certora

Security Assessment

ether.fi – Core Contracts Combined Audit Report

February-March 2026

Prepared for ether.fi

certora

ether.fi

Table of contents

Project Summary

Project Scope

Project Name	Initial Commit Hash	Latest Commit Hash	Platform	Start Date	End Date
Priority Withdrawal Queue	79386	3b6b81b	EVM	05/02/2026	13/02/2026
weETH support for Priority Withdrawals	7fc5100	7fc5100	EVM	05/03/2026	05/03/2026

Project Overview

This document describes the manual code review of several modules and changes to the core contracts repository.

The work was a 6 day effort undertaken between **05/02/2026** and **05/03/2026**

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

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	2	2	2
Medium	2	2	2
Low	3	3	3
Informational	11	11	6
Total	18	18	13

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
Priority Withdrawal Queue			
H-01	Incorrect Share Accounting in _cancelWithdrawRequest Breaks Protocol Solvency	High	Fixed
H-02	Accounting Mismatch in ethAmountLockedForPriorityWithdrawal Due to Share Price Fluctuations	High	Fixed
M-01	PriorityWithdrawalQueue Ignores Liquidity Pool Withdrawal Locks	Medium	Fixed
M-02	Remainder Shares Are Not Tracked in Cancelled Requests	Medium	Fixed
L-01	Incorrect Event Argument in RemainderHandled	Low	Fixed
L-02	Non-Idempotent CREATE2 Deploy Can DoS Deployment Script	Low	Fixed
L-03	requestWithdrawWithPermit Can Be Griefed via Permit Frontrun	Low	Fixed
weETH support for Priority Withdrawals			
-	-	-	-

Priority Withdrawal Queue

Project Overview

This report presents the findings of a manual code review for the **Priority Withdrawal Queue** audit within the **EtherFi** core contracts. The work was undertaken from **February 5th to February 13th 2026**

The following contract list is included in the scope of this audit:

- `src/EtherFiRedemptionManager.sol`
- `src/LiquidityPool.sol`
- `src/PriorityWithdrawalQueue.sol`
- `script/upgrades/CrossPodApproval/transactions.s.sol`
- `script/upgrades/priority-queue/transactionsPriorityQueue.s.sol`

The code modifications examined during this review were introduced in the following pull request - PR#356.

High Severity Issues

H-01 Incorrect Share Accounting in _cancelWithdrawRequest Breaks Protocol Solvency

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: PriorityWithdrawalQueue.sol	Status: Fixed	

Description: The `_cancelWithdrawRequest` function in `PriorityWithdrawalQueue` unconditionally transfers the originally requested `amountOfEEth` back to the user without reconciling the share value at the time of cancellation.

```
function _cancelWithdrawRequest(WithdrawRequest calldata request) internal returns (bytes32
requestId) {
    // ... [dequeuing logic]

    // Issue: Unconditionally transfers fixed token amount.
    // If share price dropped (slashing), this requires MORE shares than
request.shareOfEEth.
    IERC20(address(eETH)).safeTransfer(request.user, request.amountOfEEth);

    // ...
}
```

When a withdrawal is requested, the user transfers eETH to the contract. Since eETH is a rebasing token, the contract holds shares. If the share price changes (due to rewards or slashing) between the request creation and cancellation:

1. **Slashing (Price Drop):** The fixed `amountOfEEth` represents more shares than originally received. Transferring this amount transfers more shares than the user provided, essentially taking from other withdrawers.
2. **Rewards (Price Rise):** The fixed `amountOfEEth` represents fewer shares. The contract retains the excess shares, which become unclaimable yield.

This behavior contrasts with `_claimWithdraw`, which correctly calculates the withdrawal amount

based on the minimum of the requested amount and the current value of the shares, tracking any remainder.

Recommendations: Align `_cancelWithdrawRequest` with `_claimWithdraw` by returning `min(request.amountOfEEth,liquidityPool.amountForShare(request.shareOfEEth))` instead of always returning `request.amountOfEEth`.

Customer's response: Fixed in commit 288d12b

Fix Review: Fixed - "*The system now returns the full shares that were deposited with the request. This creates some low probability and non-critical griefing scenarios explained in a separate issue - I-10*"

H-02 Accounting Mismatch in ethAmountLockedForPriorityWithdrawal Due to Share Price Fluctuations

Severity: **High**	Impact: **High**	Likelihood: **High**
Files: PriorityWithdrawalQueue.sol	Status: Fixed	

Description: The `ethAmountLockedForPriorityWithdrawal` state variable tracks the ETH required to fulfill finalized priority withdrawal requests. However, the logic for increasing and decreasing this variable uses the eETH share price at two different points in time, leading to a permanent drift in the accounting.

1. At finalization (`fulfillRequests`): the contract locks ETH using the share price at that moment (amountForShare(shareOfEEth)), increasing ethAmountLockedForPriorityWithdrawal.
2. Later, at claim/cancel: it unlocks using a fresh, current-price computation (`min(amountOfEEth, amountForShare(shareOfEEth))`), decreasing the same variable by a different basis.

Because the share price changes between fulfillment and claiming, the amount subtracted will rarely match the amount added. This results in `ethAmountLockedForPriorityWithdrawal` becoming inaccurate over time, potentially blocking valid withdrawals or reserved-liquidity accounting to become stale or incorrect.

Recommendations: Record a per-request locked amount at `fulfillRequests` (e.g., `lockedAmountPerRequest[requestId]=min(amountOfEEth, amountForShare(shareOfEEth))`) and increase `ethAmountLockedForPriorityWithdrawal` by that exact value. On claim/cancel, take the minimum payout by both current share value and this stored lock (and the original requested amountOfEEth), but always decrement the global lock by the stored per-request amount and clear it.

Customer's response: Fixed in commit 288d12b

Fix Review: Fixed - "*The system always reserves the amountOfEEth calculated at request creation and always reduces it by that amount*"

Medium Severity Issues

M-01 PriorityWithdrawalQueue Ignores Liquidity Pool Withdrawal Locks

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

Description: The `PriorityWithdrawalQueue` contract withdraws ETH from the `LiquidityPool` via the `withdraw()` function. The `LiquidityPool.withdraw` function contains a check to ensure that `ethAmountLockedForWithdrawal` (funds reserved for NFT withdrawals) is preserved, but this check is conditional on `msg.sender == address(withdrawRequestNFT)`.

```
if (totalValueInLp < _amount || (msg.sender == address(withdrawRequestNFT) &&
ethAmountLockedForWithdrawal < _amount) || ...)
```

When `PriorityWithdrawalQueue` calls `withdraw`, `msg.sender` is the queue contract, bypassing the check against `ethAmountLockedForWithdrawal`. Consequently, the Priority Queue can withdraw ETH that was supposed to be reserved for standard NFT withdrawal requests, potentially causing the standard withdrawal queue to become insolvent or delayed.

Recommendations: Update `LiquidityPool.withdraw` to enforce that the available liquidity for `PriorityWithdrawalQueue` (and other callers) respects the `ethAmountLockedForWithdrawal` reserved for the NFT queue. Specifically, ensure `totalValueInLp - ethAmountLockedForWithdrawal >= _amount` for non-NFT callers.

In addition to that the NFT withdrawals could be considered to also be restricted by the locked priority funds depending on the business requirements of the feature.

Customer's response: Fixed in commit 3b6b81

Fix Review: Fixed – "

The priority of withdraws after the update is as follows:

1. NFT withdraws & Priority queue withdraws have the same priority and respect each other's locked amounts. All other withdrawals are limited by them.

2. etherFiRedemptionManager comes last – it can withdraw amounts past `ethAmountLockedForWithdrawal` + `ethAmountLockedForPriorityWithdrawal`

3. *Membership contract is not expected to be used any more and will be removed*

"

M-02 Remainder Shares Are Not Tracked in Cancelled Requests

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

Description: When processing a withdrawal claim in `_claimWithdraw`, the contract calculates the difference between the shares backing the request and the shares actually burned for the withdrawal amount. This difference is added to `totalRemainderShares` to be later swept to the treasury or burned. However, `_cancelWithdrawRequest` lacks this logic. If the amount transferred back to the user corresponds to fewer shares than `request.shareOfEEth` (due to share price increase), the remaining shares are left in the contract balance but are not added to `totalRemainderShares`. As a result, these shares become stuck and cannot be managed via `handleRemainder`.

Recommendations: Update `_cancelWithdrawRequest` to calculate the number of shares actually consumed by the transfer and add any excess from `request.shareOfEEth` to `totalRemainderShares`.

Customer's response: Fixed in commit 288d12b

Fix Review: Fixed

Low Severity Issues

L-01 Incorrect Event Argument in RemainderHandled

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

Description: The `RemainderHandled` event in `PriorityWithdrawalQueue.sol` is defined as `event RemainderHandled(uint96 amountToTreasury, uint96 sharesOfEEthToBurn)`. However, when emitted, the second argument passed is `liquidityPool.amountForShare(eEthSharesToBurn)`, which is an ETH amount, not a share count. This discrepancy can confuse off-chain indexers and monitoring tools.

Recommendations: Update the event emission to pass `eEthSharesToBurn` directly, or rename the event parameter to reflect that it logs an ETH amount.

Customer's response: Fixed in commit c5cc902

Fix Review: Fixed

certora

ether.fi

L-02 Non-Idempotent CREATE2 Deploy Can DoS Deployment Script

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: deployPriorityQueue.s.sol	Status: Fixed	

Description: `deployPriorityQueue.s.sol` uses the permissionless `CREATE2` factory via `Utils.deploy` without checking whether the predicted address already has code. If the contract was already deployed (e.g., script rerun, partial run, or a third party deployed the same bytecode+salt through the same factory), `factory.deploy(...)` reverts and the deployment script halts.

Recommendations: Make deployments idempotent by checking `predictedAddress.code.length` before calling `factory.deploy`; if code exists, return/log the predicted address (optionally sanity-check it) instead of reverting. Implementing this inside `Utils.deploy` fixes it once for all CREATE2-based scripts.

Customer's response: Fixed in commit 3e61412

Fix Review: Fixed

L-03 requestWithdrawWithPermit Can Be Griefed via Permit Frontrun

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

Description: `requestWithdrawWithPermit` is vulnerable to permit frontrunning grief: it always reverts if `eETH.permit()` fails and does not check whether allowance is already sufficient. Because permit execution is permissionless, a third party can submit the same signed permit first and consume the nonce, causing the user's bundled withdrawal transaction to fail in catch.

Recommendations: In the `catch` block, only revert if `eETH.allowance(msg.sender, address(this)) < amountOfEEth;` otherwise continue with `safeTransferFrom`.

Customer's response: Fixed in commit bfb5db

Fix Review: Fixed

Informational Issues

I-01. Rounding Loss in Treasury Split Calculation

Description: In `PriorityWithdrawalQueue.handleRemainder`, the calculation for `eEthAmountToTreasury` uses standard solidity integer division (floor), which systematically rounds down against the protocol's favor. While the loss is dust (wei), consistently rounding down results in a slight under-allocation to the treasury (and, thus, over-burn) over time compared to the intended basis points split.

Recommendation: Consider using `Math.ceilDiv` or standard rounding-up logic for the treasury split, or acknowledge the behavior as acceptable.

Customer's response: Fixed in commit 8d299e1

Fix Review: Fixed

I-02. Asymmetric Withdrawal Value Logic for Users

Description: Upon `requestWithdraw` the user is specifying `amountOfEEth` and transfers this amount of eETH to the queue. But the transfer is just modifying the shares of eETH and the calculation is rounding down. So, even if the user has actually specified `amountOfEEth` to transfer, the actual eETH amount that the queue took is `amountForShare(shareForAmount(amountOfEEth))` . This is the eETH amount that the queue has after the request of the withdrawal. However, the `WithdrawRequest` struct is storing the `amountOfEETH` which may be inflated by 1 wei and thus not 100% correct.

Recommendation: This is likely a design choice but still an important behaviour detail that the team should be aware of.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-03. Unused OwnableUpgradeable Inheritance

Description: The `PriorityWithdrawalQueue` contract inherits from `OwnableUpgradeable` and initializes it, but does not utilize the `onlyOwner` modifier or ownership functionality for access control. It relies entirely on `RoleRegistry` for permissions. This adds unnecessary bytecode and storage overhead.

Recommendation: Remove `OwnableUpgradeable` inheritance and initialization if `RoleRegistry` is the intended access control mechanism.

Customer's response: Fixed in commit c44e471

Fix Review: Fixed

I-04. Inaccurate NatSpec in invalidateRequests

Description: The NatSpec for `invalidateRequests` states that it "(prevents finalization)". However, the function logic allows it to target both pending (non-finalized) and already finalized requests. If targeting a finalized request, it effectively prevents the user from claiming their funds, which is a stronger action than just preventing finalization.

Recommendation: Update the documentation to accurately reflect that this function cancels requests regardless of their state (pending or finalized).

Customer's response: Fixed in commit 8d2111e

Fix Review: Fixed

I-05. Batch Claim Bypasses User Balance Check

Description: The `claimWithdraw` function includes a post-condition check `_verifyClaimPostConditions` which ensures the user's ETH balance has increased. However, `batchClaimWithdraw` uses `_verifyBatchClaimPostConditions` which checks aggregate pool balances but skips the per-user balance increase check. A caller could theoretically bypass the per-user safety check by wrapping a single claim in a batch call.

Recommendation: Ensure consistent safety checks between single and batch operations if the per-user balance check is deemed critical for safety.

Customer's response: Fixed in commit f11cdd6

Fix Review: Fixed

I-06. Potential Gas Griefing in claimWithdraw

Description: The `claimWithdraw` functions send ETH directly to `request.user`. If `request.user` is a smart contract with expensive receive logic, it can consume a significant amount of gas. This allows a user to grief third-party callers (e.g., keepers or other users fulfilling batches) who pay the gas for the transfer.

Recommendation: Document that `claimWithdraw` forwards all gas when sending ETH to `request.user`, so third parties should avoid claiming/batching for untrusted recipients.

Customer's response: Fixed in commit 4afb260

Fix Review: Fixed

I-07. Unused Return Values in Internal Functions

Description: The internal functions `_dequeueWithdrawRequest` and `_queueWithdrawRequest` return values (`requestId` and `WithdrawRequest memory req`) that are not utilized by the calling functions in the current codebase.

Recommendation: Remove the unused return values to clean up the code and potentially save a small amount of gas.

Customer's response: Acknowledged

Fix Review: Acknowledged

I-08. Deployment/Upgrade Scripts Print Inaccurate Logs

Description: `deployPriorityQueue.s.sol` prints a `dryRunWithFork()` command in `dryRun()`, but that function does not exist, so the suggested invocation is wrong. Also, the "NEXT STEPS" suggests initializing the PriorityWithdrawalQueue proxy even though it is already initialized during proxy deployment (Step 2 passes initialize() calldata). Separately, `executeUpgrade()` prints "Transaction Details" that omit the `EtherFiRedemptionManager` upgrade even though it is included in the scheduled batch.

Recommendation: Update the log strings to match actual functions and actions.

Customer's response: Fixed in commit cebb37

Fix Review: Fixed

I-09. Priority Queue Scripts Contain Unresolved TODOs and Placeholders

Description: `deployPriorityQueue.s.sol` and `transactionsPriorityQueue.s.sol` include TODOs/placeholder constants (e.g., `commitHashSalt`, deployed implementation/proxy addresses, and config notes) that must be finalized before running against mainnet. Leaving these unresolved can lead to deploying with the wrong salt/config or generating upgrade transactions targeting incorrect addresses.

Recommendation: Replace all TODO-marked fields with the final production values.

Customer's response: Acknowledged - "*Yes, we will take care of the addresses.*"

Fix Review: Acknowledged

I-10. Finalized Request Cancellation Creates Remainder-Bypass Optionality

Description: A finalized priority withdrawal request can be canceled by the user, returning the current share value in eETH rather than the capped claim amount. During positive rebase periods, this allows the user to avoid the claim cap (where excess would become `totalRemainderShares`) by canceling and redeeming via EtherFiRedemptionManager, provided redemption constraints pass and exit fee is lower than the rebase premium. This is primarily an economic/theoretical inconsistency rather than a direct funds-loss exploit.

To demonstrate it better, consider this scenario:

1. User requests 1.00 eETH in `PriorityWithdrawalQueue`, and later the request is finalized (`ethAmountLockedForPriorityWithdrawal += 1.00`).
2. A positive rebase happens, so the request's stored shares are now worth 1.05 eETH.
3. If user claims, payout is capped at 1.00 ETH and the excess value is handled as remainder logic.
4. If user instead cancels, they receive current share value (~1.05 eETH) back.
5. User redeems that 1.05 eETH via `EtherFiRedemptionManager`; if exit fee is below 0.05, this route yields more than direct claim.

Recommendation: Document this behavior and its economic implications: finalized-request cancellation returns current share value, so in some market conditions users may prefer cancel+redeem over claim.

Customer's response: Acknowledged - "*These are whitelisted users and any malicious behaviour will be observed and penalized.*"

Fix Review: Acknowledged

I-11. WithdrawRequestNFT Lock Accounting Can Retain Unconsumed ethAmountLockedForWithdrawal

Description: In `WithdrawRequestNFT` flows, `EtherFiAdmin` locks liquidity at finalization (`finaliseWithdrawalAmount`), but each NFT claim later withdraws `min(request.amountOfEEth, current share value) - fee`. If share value drops between finalization and claim (e.g., negative rebase/slash), claims can consume less than locked, leaving residual `ethAmountLockedForWithdrawal` and reducing available LP liquidity.

Recommendation: Document this `WithdrawRequestNFT` behavior clearly, and ensure off-chain oracle/reporting operations reconcile `finalizedWithdrawalAmount` against realized claims so residual `ethAmountLockedForWithdrawal` does not persist unnecessarily.

Customer's response: Acknowledged

Fix Review: Acknowledged

weETH support for Priority Withdrawals

Project Overview

This report presents the findings of a manual code review for the **weETH support for Priority Withdrawals** audit within the **EtherFi** core contracts. The work was undertaken on **March 5th 2026**

The following contract list is included in the scope of this audit:

- `src/PriorityWithdrawalQueue.sol`

The code modifications examined during this review were introduced in the following pull request - PR#374.

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.