Zellic

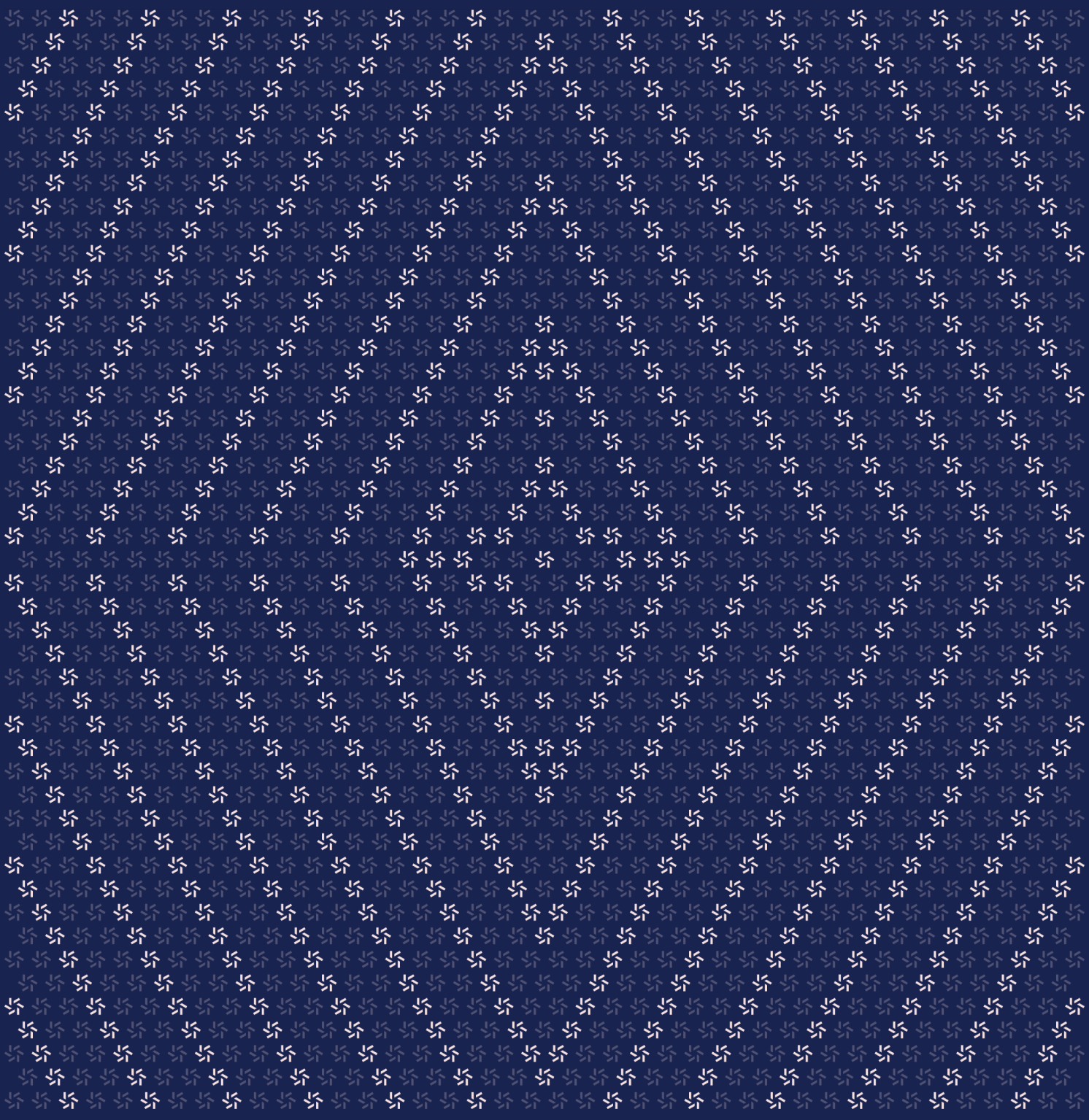
Prepared for
Mike Silagadze
EtherFi

Prepared by
Nipun Gupta
Seunghyeon Kim
Zellic

March 13, 2024

EtherFi

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for EtherFi from February 29th to March 11th, 2024. During this engagement, Zellic reviewed EtherFi's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the version migration disrupt the functionality of contracts, rendering users incapable of withdrawing their staked ETH?
- Are withdrawals, both partial and full, functioning as intended?
- Is the reward-distribution mechanism working as intended?
- Can malicious users exploit vulnerabilities to access rewards or the staked ETH of other users?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

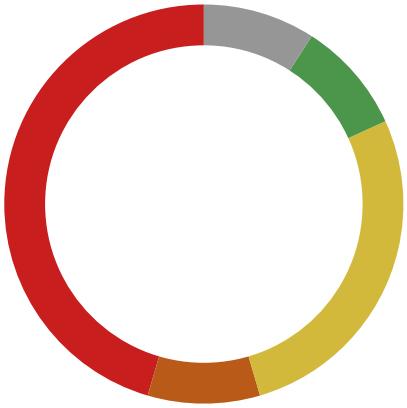
1.4. Results

During our assessment on the scoped EtherFi contracts, we discovered 11 findings. Five critical issues were found. One was of high impact, three were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for EtherFi's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	5
High	1
Medium	3
Low	1
Informational	1



2. Introduction

2.1. About EtherFi

EtherFi contributed the following description of EtherFi:

> EtherFi is a decentralized, non-custodial liquid restaking protocol built on Ethereum, allowing users to stake their Ethereum and participate in the DeFI ecosystem without losing liquidity. The protocols eETH is a liquid restaking token, serving as a representation of ETH staked on the Beacon Chain, which rebases daily to reflect the associated staking rewards.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

EtherFi Contracts

Repository	https://github.com/etherfi-protocol/smart-contracts ↗
Version	smart-contracts: `4ab08c973b51455397ebcaaeb75ebab0e985d08c`
Programs	• EtherFiNodesManager • EtherFiNode • LiquidityPool • StakingManager • TNFT
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of eight calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io

Seunghyeon Kim
Engineer
seungyhyeon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 29, 2024	Start of primary review period
March 5, 2024	Kick-off call
March 11, 2024	End of primary review period

3. Detailed Findings

3.1. Repeated validator IDs could be passed in `batchRevertExitRequest`

Target	EtherFiNodesManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

The function `batchRevertExitRequest` does not check if the `_validatorIds` passed to it are repeated nor does it check if an exit request has been previously requested for this validator ID. A malicious TNFT holder could use the same `_validatorIds` twice in an array, which would call `IEtherFiNode(etherfiNode).updateNumExitRequests(0, 1);` twice and would decrease the value of `numExitRequestsByTnft` by an unexpected amount. Or they could call `batchRevertExitRequest` with `_validatorIds` for which the exit request has not been previously requested, which would also decrease the value of `numExitRequestsByTnft`.

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

        require (msg.sender == tnft.ownerOf(_validatorId), "NOT_TNFT_OWNER");

        _updateEtherFiNode(_validatorId);

        IEtherFiNode(etherfiNode).updateNumExitRequests(0, 1);
        validatorInfos[_validatorId].exitRequestTimestamp = 0;

        emit NodeExitRequestReverted(_validatorId);
    }
}
```

Impact

An attacker could call this numerous times to decrease the value of `numExitRequestsByTnft` to 0. Later, if a validator ID is unregistered, the call would revert due to an underflow in the following line if the validator is exited using an exit request:

```
if (_info.exitRequestTimestamp != 0) numExitRequestsByTnft -= 1;
```

Recommendations

We recommend adding checks to ensure `batchRevertExitRequest` could only be called on a validator ID if an exit request has been previously requested for that ID.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit a15c35fd ↗.

3.2. Repeated validator IDs could be passed in `batchSendExitRequest`

Target	EtherFiNodesManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

The function `batchSendExitRequest` does not check if the `_validatorIds` passed to it are repeated. A malicious TNFT holder could use the same `_validatorIds` twice in an array, which would call `IEtherFiNode(etherfiNode).updateNumExitRequests(1, 0);` twice and would increase the value of `numExitRequestsByTnft` by an unexpected amount.

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

        require (msg.sender == tnft.ownerOf(_validatorId), "NOT_TNFT_OWNER");
        require (phase(_validatorId) == IEtherFiNode.VALIDATOR_PHASE.LIVE,
    "NOT_LIVE");

        _updateEtherFiNode(_validatorId);

        IEtherFiNode(etherfiNode).updateNumExitRequests(1, 0);
        validatorInfos[_validatorId].exitRequestTimestamp
    = uint32(block.timestamp);

        emit NodeExitRequested(_validatorId);
    }
}
```

Impact

The attacker can increase the number of exit requests by repeating the same `_validatorId`. The function `_getTotalRewardsPayoutsFromSafe`, which is called to calculate the total rewards payout would revert if `numExitRequestsByTnft` is nonzero. An attacker could thus increase the value of `numExitRequestsByTnft` by using repeated validator IDs, which would make it impossible to claim the staking rewards.

Recommendations

We recommend reverting the function if there are repeated validator IDs.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit a15c35fd.

3.3. BNFT holder could cancel the deposit after `processNodeExit` is called

Target	EtherFiNodesManager		
Category	Business Logic	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

Once the node's exit and funds withdrawal from Beacon is observed, the protocol calls `processNodeExit` to process their exits. This function updates the phase of the validator to `EXITED`. To fully withdraw the funds, the function `fullWithdraw` could be called, which burns the two NFTs and distributes the payouts to all the entities. However, a BNFT holder could call `batchCancelDeposit` on the validator ID, which would transfer the BNFT holder 2 ETH, assuming there is enough ETH in the liquidity pool and `numPendingDeposits` is greater than one. This call to `batchCancelDeposit` would change the phase of the validator from `EXITED` to `FULLY_WITHDRAWN` and would delete the `etherfiNodeAddress` mapping.

Later, if `fullWithdraw` is called on that validator ID, the call would revert.

Impact

A BNFT holder could get the entire 2 ETH even after penalty and could block the withdrawals of other entities for that validator ID.

Recommendations

The underlying issue was due to the function `_unRegisterValidator`, which handled the phase transition for both canceled deposits and withdrawals. We recommend separating these phase transitions for these different functionalities.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `04ba6239` ↗.

3.4. Malicious users could use the ETH of legitimate users and mint themselves NFTs

Target	StakingManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

There are two source of funds to deposit ETH in the contract: `DELEGATED_STAKING` and `EETH`. In `DELEGATED_STAKING`, the user could deposit the entire 32 ETH using the payable function `batchDepositWithBidIds` in StakingManager, which changes the phase of the validator ID to `STAKE_DEPOSITED`, and then call `batchRegisterValidators`, which registers the validator, mints them TNFT and BNFT, and changes the phase to `LIVE`. While in the case of `EETH`, the tokens are used from the liquidity pool and the phase is changed to `WAITING_FOR_APPROVAL`.

The two functions `batchDepositWithBidIds` and `batchRegisterValidators` are not required to be executed in the same transaction. Therefore, when the function `batchDepositWithBidIds` is called with the source of funds as `DELEGATED_STAKING`, the 32 ETH are stored in the StakingManager contract, which are later used in the `batchRegisterValidators` call. An attacker who has already called `batchDepositAsBnftHolder`, which internally calls `batchDepositWithBidIds` with the source of funds as `EETH`, could directly call the nonpayable `batchRegisterValidators` to get access to these 32 ETH, which are deposited by a legitimate user. The bug arises because an attacker could use these 32 ETH before the legitimate user calls `batchRegisterValidators` as this function does not check the source of these funds.

Impact

The malicious BNFT holder could get access to the ETH deposited by a legitimate user.

Recommendations

We recommend checking the source of funds in `batchRegisterValidators`.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `b2c2c31f` ↗.

3.5. ETH sent to wrong address on cancellation

Target	LiquidityPool		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

This bug was independently discovered by the EtherFi team and they presented us the fix which we reviewed. Here are the details of the issue:

When `isLpBnftHolder` is true, the LiquidityPool is the bnft holder. In this case, if the deposit is cancelled, the 2 ETH belonging to the LiquidityPool(which is also the bnft holder), shouldn't be sent out to the `_bnftStaker`, but instead should remain in the LiquidityPool.

Impact

ETH are sent to the wrong address when LiquidityPool is the bnft holder.

Recommendations

The tokens shouldn't be sent to the `_bnftStaker` in case `isLpBnftHolder` is true.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `3d2b1037` ↗.

3.6. Wrong rewards calculation due to `numAssociatedValidators`

Target	EtherFiNode, EtherFiNodesManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

When a user deposits stakes via either StakingManager or LiquidityPool, the value of `_numAssociatedValidators` will increase. After the user deposits the phase would be `IEtherFiNode.VALIDATOR_PHASE.STAKE_DEPOSITED`, after register is called on the validator the next phase of the validator id could be `WAITING_FOR_APPROVAL` if deposit is made via the LiquidityPool. If the approval is rejected the phase goes back to `NOT_INITIALIZED`.

Even if the validator ID could be rejected after the deposit is made, the rewards are divided by `numAssociatedValidators` in the `getRewardsPayouts` function. Therefore, the rewards payouts could be incorrectly calculated. The same issue could also affect the return value of `calculateTVL` as it calculates `stakingRewardsInEL` using `numAssociatedValidators`.

Impact

The reward calculation might be incorrectly calculated.

Recommendations

We recommend increasing the value of `numAssociatedValidators` only when the phase of the validator is `LIVE`.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `43a0f8e9` ↗.

3.7. The BNFT holder is compared with an incorrect address

Target	StakingManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `batchDepositWithLiquidityPoolAsBnftHolder` could be called in the LiquidityPool contract when `isLpBnftHolder` is true. If `isLpBnftHolder` is true, the LiquidityPool is expected to be the owner of the new BNFT and the BNFT of the validator ID to share the safe with.

But internally in the `_processDeposit` function, it checks if the owner of the BNFT with ID `_validatorIdToShareWithdrawalSafe` is equal to the `_staker` instead of LiquidityPool.

Impact

The function `batchDepositWithLiquidityPoolAsBnftHolder` would revert if a nonzero `_validatorIdToShareWithdrawalSafe` is passed to it because of incorrect owner check.

Recommendations

We recommend passing the address of LiquidityPool as the BNFT holder and comparing the owner of `_validatorIdToShareWithdrawalSafe` with this address.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `010bd4b7` ↗.

3.8. Queued withdrawals are not claimed by `forcePartialWithdraw`

Target	EtherFiNodesManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `forcePartialWithdraw` could be called by an admin to force a partial withdrawal. This function internally calls `_getTotalRewardsPayoutsFromSafe`, which in turn calls `getRewardsPayouts` in the etherFiNode manager, which calls `withdrawableBalanceInExecutionLayer` to calculate the rewards to be distributed. If restaking is enabled, the function `withdrawableBalanceInExecutionLayer` would add all the claimable withdrawals from the `delayedWithdrawalRouter` to the current balance of the etherFiNode.

The function `forcePartialWithdraw` does not claim the queued withdrawals first, but the queued withdrawals are added to be distributed to the different entities. As the function does not claim these queued withdrawals first, there would not be enough tokens in the etherFiNode contract to be distributed amongst the entities, and this call would revert.

Impact

The function `forcePartialWithdraw` would revert in certain cases.

Recommendations

We recommend calling `claimQueuedWithdrawals` to claim the queued withdrawals in `forcePartialWithdraw`.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit 3c2a6b50 ↗.

3.9. Deposit cancellation may fail if the etherFiNode version is not updated

Target	EtherFiNodesManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The functions `batchCancelDeposit` and `batchCancelDepositAsBnftHolder` do not call `_updateEtherFiNode` to update the etherFiNode version. These functions internally call `unRegisterValidator` in etherFiNode, which has a modifier `ensureLatestVersion` to ensure that the version of the etherFiNode is not zero, but as `_updateEtherFiNode` is never called, the version would be zero, and this call would revert.

Impact

The functions `batchCancelDeposit` and `batchCancelDepositAsBnftHolder` might revert if the version is still zero.

Recommendations

We recommend calling `_updateEtherFiNode` in the `batchCancelDeposit` / `batchCancelDepositAsBnftHolder` functions or internally before `EtherFiNode.sol:unRegisterValidator` is called.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit 405b37cc ↗.

3.10. Reward and withdrawal payout getters might fail if the etherFiNode version is not updated

Target	EtherFiNode		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

If version of etherFiNode is zero, the functions `EtherFiNodeManager.sol:getRewardsPayouts` and `EtherFiNodeManager.sol:getFullWithdrawalPayouts` will revert. This is because the modifier `ensureLatestVersion` ensures that the version of etherFiNode is nonzero.

Impact

The functions `getRewardsPayouts` and `getFullWithdrawalPayouts` will revert in certain cases.

Recommendations

We recommend removing `ensureLatestVersion` for `getRewardsPayouts` and `getFullWithdrawal-Payouts` functions.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit `405b37cc` ↗.

3.11. Admin could not revert exit request on behalf of LiquidityPool

Target	LiquidityPool		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is a `sendExitRequests` function to send exit requests when the TNFT holder is LiquidityPool, which calls `batchSendExitRequest`, but there is no function to call `batchRevertExitRequest` when the TNFT holder is LiquidityPool.

Impact

There is no function to call `batchRevertExitRequest` on behalf of LiquidityPool.

Recommendations

We recommend adding a function to call `batchRevertExitRequest` on behalf of LiquidityPool.

Remediation

This issue has been acknowledged by EtherFi, and a fix was implemented in commit 405b37cc ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Unused variables could be removed

There are two instances of variables that are declared but not used:

1. In the `batchCancelDepositAsBnftHolder` function, the variables `numberOfEethValidators` and `numberOfEtherFanValidators` are declared, and the value of these variables are increased depending upon the source of funds, but they are not used anywhere.

```
function batchCancelDepositAsBnftHolder(uint256[] calldata _validatorIds,
    address _caller) public whenNotPaused nonReentrant {
    require(msg.sender == liquidityPoolContract, "INCORRECT_CALLER");

    uint32 numberOfEethValidators;
    uint32 numberOfEtherFanValidators;
    for (uint256 x; x < _validatorIds.length; ++x) {
        ILiquidityPool.SourceOfFunds source
    = bidIdToStakerInfo[_validatorIds[x]].sourceOfFund;
        require(source != ILiquidityPool.SourceOfFunds.DELEGATED_STAKING,
    "Wrong flow");

        if (source == ILiquidityPool.SourceOfFunds.EETH){
            numberOfEethValidators++;
        } else if (source == ILiquidityPool.SourceOfFunds.ETHER_FAN) {
            numberOfEtherFanValidators++;
        }

        if(nodesManager.phase(_validatorIds[x]) ==
    IEtherFiNode.VALIDATOR_PHASE.WAITING_FOR_APPROVAL) {
            uint256 nftTokenId = _validatorIds[x];
            TNFTInterfaceInstance.burnFromCancelBNftFlow(nftTokenId);
            BNFTInterfaceInstance.burnFromCancelBNftFlow(nftTokenId);
        }

        _cancelDeposit(_validatorIds[x], _caller);
    }
}
```

2. In the `_cancelDeposit` function, the variable `validatorPhase` is set as the phase of the

validator ID but is not used anywhere.

```
function _cancelDeposit(uint256 _validatorId, address _caller) internal {
    require(bidIdToStakerInfo[_validatorId].staker == _caller,
    "INCORRECT_CALLER");

    IEtherFiNode.VALIDATOR_PHASE validatorPhase
    = nodesManager.phase(_validatorId);

    bidIdToStakerInfo[_validatorId].staker = address(0);
    nodesManager.unregisterValidator(_validatorId);

    // Call function in auction contract to re-initiate the bid that won
    auctionManager.reEnterAuction(_validatorId);

    bool isFullStake = (msg.sender != liquidityPoolContract);
    if (isFullStake) {
        _refundDeposit(msg.sender, stakeAmount);
    }

    emit DepositCancelled(_validatorId);
}
```

We recommend removing the unused variables to save gas.

This issue has been acknowledged by EtherFi, and fixes were implemented in the following commits:

- 405b37cc ↗
- 04ba6239 ↗

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: EtherFiNodesManager.sol

Function: `batchFullWithdraw(uint256[] _validatorIds)`

The function processes the full withdrawal for multiple validators.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: No specific constraints.
  - **Impact**: Validator IDs for which full withdrawals need to be executed.

Branches and code coverage

Intended branches

- Updates the etherFiNode for the validator ID.
  - ☑ Test coverage
- Sets the validator phase to `FULLY_WITHDRAWN`.
  - ☑ Test coverage
- The function will unregister the validators, distribute the payouts, and burn the TNFT and BNFT related to the validator ID.
  - ☑ Test coverage

Negative behavior

- Revert if there are pending withdrawals left.
  - ☑ Negative test
- Revert if the phase is not `EXITED`.
  - ☑ Negative test

Function call analysis

- `this.fullWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).version()`

- **What is controllable?** Not controllable.
- **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the version of etherFiNode.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the EtherFiNode or an unexpected state.

- `this.fullWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitRequestTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit-request timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).migrateVersion(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the EtherFiNode version related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with migrating the version — no reentrancy scenario.
- `this.fullWithdraw(_validatorIds[i]) -> IEtherFiNode(etherfiNode).claimQueuedWithdrawals(this.maxEigenlayerWithdrawals, true)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Claims queued withdrawals from the EtherFiNode related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with claiming queued withdrawals — no reentrancy scenario.
- `this.fullWithdraw(_validatorIds[i]) -> this.getFullWithdrawalPayouts(_validatorId) -> IEtherFiNode(etherfiNode).getFullWithdrawalPayouts(this.getValidatorInfo(_validatorId), this.stakingRewardsSplit)`

- **What is controllable?** `_validatorId`.
- **If the return value is controllable, how is it used and how can it go wrong?** Calculates full withdrawal payouts and retrieves full withdrawal payouts from the EtherFiNode based on validator info and staking rewards split.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with calculating full withdrawal payouts — no reentrancy scenario.

- `this.fullWithdraw(_validatorIds[i]) -> this._unRegisterValidator(_validatorId) -> this._setValidatorPhase(safeAddress, _validatorId, VALIDATOR_PHASE.FULLY_WITHDRAWN) -> IEtherFiNode(_node).validatePhaseTransition(this.phase(_validatorId), _newPhase)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Sets the validator phase to fully withdrawn and validates the phase transition on the EtherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue the phase transition — no reentrancy scenario.
- `this.fullWithdraw(_validatorIds[i]) -> this._unRegisterValidator(_validatorId) -> IEtherFiNode(safeAddress).unRegisterValidator(_validatorId, this.validatorInfos[_validatorId])`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Unregisters the validator on the EtherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with unregistering the validator on the EtherFiNode — no reentrancy scenario.
- `this.fullWithdraw(_validatorIds[i]) -> this._unRegisterValidator(_validatorId) -> IEtherFiNode(safeAddress).numAssociatedValidators()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the number of associated validators from the EtherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> IEtherFiNode(_etherfiNode).withdrawFunds(this.treasuryContract, _toTreasury, this.auctionManager.getBidOwner(_validatorId), _toOperator, this.tnft.ownerOf(_validatorId), _toTnft, this.bnft.ownerOf(_validatorId), _toBnft)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Distributes payouts from the EtherFiNode to relevant parties.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the reward distribution — no reentrancy scenario.

- `this.fullWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.auctionManager.getBidOwner(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the bid owner from the auction manager.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.tnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the TNFT owner.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.bnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the BNFT owner.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this.tnft.burnFromWithdrawal (_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Burns the TNFT from withdrawal.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.fullWithdraw(_validatorIds[i]) -> this.bnft.burnFromWithdrawal (_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Burns the BNFT from withdrawal.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `batchPartialWithdrawOptimized(uint256[] _validatorIds)`

Optimized version of `batchPartialWithdraw`.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: No specific constraints.

- **Impact**: Validator IDs for which optimized partial withdrawals need to be executed.

Branches and code coverage

Intended branches

- Updates the etherFiNode related to each validator ID.
  - ☑ Test coverage
- Distributes the rewards to all the entities.
  - ☑ Test coverage

Negative behavior

- Revert if the owners of TNFT, BNFT, and and the node operator are not consistent among the provided validator IDs.
  - ☐ Negative test

Function call analysis

- `this.auctionManager.getBidOwner(_validatorIds[0])`
  - **What is controllable?** `_validatorIds[0]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the bid owner from the auction manager related to the first validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.tnft.ownerOf(_validatorIds[0])`
  - **What is controllable?** `_validatorIds[0]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the TNFT related to the first validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.bnft.ownerOf(_validatorIds[0])`
  - **What is controllable?** `_validatorIds[0]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the BNFT related to the first validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.auctionManager.getBidOwner(_validatorId)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the bid owner from the auction manager related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.tnft.ownerOf(_validatorId)`
  - **What is controllable?** Not controllable.

  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the TNFT related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.bnft.ownerOf(_validatorId)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the BNFT related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).version()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the version of etherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the EtherFiNode or an unexpected state.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitRequestTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit-request timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).migrateVersion(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the EtherFiNode version related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with migrating the version — no reentrancy scenario.
- `IEtherFiNode(etherfiNode).claimQueuedWithdrawals (this.maxEigenlayerWithdrawals, False)`

- **What is controllable?** Not controllable.
- **If the return value is controllable, how is it used and how can it go wrong?** Claims queued withdrawals from the EtherFiNode related to each validator ID.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with claiming queued withdrawals — no reentrancy scenario.

- `this._getTotalRewardsPayoutsFromSafe(_validatorId, True) -> IEtherFiNode(etherfiNode).numExitRequestsByTnft()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the number of exit requests by TNFT from the EtherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._getTotalRewardsPayoutsFromSafe(_validatorId, True) -> IEtherFiNode(etherfiNode).getRewardsPayouts(this.validatorInfos [_validatorId].exitRequestTimestamp, this.stakingRewardsSplit)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Calculates total rewards payouts from the safe related to each validator ID.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with calculating rewards payouts — no reentrancy scenario.
- `IEtherFiNode(etherfiNode).moveFundsToManager(total)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Moves funds from the EtherFiNode related to each validator ID to the manager.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with moving funds to the manager — no reentrancy scenario.

Function: `batchPartialWithdraw(uint256[] _validatorIds)`

Calls `partialWithdraw` for multiple validator IDs.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: No specific constraints.
  - **Impact**: Validator IDs for which partial withdrawals need to be executed.

Branches and code coverage

Intended branches

- Calls `this.partialWithdraw(_validatorIds[i])` for each validator ID.
  - ☑ Test coverage
- The `partialWithdraw` function updates the EtherFiNode for the validator ID.
  - ☑ Test coverage
- The `partialWithdraw` function calls `claimQueuedWithdrawals` to claim queued withdrawals from EigenPod.
  - ☑ Test coverage
- Distributes the payouts to all the entities.
  - ☑ Test coverage

Negative behavior

- Revert if `numExitRequestsByTnft` for any etherFiNode is nonzero.
  - ☑ Negative test
- Revert if balance of any etherFiNode is less than 16 Ether.
  - ☑ Negative test

Function call analysis

- `this.partialWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).version()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the version of the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the EtherFiNode or an unexpected state.
- `this.partialWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitRequestTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit-request timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.partialWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `this.partialWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`

- **What is controllable?** Not controllable.
- **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase.
- **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- `this.partialWithdraw(_validatorIds[i]) -> this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).migrateVersion(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Migrates the version related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with migrating the version — no reentrancy scenario.
- `this.partialWithdraw(_validatorIds[i]) -> IEtherFiNode(etherfiNode).claimQueuedWithdrawals(this.maxEigenlayerWithdrawals, False)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Claims queued withdrawals in the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with claiming withdrawals — no reentrancy scenario.
- `this.partialWithdraw(_validatorIds[i]) -> this._getTotalRewardsPayoutsFromSafe(_validatorId, True) -> IEtherFiNode(etherfiNode).numExitRequestsByTnft()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the number of exit requests by TNFT in the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.partialWithdraw(_validatorIds[i]) -> this._getTotalRewardsPayoutsFromSafe(_validatorId, True) -> IEtherFiNode(etherfiNode).getRewardsPayouts(this.validatorInfos[_validatorId].exitRequestTimestamp, this.stakingRewardsSplit)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves rewards payouts from the EtherFiNode related to the given validator and returns the split between different entities.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.partialWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> IEtherFiNode(_etherfiNode).withdrawFunds(this.treasuryContract, _toTreasury, this.auctionManager.getBidOwner(_validatorId), _toOperator, this.tnft.ownerOf(_validatorId), _toTnft, this.bnft.ownerOf(_validatorId), _toBnft)`

- **What is controllable?** Not controllable.
- **If the return value is controllable, how is it used and how can it go wrong?** Withdraws funds from the EtherFiNode related to the given validator and distributes them.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with withdrawing funds; the checks-effects-interactions pattern is followed.

- `this.partialWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.auctionManager.getBidOwner(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the bid owner from the auction manager related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.partialWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.tnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the TNFT related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this.partialWithdraw(_validatorIds[i]) -> this._distributePayouts(etherfiNode, _validatorId, toTreasury, toOperator, toTnft, toBnft) -> this.bnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the owner of the BNFT related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `batchQueueRestakedWithdrawal(uint256[] _validatorIds)`

Queue a withdrawal of ETH from an EigenPod.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: No specific constraints.
  - **Impact**: Validator IDs for which restaked withdrawals need to be queued.

Branches and code coverage

Intended branches

- Calls `IEtherFiNode(etherfiNode).queueRestakedWithdrawal()` for each etherFiNode of the validator ID.
  - ☑ Test coverage

Negative behavior

- N/A.

Function call analysis

- `IEtherFiNode(etherfiNode).queueRestakedWithdrawal()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Initiates the queuing of a restaked withdrawal in the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the queuing process — no reentrancy scenario.

Function: `batchRevertExitRequest(uint256[] _validatorIds)`

Revert the exit request for the validators as their TNFT holder.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: `msg.sender` must be the TNFT holder of this validator ID.
  - **Impact**: This is the validator ID for which the exit request must be reverted.

Branches and code coverage

Intended branches

- Updates the number of exit requests.
  - ☑ Test coverage
- Updates the `exitRequestTimestamp` of the validator ID to 0.
  - ☑ Test coverage
- Updates the etherFiNode for the validator ID.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner of the TNFT.
  - ☐ Negative test

Function call analysis

- `this.tnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the owner of the TNFT. The ownership of the TNFT is verified to ensure the caller is the TNFT holder.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the function will revert — no reentrancy scenarios.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).version()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the version of the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the EtherFiNode or an unexpected state.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitRequestTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit-request timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).migrateVersion(_validatorId)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?**

Migrates the version related to the given validator.

- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with migrating the version — no reentrancy scenario.

- `IEtherFiNode(etherfiNode).updateNumExitRequests(0, 1)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the number of exit requests in the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with updating the number of exit requests — no reentrancy scenario.

Function: `batchSendExitRequest(uint256[] _validatorIds)`

Send the request to exit the validators as their TNFT holder. The BNFT holder must serve the request; otherwise, their bond will get penalized gradually.

Inputs

- `_validatorIds`
  - **Control**: Fully controlled.
  - **Constraints**: `msg.sender` must be the TNFT holder of this validator ID.
  - **Impact**: This is the validator ID for which the exit request must be sent.

Branches and code coverage

Intended branches

- Updates the number of exit requests.
  - ☑ Test coverage
- Updates the `exitRequestTimestamp` of the validator ID.
  - ☑ Test coverage
- Updates the etherFiNode for the validator ID.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner of the TNFT.
  - ☑ Negative test
- Revert if the phase is not `LIVE`.
  - ☑ Negative test

Function call analysis

- `this.tnft.ownerOf(_validatorId)`
  - **What is controllable?** `_validatorId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the owner of the TNFT. The ownership of the TNFT is verified to ensure the caller is the TNFT holder.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, the function will revert — no reentrancy scenarios.
- `this.phase(_validatorId) -> IEtherFiNode(etherfiNode).version()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the version of the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with the EtherFiNode or an unexpected state.
- `this.phase(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase of the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitRequestTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit-request timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_exitTimestamp()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated exit timestamp.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).DEPRECATED_phase()`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the deprecated phase.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `this._updateEtherFiNode(_validatorId) -> IEtherFiNode(etherfiNode).migrateVersion(_validatorId)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?**

Migrates the version related to the given validator.

- **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with migrating the version — no reentrancy scenario.

- `IEtherFiNode(etherfiNode).updateNumExitRequests(1, 0)`
  - **What is controllable?** Not controllable.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the number of exit requests in the EtherFiNode related to the given validator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it might indicate an issue with updating the number of exit requests — no reentrancy scenario.

Function: `createUnusedWithdrawalSafe(uint256 _count, bool _enableRestaking)`

This function is responsible for instantiating EtherFiNode and EigenPod proxy instances based on the specified count. It allows the creation of withdrawal safes and an option to enable restaking.

Inputs

- `_count`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: The count affects the number of withdrawal safes created.
- `_enableRestaking`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Determines whether restaking is enabled for the EtherFiNode instances.

Branches and code coverage

Intended branches

- Instantiates new EtherFiNodes and pushes them to the `unusedWithdrawalSafes` array.
  - ☑ Test coverage

Negative behavior

- N/A.

Function call analysis

- `IStakingManager(this.stakingManagerContract).instantiateEtherFiNode(_enableRestaking)`
  - **What is controllable?** The boolean flag `_enableRestaking` controls whether restaking is enabled for the instantiated EtherFiNode.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not controllable by the caller and is used to obtain the address of the newly created EtherFiNode.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it indicates a failure in instantiating the EtherFiNode; no reentrancy scenario is expected.

5.2. Module: EtherFiNode.sol

Function: `claimQueuedWithdrawals(uint256 maxNumWithdrawals, bool _checkIfHasOutstandingEigenLayerWithdrawals)`

Claims queued withdrawals from the EigenPod to this withdrawal safe.

Inputs

- `maxNumWithdrawals`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: Maximum number of queued withdrawals to claim in this TX.
- `_checkIfHasOutstandingEigenLayerWithdrawals`
  - **Control**: Fully controlled.
  - **Constraints**: No constraints.
  - **Impact**: True if user wants to check outstanding EigenLayer withdrawals.

Branches and code coverage

Intended branches

- Returns false if `isRestakingEnabled` is false.
  - ☐ Test coverage
- Calls `hasOutstandingEigenLayerWithdrawals` if `_checkIfHasOutstandingEigenLayerWithdrawals` is true.
  - ☑ Test coverage

Negative behavior

- N/A.

Function call analysis

- `IEtherFiNodesManager(this.etherFiNodesManager).delayedWithdrawalRouter()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, the entire transaction would revert — no reentrancy scenario.
- `delayedWithdrawalRouter.getUserDelayedWithdrawals(address(this))`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the delayed withdrawals of a user.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, the entire transaction would revert — no reentrancy scenario.
- `delayedWithdrawalRouter.claimDelayedWithdrawals(address(this), maxNumWithdrawals)`
  - **What is controllable?** `maxNumWithdrawals`.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, the entire transaction would revert — no reentrancy scenario.
- `this.hasOutstandingEigenLayerWithdrawals()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns true if there are any more outstanding EigenLayer withdrawals.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, the entire transaction would revert — no reentrancy scenario.

5.3. Module: LiquidityPool.sol

Function: `batchDepositAsBnftHolder(uint256[] _candidateBidIds, uint256 _numberOfValidators, uint256 _validatorIdToShareSafeWith)`

This function allows a BNFT player to deposit 2 ETH and pair with 30 ETH from the LP.

Inputs

- `_candidateBidIds`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Determines the validators to be used for the deposit.
- `_numberOfValidators`
  - **Control**: Fully controlled by the caller.

- **Constraints**: No specific constraints mentioned.
- **Impact**: Affects the number of validators to be spun up.

- `_validatorIdToShareSafeWith`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Specifies the validator to share the withdrawal safe with.

Branches and code coverage

Intended branches

- Calls `batchDepositWithBidIds` on the staking manager and returns the funds to the caller for bids that are already taken.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not a registered BNFT holder.
  - ☑ Negative test
- Revert if the balance of the pool is not enough to cover the staking.
  - ☐ Negative test
- Revert if the `msg.value` passed to the function is not equal to `_numberOfValidators * _stakerDepositAmountPerValidator`.
  - ☑ Negative test

Function call analysis

- `this._batchDeposit(_candidateBidIds, _numberOfValidators, 2, _validatorIdToShareSafeWith)` -> `this.stakingManager.batchDepositWithBidIds(_candidateBidIds, _numberOfValidators, msg.sender, SourceOfFunds.EETH, this.restakeBnftDeposits, _validatorIdToShareSafeWith)`
  - **What is controllable?** `_candidateBidIds`, `_numberOfValidators`, and `_validatorIdToShareSafeWith`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents the array of bids that were successfully processed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this call reverts, it indicates a failure in the deposit process; no reentrancy scenario is expected.

Function: `batchDepositWithLiquidityPoolAsBnftHolder(uint256[] _candidateBidIds, uint256 _numberOfValidators, uint256 _validatorIdToShareSafeWith)`

Batch deposit with the liquidity pool as the BNFT holder.

Inputs

- `_candidateBidIds`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Determines the validators to be used for the deposit.
- `_numberOfValidators`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Affects the number of validators to be spun up.
- `_validatorIdToShareSafeWith`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Specifies the validator to share the withdrawal safe with.

Branches and code coverage

Intended branches

- Calls `batchDepositWithBidIds` on the staking manager and returns the funds to the caller for bids that are already taken.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not a registered BNFT holder.
  - ☑ Negative test
- Revert if the balance of the pool is not enough to cover the staking.
  - ☐ Negative test
- Revert if the `msg.value` passed to the function is not equal to `_numberOfValidators * _stakerDepositAmountPerValidator`; `_stakerDepositAmountPerValidator` is 0 for this function, so `msg.value` should be 0, too.
  - ☑ Negative test

Function call analysis

- `this._batchDeposit(_candidateBidIds, _numberOfValidators, 0, _validatorIdToShareSafeWith)` ->

`this.stakingManager.batchDepositWithBidIds(_candidateBidIds, _numberOfValidators, msg.sender, SourceOfFunds.EETH, this.restakeBnftDeposits, _validatorIdToShareSafeWith)`

- **What is controllable?** `_candidateBidIds`, `_numberOfValidators`, and `_validatorIdToShareSafeWith`.
- **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents the array of bids that were successfully processed.
- **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, it indicates a failure in the deposit process; no reentrancy scenario is expected.

5.4. Module: StakingManager.sol

Function: `batchDepositWithBidIds(uint256[] _candidateBidIds, bool _enableRestaking)`

Allows depositing multiple stakes at once. This function processes an array of bid IDs, matches them with stakes, and returns an array of bid IDs that were processed and assigned.

Inputs

- `_candidateBidIds`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Determines the bids to be processed and matched with stakes.
- `_enableRestaking`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No specific constraints mentioned.
  - **Impact**: Determines whether restaking is applied.

Branches and code coverage

Intended branches

- Deposits a multiple of `stakeAmount` for the bid IDs passed in the argument and refunds the unmatched bid amount.
  - ☑ Test coverage

Negative behavior

- Revert if `msg.value` is not a multiple of `stakeAmount` or if `msg.value / stakeAmount` is 0.
  - ☑ Negative test
- Revert if there are not enough bids.

- ☑ Negative test
- Revert if there is an incorrect number of bids.
  - ☑ Negative test
- Revert if the contract is paused.
  - ☑ Negative test

Function call analysis

- `this.auctionManager.numberOfActiveBids()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller. It ensures that there are enough active bids for processing.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this.auctionManager.getBidOwner(bidId)`
  - **What is controllable?** `bidId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents the owner of the bid.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this.auctionManager.isBidActive(bidId)`
  - **What is controllable?** `bidId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents whether the bid is active.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this._verifyNodeOperator(operator, _source)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents the verification result.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this verification call reverts, it indicates a failure in verifying the node operator, impacting the deposit process.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this.auctionManager.updateSelectedBidInformation(bidId)`
  - **What is controllable?** `bidId`.

- **If the return value is controllable, how is it used and how can it go wrong?** No return value.
- **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, it indicates a failure in updating the selected bid information and would revert the entire transaction — no reentrancy scenarios.

- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this._processDeposit(bidId, _staker, _enableRestaking, _source, _validatorIdToShareWithdrawalSafe) -> this.nodesManager.allocateEtherFiNode(_enableRestaking)`
  - **What is controllable?** `_enableRestaking`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and returns the allocated withdrawal safe address.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0) -> this._processDeposit(bidId, _staker, _enableRestaking, _source, _validatorIdToShareWithdrawalSafe) -> this.nodesManager.registerValidator(validatorId, _enableRestaking, etherfiNode)`
  - **What is controllable?** `validatorId` and `_enableRestaking`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, it indicates a failure in registering the validator — no reentrancy scenario.
- `this._processDeposits(_candidateBidIds, numberOfDeposits, msg.sender, ILiquidityPool.SourceOfFunds.DELEGATED_STAKING, _enableRestaking, 0)`
  - **What is controllable?** `_candidateBidIds` and `_enableRestaking`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not directly controllable by the caller and represents the bid IDs that are processed.
  - **What happens if it reverts, reenters or does other unusual control flow?** If this call reverts, it indicates a general failure in the deposit process — no reentracy scenario.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped EtherFi contracts, we discovered 11 findings. Five critical issues were found. One was of high impact, three were of medium impact, one was of low impact, and the remaining finding was informational in nature. EtherFi acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.