SOLIDIFIED

Audit Report for ether.fi - October 26, 2023

Summary

Audit Report prepared by Solidified covering ether.fi's LSD solution.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on October 6, 2023 and the results are presented here.

Audited Files

The source code has been supplied in the following source code repository:

Repo: https://github.com/GadzeFinance/dappContracts
Commit hash: `56c67eff7d313d389ff612c0f0b2c41ae60e9b7a`
Additional reviewed commits:

- `f9c4927829efe9cbff1122ddd7b04ce3aa9916cf`
- `61be7a5805aedad6d4e3f49e4e878472110a9979`
- `66cd3ef84248663490eaef4b44e1799445aae641`
- `c2a042ad4c7de1894c1ac8984bb6c0b4bb16737c`
- `3994d2c18d1434adee7d75a124a926dfcff1d096`
- `a647686213a3b4051bf758059d727ec42d54fb90`

```
src
├── AuctionManager.sol
├── EETH.sol
├── EtherFiAdmin.sol
├── EtherFiNode.sol
├── EtherFiNodesManager.sol
├── EtherFiOracle.sol
├── LiquidityPool.sol
├── MembershipManager.sol
├── MembershipNFT.sol
├── StakingManager.sol
├── WithdrawRequestNFT.sol
└── libraries
    └── GlobalIndexLibrary.sol
```

(Only the functions `processAuctionFeeTransfer` and `transferAccumulatedRevenue` from `AuctionManager.sol` were in scope)

Intended Behavior

The code base implements a liquid staking protocol that allows stakers to keep control of their keys.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium-High	There are a lot of external calls between the different contracts that need to be authorized and that pass important data such as the original caller as a function argument.
Code readability and clarity	Medium	-
Level of Documentation	Medium	-
Test Coverage	Medium-High	-

Issues Found

Solidified found that the ether.fi contracts contain 3 critical issues, 2 major issues, 15 minor issues, and 5 informational notes.

We recommend issues are amended, while informational notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Anyone can call StakingManager.batchCancelDepositAsBnftHolder to cancel a deposit	Critical	Resolved
2	A BNFT Holder can cancel a deposit by bypassing the liquidity pool and receive the entire stake	Critical	Resolved
3	Funds of recycled EtherFiNodes can be stolen	Critical	Resolved
4	EtherFiOracle will not be able to reach consensus under some circumstances	Major	Resolved
5	The liquidity pool owner can redeem fully withdrawn, or slashed TNFTs for 30 ether	Major	Resolved
6	Dust of eETH could be stuck in the WithdrawRequestNFT contract	Minor	Resolved
7	Reliance on hard-coded gas number for ETH transfers can be problematic	Minor	Resolved
8	LiquidityPool's TNFT rewards and principal could be redirected to the treasury	Minor	Acknowledged
9	Partial withdrawals can be avoided by malicious users	Minor	Resolved
10	Error in WithdrawRequestNFT.requestWithdraw validation logic	Minor	Resolved
11	EtherFiOracle.numActiveCommitteeMembers	Minor	Resolved

	can return wrong values		
12	Any MembershipManager admin could rebase using an arbitrary amount of accrued rewards.	Minor	Resolved
13	MembershipManager.addNewTier uses the wrong length	Minor	Resolved
14	MembershipManager functions use LiquidityPool's eETH deposit flow	Minor	Resolved
15	Misleading SourceOfFunds emitted in events	Minor	Resolved
16	numPendingDeposits is not updated on unsuccessful deposits	Minor	Resolved
17	Centralization Issues	Minor	Partially Resolved
18	Non exit penalty daily rate cannot be updated to a value greater than 1%	Minor	Resolved
19	EtherFiNodesManager does not verify that staking rewards split sums up to 100%	Minor	Resolved
20	Burn fees from MembershipManager.unwrapForEEthAndBurn stay in the MembershipManager contract decreasing rewards for EtherFan holders.	Minor	Resolved
21	console.sol is imported in various files	Note	Acknowledged
22	Hardcoded values	Note	Resolved
23	LiquidityPool.requestWithdraw checks against wrong upper limit	Note	Resolved
24	Incorrect report finalization check in EtherFiOracle.verifyReport	Note	Resolved
25	Any LiquidityPool admin could increase ethAmountLockedForWithdrawal	Note	Resolved

Critical Issues

1. Anyone can call StakingManager.batchCancelDepositAsBnftHolder to cancel a deposit

The function `StakingManager.batchCancelDepositAsBnftHolder` is usually called by the liquidity pool and passes `msg.sender` for the `_caller` argument in this flow. This argument should be the “address of the bNFT holder who initiated the transaction” and is “used for verification” according to the docs. However, the function is public with no access control. A malicious user can call it and pass in the address of the corresponding BNFT holder. This will cancel the deposit of this user. The function `_cancelDeposit` then performs the following operation:

```
_refundDeposit(msg.sender, 31 ether);
```

It does not use the address in `_caller` for the refund, but the address of the user that has performed the call (i.e., the attacker in this scenario). This leads to a loss of funds for the BNFT holder.

Recommendation

Add access control to the function and only allow the liquidity pool to call it.

2. A BNFT Holder can cancel a deposit by bypassing the liquidity pool and receive the entire stake

When making a deposit a BNFT holder calls `LiquidityPool.batchDepositAsBnftHolder` and deposits `2 ether` which is paired with `30 ether` from the liquidity pool to generate the `32 ether` needed to spin up a validator node which is then sent to the `StakingManager` contract. When canceling a deposit the BNFT holder calls `LiquidityPool.batchCancelDeposit` and

will be refunded either the full `2 ether` or `1 ether` depending on whether the node has already been registered or not. However, in this process the `LiquidityPool` contract will get back its `30 ether` from the `StakingManager` contract.
A BNFT holder can avoid this process by calling `StakingManager.batchCancelDeposit` and bypass the `LiquidityPool` altogether. This will result in the `StakingManager` contract refunding the full stake to the BNFT holder directly instead of refunding the `LiquidityPool` contract first which then issues the appropriate refund to the BNFT holder. A sample scenario would be where a malicious BNFT holder deposits `12 ether` to spin up 6 nodes and have `180 ether` provided by the `LiquidityPool` contract. Then they could cancel the deposit immediately afterwards by calling `StakingManager.batchCancelDeposit` and have the `StakingManager` contract refund them their `12 ether` plus the `180 ether` from the `LiquidityPool`.

Recommendation
Remove the `batchCancelDeposit` function from the `StakingManager` contract, therefore only allowing deposit cancellations via the `LiquidityPool`.

3. Funds of recycled EtherFiNodes can be stolen

When a node is fully withdrawn using `EtherFiNodesManager.fullWithdraw` the `EtherFiNode` is recycled, if the `totalBalanceInExecutionLayer` is 0. However, this does not reset `etherfiNodeAddress[_validatorId]` to `address(0)`, which could still point to the `EtherFiNode`. Thus, when the node is reused, an attacker could use the old validator ID to partially withdraw rewards or fully withdraw the node's balance upon exit, distributing funds to the previous BNFT & TNFT holders and node operator, instead of the legitimate owners. The previous TNFT holder could also send an exit request on the `EtherFiNode`, using the old validator ID.
Note that if the total balance in the execution layer upon full withdrawal is 0, the admin will not be able to prevent this using `EtherFiNodesManager.resetWithdrawalSafes` (which would set `etherfiNodeAddress[_validatorId]` to `address(0)`, since `IEtherFiNode(etherfiNode).resetWithdrawalSafe()` sets the phase to `READY_FOR_DEPOSIT` and can only be executed once.

Recommendation

Set `etherfiNodeAddress[_validatorId] = address(0)` after pushing the node to `unusedWithdrawalSafes` in `EtherFiNodesManager.fullWithdraw`.

Major Issues

4. `EtherFiOracle` will not be able to reach consensus under some circumstances

The function `EtherFiOracle.submitReport` checks if the condition `consenState.support == quorumSize` is `true` whenever a report is submitted. However, the value of `quorumSize` can be modified by an owner with the function `setQuorumSize`. This can become very problematic when the quorum size is reduced and set to a value that is smaller than the current number of votes.

For instance, assume that `consenState.support` is 6 and `quorumSize` is 7. The owner now reduces the `quorumSize` to 5. While the quorum was reached, the report will never be published, because `consenState.support == quorumSize` will not be true, even if more submissions come in.

Recommendation

Consider implementing a function that can be called to publish these reports (by checking if `consenState.support <= quorumSize`).

5. The liquidity pool owner can redeem fully withdrawn, or slashed TNFTs for 30 ether

The function `LiquidityPool.swapTnftForEth` allows the owner of the liquidity pool to swap any TNFT for `30 ether`, regardless of its principal value or if it has been fully withdrawn and the TNFT holder already received back their stake.

Recommendation
Consider removing the functionality, or ensure any swapped TNFTs belong to an operational validator and are redeemed to the principal value (taking into account any slashing that might have occurred).

Minor Issues

6. Dust of eETH could be stuck in the WithdrawRequestNFT contract

When `eETH` holders request withdrawal using `LiquidityPool.requestWithdraw`, or `EtherFan` holders request withdrawal using `LiquidityPool.requestMembershipNFTWithdraw`, the shares corresponding to the ether amount are calculated and sent to the `WithdrawRequestNFT` contract. Upon `WithdrawRequestNFT.claimWithdraw`, the amount is recalculated and the smaller of the request or the current amount is sent to the user. The share is recalculated in `LiquidityPool.withdraw`, rounded up, and burnt from the `WithdrawRequestNFT` contract. This could be problematic, because if the value of `eETH` is altered between the time of request and claim of withdrawal, excess `eETH` will be stuck in the `WithdrawRequestNFT` contract.

Recommendation
Consider burning the originally calculated share of the request in `LiquidityPool.withdraw`.

7. Reliance on hard-coded gas number for ETH transfers can be problematic

---

The function `EtherFiNode.withdrawFunds` has hard-coded gas limits for the ETH transfers to the different users. While setting a limit in this function can be useful (to prevent DoS attacks by malicious actors), the hard-coded values can be problematic. For instance, the transfer to the BNFT holder uses a limit of 2300. This can be too low for smart contracts that emit events or access state when receiving ETH. As smart contract wallets are getting more popular, the user may not be able to control the logic of the `fallback` / `receive` function and they may not be aware of the problem.

Recommendation
Reconsider the chosen values and consider making them configurable (for instance, when there is an update in the future that changes the gas usage).

8. LiquidityPool's TNFT rewards and principal could be redirected to the treasury

---

In scenarios where the `totalValueOutOfLp` is less than the rewards accrued, withdrawing a node's balance that belongs to a TNFT in the liquidity pool might revert due to underflow in `LiquidityPool`'s `receive` function. This could happen if the withdrawal occurs before rebasing and would effectively redirect the funds to the treasury, disadvantaging `eETH` and `EtherFan` holders. Unless the treasury sends the ether back to the LP, `totalValueOutOfLp` after rebasing will be increased by the expected rewards but those will be at the discretion of the treasury - essentially artificially inflating the value of `eETH`.

Recommendation
Consider tracking when this occurs and storing the amount in the treasury. Then, anyone should be allowed to initiate the transfer back to the liquidity pool (after the rebasing), guaranteeing that the pool will receive these funds in the future.

9. Partial withdrawals can be avoided by malicious users

Unlike the `EtherFiNode.withdrawFunds` function, `EtherFiNodesManager.partialWithdrawBatchGroupByOperator` does not set any limits when sending ether to the different roles. Therefore, a malicious actor could prevent partial withdrawals by using up all the remaining gas.

Recommendation
Consider using the same logic for sending ether when partial withdrawals are performed.

10. Error in `WithdrawRequestNFT.requestWithdraw` validation logic

The function `requestWithdraw` requires that `tokenId <= nextRequestId` and returns the error "Request does not exist" if this is not the case. However, `nextRequestId` is not minted yet, so when the token ID is equal to `nextRequestId`, this check should also fail. The impact of this off-by-one error is limited because there is an `ownerOf` check afterwards which will fail for an unminted token.

Recommendation
Use `<` instead of `<=` in the comparison.

11. `EtherFiOracle.numActiveCommitteeMembers` can return wrong values

The variable `numActiveCommitteeMembers` keeps track of the number of active validators in `EtherFiOracle`. It is properly decremented when a committee member is disabled using `manageCommitteeMember`. However, it is also possible to remove an active committee member

with the function `removeCommitteeMember`. In this case, only `numCommitteeMembers` is decremented.

Recommendation
Check if the member is active and decrement `numActiveCommitteeMembers` if so in `removeCommitteeMember`.

12. Any `MembershipManager` admin could rebase using an arbitrary amount of accrued rewards.

The function `MembershipManager.rebase` should only be called from `EtherFiAdmin` based on the report from the `EtherFiOracle` to ensure the integrity of value of `eETH`. However, any admin could rebase using an arbitrary amount of accrued rewards.

Recommendation
Consider restricting the access control of rebase to only allow it to be executed from the `EtherFiAdmin` contract.

13. `MembershipManager.addNewTier` uses the wrong length

The function `addNewTier` in `MembershipManager` performs a length check against `tierDeposits.length` (to impose a maximum length) and returns `tierDeposits.length - 1`. However, this array is never modified in the function, it modifies `tierData` instead.

Recommendation
Replace `tierDeposits.length` with `tierData.length`.

14. MembershipManager functions use LiquidityPool's eETH deposit flow

The `MembershipManager`'s functions `wrapEthForEap`, and `topUpDepositWithEth` use `LiquidityPool`'s deposit function that takes only `_referral` as a parameter. The aforementioned function is intended for `eETH` staking flow, and the function `LiquidityPool.deposit(_user, _referral)` should have been used instead. The incorrect use results in misleading events as the `SourceOfFunds` is set to `EETH` instead of `ETHER_FAN`, bypasses the whitelist for the users, and would revert if the `MembershipContract` address is not whitelisted (as a user) in the `LiquidityPool`.

Recommendation
Consider using `LiquidityPool.deposit(_user, _referral)` function instead, for deposits relating to `ETHER_FAN` staking flow.

15. Misleading SourceOfFunds emitted in events

Both the `eETH` and `ETHER_FAN` withdrawals are handled by the protocol by the `WithdrawRequestNFT` contract, which calls `LiquidityPool.withdraw`. The withdraw function attempts to distinguish the `SourceOfFunds` and emit the information by using `msg.sender` to see if it is the `MembershipManager` or the `WithdrawRequestNFT` contract. However since all the withdrawals are handled via `WithdrawRequestNFT`, `SourceOfFunds.EETH` is always emitted.

Recommendation
If monitoring the source of funds off-chain is important, consider storing it in the `WithdrawRequestNFT` and emitting it according to the request ID.

16. numPendingDeposits is not updated on unsuccessful deposits

---

`LiquidityPool.batchDepositAsBnftHolder` increases `numPendingDeposits` by the `_numberOfValidators`, however if not all the validators get deposited successfully, the variable is not modified to reflect the actual number of successful deposits awaiting registration.

Recommendation

Decrease the number of pending deposits by `_numberOfValidators - newValidators.length` if not all deposits are successful.

17. Centralization Issues

Many aspects of the protocol are centralized and are subject to the discretion of the EtherFi team:

- Admin can set the loyalty and tier points of each `MembershipNFT` to arbitrary values at any time.
- `MembershipManager.withdrawFees` allows any of the contract admins to withdraw its ether balance to an arbitrary account. This includes significant revenues for stakeholders like mint / burn / upgrade fees of `MembershipNFTs` and auction bids fees from `AuctionManager`, which can be withdrawn instead of distributed.
- During rebasing, only the `boostThreshold` amount is sent to the `LiquidityPool`. This amount can be set to 0 at any time, essentially trimming out all the extra rewards for `ETHER_FAN` holders.
- `EtherFiOracle.quorumSize` could be set too low, exercising centralized control over the protocol.
- `WithdrawRequests` can be invalidated by the `EtherFiOracle`.
- `nonExitPenaltyDailyRate` can be set up to 100% per day, up to `1 ether` total.
- The owners of the contracts could update the following dependencies:
    - `WithdrawRequestNFT` can set the `LiquidityPool`, `EETH`, and `MembershipManager` to any addresses.
    - `MembershipNFT` can set the `LiquidityPool`, and `MembershipManager` to any addresses.

- `LiquidityPool` owner can set `EETH`, `EtherFiNodesManager`, `StakingManager`, `TNFT`, `EtherFiAdmin`, and `WithdrawRequestNFT` to any addresses.
- `StakingManager` can set the `depositContractEth2` to any address, which could result in irreversible loss of funds for the users.

Recommendation
Consider enforcing hard lower and upper limits to variables like `quorumSize`, `boostThreshold`, `nonExitPenaltyDailyRate`, ensure dependencies can only be set once, and remove any unnecessary points of centralization.

18. Non exit penalty daily rate cannot be updated to a value greater than 1%

In the pull request 1439, `nonExitPenaltyDailyRate` was changed from percentages to basis points. However, the setter function `setNonExitPenaltyDailyRate` within `EtherFiNodesManager` was not updated and still enforces a maximum value of 100, which is now 1%. Therefore, it will not be possible to update the rate to a value that is greater than 1%.

Recommendation
Update the upper boundary within the setter function to use basis points instead of percentages.

19. `EtherFiNodesManager` does not verify that staking rewards split sums up to 100%

While the comment for the function `setStakingRewardsSplit` states that “Splits must add up to the SCALE of 1_000_000”, this check was removed in pull request 1445. It is therefore now possible to set staking rewards split that do not sum up to a value that is smaller or greater than 100%. Both would be problematic, as it would either lead to a loss of funds (for a value that is too small) or failing withdrawals (for a value that is too great).

Recommendation

Enforce that the splits sum up to 1,000,000 to avoid mistakes by the administrator.

20. Burn fees from `MembershipManager.unwrapForEEthAndBurn` stay in the `MembershipManager` contract decreasing rewards for EtherFan holders.

The function `unwrapForEETHAndBurn`, burns an `EtherFan` NFT for the corresponding amount of `eETH`. If the burn fee waiver period has not been met, the `MembershipManager` transfers the share of `eETH` to the user, subtracting the fee, while the fee is left as an excess in the `MembershipManager`. While this does not negatively affect the value of `eETH` as the total shares remain the same, the surplus of `eETH` in the contract accumulates rewards. During rebase, ETH in the `MembershipManager`, accrued from fees, is deposited to the liquidity pool, and the minted `eETH` shares are divided amongst the total supply to be split amongst the vaults. The surplus of `eETH` in the contract would slightly decrease the amount distributed to each vault, and subsequently, NFT holders' boosted rewards.

Recommendation
Remove the burn fee for wrapping to `eETH`, alternatively, burn the excess `eETH` to boost its value, or send it to the treasury.

Informational Notes

21. `console.sol` is imported in various files

Various files (`MembershipNFT.sol`, `MembershipManager.sol`, `EtherFiAdmin.sol`, `EtherFiOracle.sol`, `LiquidityPool.sol`, `GlobalIndexLibrary.sol`) import `forge-std/console.sol`.

Recommendation

Remove these imports before the deployment.

22. Hardcoded values

The function `EtherFiNodeManager.fullWithdraw` uses a hardcoded value for the max number of withdrawals to be claimed if staking is enabled, instead of the `maxEigenlayerWithrawals` variable that exists for this purpose and can be modified by the owner.
Also, the function `EtherFiNode.setReportStartSlot` uses a hardcoded value for the slots per epoch, instead of the constant `SLOTS_PER_EPOCH` that denotes the number of slots in one epoch (32).

Recommendation

Consider using the corresponding variables and constants in those functions, to avoid issues when the values are changed in the future.

23. `LiquidityPool.requestWithdraw` checks against wrong upper limit

`LiquidityPool.requestWithdraw` performs the following check:
`if (amount > type(uint128).max || amount == 0 || share == 0) revert InvalidAmount();`
However, the variable `amount` is afterwards cast to a `uint96`, not a `uint128`. This would only lead to problems if the user has an `eETH` balance that is larger than `type(uint96).max`, which is highly improbable. Nevertheless, it is recommended to use the correct upper limits for the input sanitization.

Recommendation

Revert when the `amount` is larger than `type(uint96).max`.

24. Incorrect report finalization check in `EtherFiOracle.verifyReport`

The comment in line `130` of `EtherFiOracle.sol` specifies that a report is considered finalized at `current_epoch - 2`. However, the check in line `134` is `require(reportEpoch + 2 < currentEpoch)`. This means that a report is finalized while it is less than `current_epoch - 2` and not finalized when it equals `current_epoch - 2` which contradicts the comment. The same issue exists in line `145` in the `_isFinalized` function.

Recommendation

Either update the check to `require(reportEpoch + 2 <= currentEpoch)` or clarify the comment.

25. Any `LiquidityPool` admin could increase `ethAmountLockedForWithdrawal`

The function `LiquidityPool.addEthAmountLockedForWithdrawal` should only be called from `EtherFiAdmin` based on the report from the `EtherFiOracle`. However, any admin could call the function inflating the amount of ETH locked for withdrawals.

Recommendation

Consider restricting the access control of `addEthAmountLockedForWithdrawl` to only allow it to be executed from the `EtherFiAdmin` contract.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Gadze Finance SEZC or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Oak Security GmbH