted event values

Description: The wrapEthForEap function calls _mintMembershipNFT, which triggers the ERC1155 _mint function. This function invokes onERC1155Received on the recipient, passing control back to the caller before the transaction completes.

This allows the recipient to re-enter the membership contract and potentially change the return value of valueOf() before the subsequent _emitNftUpdateEvent is called.

Recommendation: If the event is used by offchain indexers, consider moving the emission before the NFT mint to make sure the value cannot be tampered with

Customer's response: Acknowledged

Fix Review: Acknowledged

I-09. Unfair Dilution in _topUpDeposit

Description: When a user tops up their deposit, if the new total exceeds the penalty threshold, the contract dilutes their baseTierPoints. The current formula dilutes the points based on the ratio of the entire new deposit to the old deposit.

This means a user topping up just slightly above the threshold suffers a dilution proportional to their entire capital addition, which may be excessively punitive compared to penalizing only the amount exceeding the threshold.

Recommendation: Review the dilution formula to ensure it aligns with the intended economic incentives. Consider diluting based only on the "excess" capital (amount above maxDepositWithoutPenalty)

Customer's response: Acknowledged

Fix Review: Acknowledged

I-10. Unused Parameter in deposit function

Description: The deposit(address _user, address _referral) function is called by the MembershipManager contract. It accepts a _user parameter but its value is never used within the function body. The subsequent internal call to _deposit credits the minted shares to msg.sender (which is the MembershipManager contract), not the _user address. This can be misleading for developers and integrators reading the code.

Recommendation: Remove the unused _user parameter from the function signature to improve code clarity and prevent confusion.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-11. Redundant State Variable and Function

Description: The contract includes a state variable restakeBnftDeposits and a corresponding setter function setRestakeBnftDeposits. However, the restakeBnftDeposits variable is never read or used anywhere in the contract's logic. This constitutes dead code that adds unnecessary contract size.

Recommendation: Remove the setRestakeBnftDeposits function

Customer's response: Acknowledged - "There are a lot of variables to deprecate in the Liquidity Pool. Will deprecate this one later along with the others."

Fix Review: Acknowledged

I-12. Mismatch in EigenPod_validatorPubkeyHashToInfo Implementation

Description: The function EigenPod_validatorPubkeyHashToInfo is named to suggest it performs a lookup using validator public key hashes. However, its signature accepts full public keys (bytes[][]), and its implementation calls validatorPubkeyToInfo on the EigenPod contract. This logic is identical to the subsequent function EigenPod_validatorPubkeyToInfo, making it redundant and misleading.

It appears the intention was for this function to accept bytes32 hashes and query the validatorPubkeyHashToInfo mapping directly on the EigenPod, which would be more gas-efficient for off-chain viewers that already possess the hashes.

Recommendation: Update the function signature to accept bytes32[][] memory _validatorPubkeyHashes and modify the implementation to call validatorPubkeyHashToInfo on the EigenPod contract.

Customer's response: Fixed in commit 677668d

Fix Review: Fixed

I-13. Inconsistent Access Control Mechanism in EETH

Description: Both the EETH::mintShares and EETH::burnShares functions are restricted to being called only by the liquidity pool contract. However, mintShares uses the modifier in order to achieve that, while burnShares uses an inline check inside its function body for that, which is unnecessary, given the presence of the onlyPoolContract modifier.

Recommendation: Consider using the onlyPoolContract modifier in both the mintShares and burnShares functions.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-14. Missing Sanity Check in _handleAccruedRewards

Description: In EtherFiAdmin._handleAccruedRewards, the contract calculates the time elapsed since the last report to verify that the APR is within acceptable bounds.

While the protocol logic generally implies that _report.refSlotTo should be greater than lastHandledReportRefSlot, there is no explicit check enforcing this. If a malformed report were to pass validation where refSlotTo < lastHandledReportRefSlot, elapsedSlots would be negative, leading to a negative elapsedTime and potentially incorrect APR calculations or underflows.

Recommendation: Add a sanity check

Customer's response: Acknowledged

Fix Review: Acknowledged

I-15. Unused State Variable in MembershipManager

Description: In the current state of the MembershipManager contract, the treasury state variable isn't being read or used anywhere. This is redundant, adds unnecessary complexity and makes the contract storage bigger than it should be.

Recommendation: Consider removing the treasury state variable from the MembershipManager contract.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-16. Redundant Boolean Equation Check

Description: In the processDepositFromEapUser function of the MembershipNFT contract there is the following validation check, that ensures that a given user hasn't yet executed an EAP deposit.

```
JavaScript
function processDepositFromEapUser(address _user, uint32  _eapDepositBlockNumber, uint256
_snapshotEthAmount, uint256 _points, bytes32[] calldata _merkleProof)
onlyMembershipManagerContract external {
        if (eapDepositProcessed[_user] == true) revert InvalidEAPRollover();
```

As it can be seen, the value of the user inside the eapDepositProcessed mapping is being taken and then checked whether it's equal to true or not. This however is redundant, as boolean values don't need to be checked for being true or false, as this results in a boolean value in itself.

The same redundancy can also be found in EtherFiOracle.sol#L232;L241;L251 and RegulationsManager.sol#L68.

Recommendation:
Consider simplifying the aforementioned boolean check

```
JavaScript
function processDepositFromEapUser(address _user, uint32  _eapDepositBlockNumber, uint256
_snapshotEthAmount, uint256 _points, bytes32[] calldata _merkleProof)
onlyMembershipManagerContract external {
-       if (eapDepositProcessed[_user] == true) revert InvalidEAPRollover();
+       if (eapDepositProcessed[_user]) revert InvalidEAPRollover();
```

Customer's response: Acknowledged

Fix Review: Acknowledged

I-17. Incorrect Burn Amount Reported in Event Due to amountForShare Mismatch

Description: The handleRemainder function in WithdrawRequestNFT emits an incorrect amount in the HandledRemainderOfClaimedWithdrawRequests event. The event uses liquidityPool.amountForShare(eEthSharesToBurn) instead of the actual eEthAmountToBurn value. After burning eEthSharesToBurn, the value of eEthSharesToBurn increases, causing the function to return an amount greater than eEthAmountToBurn.

Recommendation: Use eEthAmountToBurn instead of liquidityPool.amountForShare(eEthSharesToBurn).

Customer's response: Fixed in commit 3c6379d

Fix Review: Fixed

I-18. Broken domain separator caching in proxy pattern

Description: The EETH contract implements a caching mechanism for the EIP712 domain separator to optimize gas costs, but the cache check is fundamentally broken when the contract is used through a UUPS proxy pattern. The contract stores _CACHED_THIS in the constructor when the implementation contract is deployed:

However, when the contract is called through a proxy via delegatecall, the address(this) in the implementation code refers to the proxy's address, not the implementation's address. This creates a permanent mismatch:
- _CACHED_THIS = Implementation contract address(set during deployment)
- address(this) at runtime = Proxy contract address(when called through proxy)

The cached domain separator(_CACHED_DOMAIN_SEPARATOR) is never returned because the condition address(this) == _CACHED_THIS always evaluates to false when called through a proxy.

Recommendation: Use the proxy address for the caching instead of the implementation contract one

Customer's response: Acknowledged

Fix Review: Acknowledged

I-19. Deposit Cap inconsistency in Liquifier Contract

Description: The deposit cap check in Liquifier uses the dx value instead of the original amount. It checks the cap against dx(discounted value after discount) instead of the market value (before discount).

- Deposit caps should limit the market value of tokens deposited
- Currently, the code checks the cap against dx (discounted value)
- This means deposits can exceed the cap

Recommendation: Use market value instead of dx when checking for deposit cap.

Customer's response: Acknowledged

Fix Review: Acknowledged

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.