Zellic

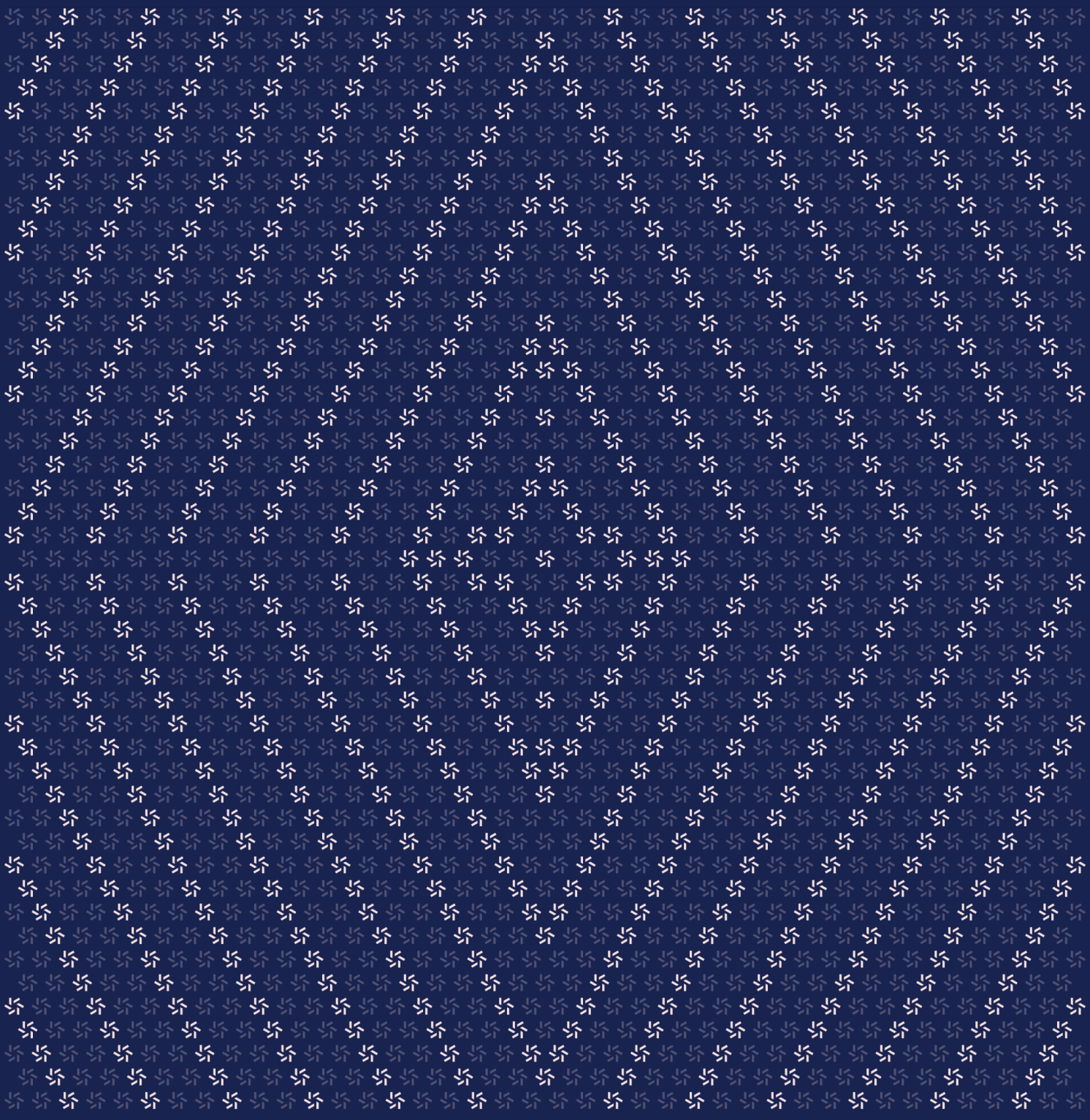
Prepared for
Mike Silagadze
Gadze Finance SEZC

Prepared by
Filippo Cremonese
Kuilin Li
Zellic

January 11, 2024

ether.fi

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 infosec 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. Executive Summary

Zellic conducted a security assessment for Gadze Finance SEZC from January 5th to January 9th, 2024. During this engagement, Zellic reviewed ether.fi's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. 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:

- Is there any way for users to mint eETH without depositing the proper amounts of liquid staking tokens (LSTs)?
- Are there any attacks that can pause the system's operation?
- Are there any attacks that can lead to loss of funds from the users or protocol?

1.2. 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 and centralization risk
- The broader ether.fi protocol and eETH token
- Integration risks with versions of EigenLayer other than M1

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

1.3. Results

During our assessment on the scoped ether.fi contracts, we discovered nine findings. No critical issues were found. Three findings were of high impact, two were of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Gadze Finance SEZC's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	2
Low	2
Informational	2

2. Introduction

2.1. About ether.fi

ether.fi 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:

ether.fi Contracts

Repository	https://github.com/GadzeFinance/dappContracts ↗
Version	dappContracts: `912a4d5a4873a92978ddceec774c3cc21ffd0fa5`
Program	Liquifier
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-days. The assessment was conducted over the course of three calendar days.

Upon completing the assessment, Gadze Finance SEZC requested a focused review of the changes made to LiquidityPool in pull request (PR) #1568 ↗. Refer to section 4.3. ↗, which details the changes introduced in the PR.

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:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 22, 2024	Start of primary review period
January 24, 2024	End of primary review period

3. Detailed Findings

3.1. Incorrect calculation effectively removes fee

Target	Liquifier		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When a queued withdrawal is deposited, the fee per strategy is calculated by `getFeeAmount`:

```
function getFeeAmount() public view returns (uint256) {
    uint256 gasSpendings = EigenLayerWithdrawalClaimGasCost;
    uint256 feeAmount = gasSpendings * block.basefee;
    return gasSpendings;
}
```

The constant `EigenLayerWithdrawalClaimGasCost` is defined to be this:

```
uint32 internal constant EigenLayerWithdrawalClaimGasCost = 150_000;
```

As shown above, `getFeeAmount` returns `gasSpendings` instead of `feeAmount`, which means it will always return 150,000 WEI. This amount of ETH is so small that the fee is effectively not present.

Additionally, when the fee should be withdrawn, the admin calls `withdrawEther`:

```
function withdrawEther() external onlyAdmin {
    // ... snip ...
    if (accumulatedFee > 0.2 ether) {
        uint256 amountToTreasury = accumulatedFee;
        accumulatedFee = 0;
        (sent, ) = payable(treasury).call{value: amountToTreasury, gas:
    5000}("");
```

The quantity 0.2 Ether divided by 150,000 is about 10^12, which means it will take that many deposited strategies for the minimum threshold to be met.

Impact

Due to a coding mistake, the fee is reduced to a negligible quantity.

Recommendations

Although the obvious way to remediate this is to have `getFeeAmount` return `feeAmount`, we recommend against this strategy. This is because `block.basefee` is controllable by the ETH validator mining the transaction that contains the block. Using MEV relayers like MEV-Share, a user can submit a transaction, pay the validator the transaction fee directly, and have the validator set `block.baseFee` to zero. This results in the user not having to pay any fee to the treasury.

Instead, we recommend having this fee be a constant amount set by governance, or a constant amount per strategy depending on the typical gas usage of withdrawing from that particular strategy. This also protects against users paying high fees a second time if they submit a deposit during network congestion that incurs high fees, since the high fee would then be more than the gas cost the admin pays when completing the withdrawal.

Also, we recommend removing the 0.2 ETH minimum from `withdrawEther`, since the admin can already be responsible to call it at a sensible rate, and if it is called too much, the only impact is that the admin pays an unnecessary amount of gas.

Remediation

This issue has been acknowledged by Gadze Finance SEZC, and a fix was implemented in commit `5217c0bc` ↗. This finding was remediated by having `getFeeAmount` return `feeAmount`. The part of this finding where this fee may be dodged using an MEV relayer was acknowledged as unlikely and low-risk.

3.2. Front-runners can cancel any permit deposit

Target	Liquifier		
Category	Protocol Risks	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The `depositWithERC20WithPermit` function allows a user to use an ERC-20 permit, instead of an allowance, to deposit tokens:

```
function depositWithERC20WithPermit(address _token, uint256 _amount,
    address _referral, PermitInput calldata _permit)
    external whenNotPaused returns (uint256) {
    IERC20Permit(_token).permit(msg.sender, address(this), _permit.value,
    _permit.deadline, _permit.v, _permit.r, _permit.s);
    return depositWithERC20(_token, _amount, _referral);
}
```

It processes the permit and then calls the public `depositWithERC20`, which assumes that the sender has given the Liquifier contract an allowance.

However, per the ERC-20 permit specification, given a signed permit, anyone can submit the permit to the ERC-20. So, if a front-runner notices that a legitimate call to `depositWithERC20WithPermit` is in the mempool, they can race to submit that signed permit first.

Impact

If a front-runner successfully submits the signed permit first, the call to `IERC20Permit(_token).permit` will revert due to the reused nonce, cancelling the deposit transaction.

Recommendations

Use a try-catch to allow for the failure of the `permit` call.

If a failure is ignored rather than reverted upon, in the event that someone front-runs it and submits the permit first, the call to `permit` in this function will revert, but the approval will still be there in either case, so the next call to `depositWithERC20` should succeed anyways.

Remediation

This issue has been acknowledged by Gadze Finance SEZC, and a fix was implemented in commit e4119fa0 ↗.

3.3. Completing unqueued withdrawal loses and locks funds

Target	Liquifier		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In the queued withdrawal-deposit process, a user first initiates a queued withdrawal with EigenLayer where the withdrawer is set as the Liquifier contract, and then they call `depositWithQueuedWithdrawal` on Liquifier to get minted a corresponding amount of eETH. This sets an entry of the mapping `isRegisteredQueuedWithdrawals` to true in order to prevent the user from reusing the same withdrawal.

Later, after the withdrawal can be finalized, an admin calls `completeQueuedWithdrawals` to complete the withdrawal and receive the funds. However, this function does not check the `isRegisteredQueuedWithdrawals`, so an admin could maliciously, accidentally, or be tricked into completing a withdrawal that has not yet been used to mint eETH.

Impact

If an admin completes a queued withdrawal that was not deposited, the depositor can still use it to mint eETH. This means that the amount of eETH minted can correspond to the value of the shares at a future time instead of at or before the Liquifier cashes in, which means the Liquifier loses money to the depositor.

Additionally, while the depositor has not deposited the withdrawal, a corresponding amount of funds are locked in the Liquifier. This is because the withdrawal process will decrease `tokenInfos[token].strategyShare` even though no deposit increased it by the amount of shares. So, if the admin tries to complete other legitimate withdrawals, during the last ones, the decreasing of this statistic will underflow and cause the completion to revert.

Recommendations

Check `isRegisteredQueuedWithdrawals` in `completeQueuedWithdrawals` and skip the withdrawal if it has not been deposited.

Alternatively, in order to better help users who mistakenly initiate a withdrawal but then do not call `depositWithQueuedWithdrawal` — if called on a withdrawal that has not been deposited, the completion function could mint them an amount of eETH that they would have gotten if they deposited immediately prior to the completion.

Remediation

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

- 0c998ca4 ↗
- 46dee234 ↗

3.4. More than one strategy per token breaks accounting

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

Description

The `tokenInfos` mapping is defined as a storage variable in Liquifier:

```
mapping(address => TokenInfo) public tokenInfos;
```

This mapping is keyed by ERC-20–token contract addresses and stores information about the particular token; `TokenInfo` is defined in ILiquifier:

```
struct TokenInfo {
    IStrategy strategy;
    uint128 strategyShare;
    uint128 ethAmountPendingForWithdrawals;
    bool isWhitelisted;
}
```

So, this storage mapping stores the strategy contract address corresponding to the token.

However, EigenLayer does not guarantee that one ERC-20 token will only have one whitelisted strategy. In fact, in a pre-M1 version of EigenLayer, the design allowed for the unpermissioned deployment and use of strategies, but a whitelist was added due to issues relating to reentrancy caused by malicious strategies.

Once EigenLayer whitelists more than one strategy for a particular ERC-20 token, this accounting breaks. The `TokenInfo` corresponding to the ERC-20 token cannot mention both strategies, and the accounted `strategyShare` will contain the sum of the shares for the strategies. These shares may be worth different amounts, so when the view function `getTotalPooledEtherSplits` is called and it calculates the restaked ETH value,

```
uint256 restaked = tokenAmountToEthAmount(_token,
    info.strategy.sharesToUnderlyingView(info.strategyShare));
```

this assumes that the strategy in the `TokenInfo` was the strategy that produced all the shares in `strategyShare`, which means it will return an incorrect value.

Impact

Currently, there is no impact. At the time of this audit, EigenLayer has only whitelisted nine strategies, all of which correspond to different ERC-20 tokens.

In the future, if EigenLayer ever whitelists two strategies that use the same underlying ERC-20, the view functions that estimate how much is pending liquification in the Liquifier will be incorrect.

Recommendations

Change the storage layout so that this mapping is keyed by strategy address instead of token address.

When remediating this issue, also consider Finding 3.7. ↗.

Remediation

This issue has been acknowledged by Gadze Finance SEZC, and a fix was implemented in commit dcd42be8 ↗. This finding was remediated by checking the strategy address of the withdrawal request against the one assigned to the token returned by an external call to the strategy. The design choice was made that one whitelisted token will only be covered by one strategy, set at the time the token is whitelisted.

3.5. Admins can steal funds by self-sandwiching swaps

Target	Liquifier		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

Admins are authorized to run the day-to-day operations of the contract, such as initiating Lido withdrawals and completing EigenLayer withdrawals. However, they are not authorized to do things like withdraw funds from the contract to themselves or upgrade the contracts.

One thing admins are allowed to do is use predefined Curve pools to swap cbETH and wbETH into ETH:

```
function swapCbEthToEth(uint256 _amount, uint256 _minOutputAmount)
    external onlyAdmin returns (uint256) {
    cbEth.approve(address(cbEth_Eth_Pool), _amount);
    return cbEth_Eth_Pool.exchange_underlying(1, 0, _amount,
    _minOutputAmount);
}

function swapWbEthToEth(uint256 _amount, uint256 _minOutputAmount)
    external onlyAdmin returns (uint256) {
    wbEth.approve(address(wbEth_Eth_Pool), _amount);
    return wbEth_Eth_Pool.exchange(1, 0, _amount, _minOutputAmount);
}
```

Note how the minimum output amount is controlled entirely by the admin calling these functions.

Impact

An admin wishing to steal funds can conduct a sandwich attack on the protocol's swap easily and safely, and move value from cbETH/wbETH into their own pockets during this swap. If the admin is a smart contract, then it may even be possible to do the sandwiching with a flash loan, which would make this attack require no startup funds / off-chain loans.

Recommendations

Have a lower bound on the minimum output amount such that the protocol does not lose money. Alternatively, make the swap functions owner-only.

Remediation

This issue has been acknowledged by Gadze Finance SEZC. Gadze Finance SEZC will ensure that admins do not conduct sandwich attacks by requiring that they submit swap requests through an oracle node.

3.6. Accumulated fee logic can prevent withdrawals

Target	Liquifier		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `accumulatedFee` storage variable represents the accumulated fee, charged per strategy in deposited queued withdrawals that should be sent to the treasury when the admin calls `withdrawEther`.

However, it is increased when a queued withdrawal is deposited, not when it is completed:

```
function depositWithQueuedWithdrawal(IStrategyManager.QueuedWithdrawal
    calldata _queuedWithdrawal, address _referral)
    external whenNotPaused nonReentrant returns (uint256) {
    // ... snip ...
    /// handle fee
    uint256 feeAmount = _queuedWithdrawal.strategies.length
    * getFeeAmount();
    require(amount >= feeAmount + BALANCE_OFFSET, "less than the fee
    amount");
    amount -= feeAmount;
    accumulatedFee += uint128(feeAmount);
```

This means that, at the specific time that `accumulatedFee` is increased, the amount that it represents has not actually entered the Liquifier's balance yet.

So, when the number of deposited withdrawal strategies spikes, due to market conditions or an attack, when an admin calls `withdrawEther`,

```
function withdrawEther() external onlyAdmin {
    require(address(this).balance >= accumulatedFee, "not enough balance");
```

the "`not enough balance`" check can be made to fail because `accumulatedFee` can be arbitrarily increased by depositors.

Impact

Admins are unable to directly call `withdrawEther` until they complete withdrawals, which may not be possible until time passes.

The impact is limited because, since the contract is upgradable, the funds are not locked. Also, a self-destruct payment (to bypass the receive-function sender whitelist) can be sent to the contract to increase its balance atomically right before `withdrawEther` is called and then the same amount repaid from the liquidity pool.

Recommendations

A simple solution would be to remove this require statement and add logic such that if the balance is less, all of the withdrawn balance goes to the treasury and `accumulatedFee` decreases by the amount sent out.

Alternatively, ensure that the accounting of this fee is always solvent by, instead of making the fee payable in ETH, minting the fee amount of eETH to the treasury, either at deposit time or at `withdrawEther` time. Logically, this is equivalent to the user paying the fee out of the minted eETH they received from their deposit.

Remediation

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

- `d5246d80` ↗
- `ded3c1a7` ↗

3.7. Separate ERC-20 deposit and queued withdrawal whitelists

Target	Liquifier		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `tokenInfos` mapping is defined as a storage variable in Liquifier:

```
mapping(address => TokenInfo) public tokenInfos;
```

This mapping is keyed by ERC-20–token contract addresses and stores information about the particular token; `TokenInfo` is defined in ILiquifier:

```
struct TokenInfo {
    IStrategy strategy;
    uint128 strategyShare;
    uint128 ethAmountPendingForWithdrawals;
    bool isWhitelisted;
}
```

The boolean `isWhitelisted` is used both for the whitelist of tokens that `depositWithERC20WithPermit` can directly accept and for the whitelist of tokens that queued deposits made through `depositWithQueuedWithdrawal` can contain.

Impact

In the future, Gadze Finance SEZC may want to whitelist tokens for `depositWithERC20` that are not whitelisted by EigenLayer yet. Doing this would be awkward, since the strategy field of the `TokenInfo` struct must either be null or not null. If it is not null, then a false strategy must be deployed to satisfy the `underlyingToken()` call matching the token. If it is null (which can be achieved by calling `updateWhitelistedToken` without calling `registerToken`), then `getTotalPooledEther` and `getTotalPooledEtherSplits` for the token will revert because it expects to be able to call into `info.strategy.sharesToUnderlyingView`.

Also, in the future, EigenLayer could whitelist new strategies for existing tokens without considering the impact on Gadze Finance SEZC. This impact could be large if Gadze Finance SEZC whitelisted a token for direct deposits without considering the risk associated with the way EigenLayer's whitelisted strategy slashes withdrawals, and it is not possible to consider this risk

because the former whitelisting could happen before the latter strategy is even created.

Recommendations

With the remediation for Finding 3.4. ↗ in mind, we recommend completely separating the whitelist for queued deposits through EigenLayer (which should be a strategy whitelist, according to that finding, instead of a token whitelist that maps to one strategy per token) and the whitelist for directly depositing ERC-20s, which should still be a token whitelist.

Additionally, consider the amount of logic that needs to be added if another token is whitelisted. The curve pools and the `getTotalPooledEther` function already hardcode Lido, cbETH, and wbETH. If an upgrade to the contract is likely to support more ERC-20s, then a dynamic storage whitelist is unnecessary — the whitelist can just be expressed in the code of the contract, which costs much less gas for the user.

Remediation

This finding was made irrelevant by the design choice to only whitelist one strategy per token in the remediation for Finding 3.4. ↗.

3.8. Deposit is not compatible with noncompliant ERC20s

Target	Liquifier		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When `depositWithERC20` is called to deposit an ERC-20 token, the `transferFrom` is called like this:

```
bool sent = IERC20(_token).transferFrom(msg.sender, address(this), _amount);
require(sent, "erc20 transfer failed");
```

However, some noncompliant ERC-20 tokens, such as USDT, do not return a boolean. So, on success, `sent` will be interpreted as false and the contract will not support it.

Impact

The Liquifier does not support some noncompliant ERC-20 tokens for direct deposit.

Recommendations

We recommend using a wrapper such as SafeERC20 so that noncompliant ERC-20 tokens that do not return a boolean are safely handled.

Remediation

This issue has been acknowledged by Gadze Finance SEZC, and a fix was implemented in commit `21ebc30e` ↗.

3.9. Balance-offset check uses requested amount

Target	Liquifier		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When `depositWithERC20` is called to deposit an ERC-20 token,

```
function depositWithERC20(address _token, uint256 _amount,
    address _referral) public whenNotPaused nonReentrant returns (uint256) {
    require(isTokenWhitelisted(_token), "token is not whitelisted");
    crossover

    uint256 balance = tokenAmountToEthAmount(_token,
    IERC20(_token).balanceOf(address(this)));
    bool sent = IERC20(_token).transferFrom(msg.sender, address(this),
    _amount);
    require(sent, "erc20 transfer failed");
    uint256 newBalance = tokenAmountToEthAmount(_token,
    IERC20(_token).balanceOf(address(this)));
    uint256 dx = newBalance - balance;

    require(!isDepositCapReached(dx), "Deposit cap reached");

    require(_amount > BALANCE_OFFSET, "amount is too small");
    dx -= BALANCE_OFFSET;
```

the variable `dx` properly captures the amount of value actually received by the contract, because the ERC-20 might have a fee on send. However, the check against `BALANCE_OFFSET` is made against `_amount` rather than `dx`. This means that the next subtraction may underflow and revert.

Impact

There is currently no impact because `BALANCE_OFFSET` is zero.

Recommendations

Replace the `_amount` with `dx` or remove `BALANCE_OFFSET` if it is unused.

Additionally, for the transfer, we recommend using a wrapper such as SafeERC20 so that non-compliant ERC-20 tokens that do not return a boolean, such as USDT, can still work.

Remediation

This issue has been acknowledged by Gadze Finance SEZC, and a fix was implemented in commit `614b59d1` ↗.

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. Counterparty and exchange-rate risk

By offering to unconditionally advance to the user eETH in return for whitelisted tokens or whitelisted queued strategies, the Liquifier upgrade takes on a considerable amount of risk by making more value available to attackers who exploit the whitelisted protocols. This is because, for example, if an exploit is discovered in EigenLayer that allowed the queuing of unauthorized withdrawals, in addition to EigenLayer losing funds, the attacker also has the ability to deposit the funds to the Liquifier, minting them eETH.

This risk is limited by the deposit limit, which limits the rate at which the contract can mint eETH. In theory, during one period of the deposit cap, an exploit in the whitelisted contracts can be identified and the contract can be paused for safety pending a fix. The admins in charge of pausing the contract must actively monitor the whitelisted contracts for unexpected upgrades and exploits in order to pause it before the value of eETH drops too much.

Additionally, even disregarding future contract upgrades, the current M1 version of EigenLayer allows queued withdrawals to be slashed. Although EigenLayer does not expect to slash withdrawals, the risk still exists that the contract may do that, and if it does it, then the advanced eETH will similarly not be backed by real value.

4.2. Permit can allow for gasless deposits

The `depositWithERC20WithPermit` function is implemented in order to not require the user to issue a second transaction when directly depositing ERC-20 tokens, because they can instead supply a permit. However, it assumes that the permit is signed by `msg.sender`, which means that a different sender cannot submit the permit.

The other use case for permits is to allow for gasless transactions — a user who does not hold any ETH can still authorize the deposit of an amount of an ERC-20 they hold into the Liquifier. This can be achieved by recovering the owner's address from the permit, or adding it to the `PermitInput` struct and checking it against the signature, and then using that to transfer the ERC-20 and as the target of the eETH mint.

Although not required, adding this feature would allow for gasless deposits without significant changes to the contract.

4.3. Pull request #1568 review

Gadze Finance SEZC requested an isolated review of the changes made to LiquidityPool in pull request (PR) #1568 ↗. The PR is meant to introduce an alternative flow where the LiquidityPool contract acts as the owner of the B-NFT. This section documents our observations regarding that PR. We note that LiquidityPool was not part of the original scope of this audit and that this subengagement was limited to a review of the changes (and possible issues) introduced by the PR in question, in commits 053c9023 ↗, c951b45a ↗, and c40bd506 ↗ (the HEAD of the PR at the time of writing). Note that the PR also contains merge commit 0d649884 ↗, which includes changes not related to the scope of this review.

Introduction of `batchDepositWithLiquidityPoolAsBnftHolder` and `batchRegisterWithLiquidityPoolAsBnftHolder`

Gadze Finance SEZC introduced an additional deposit and registration flow that allows to use the liquidity pool contract as the B-NFT holder. The new flows are intended to be used as an exclusive alternative to the regular flows. The contract has a variable that determines if the liquidity pool should operate as the B-NFT holder (`isLpBnftHolder`). It is initialized as false (allowing to use the regular flow) and can be updated only by the contract administrators.

As also noted in a comment, `updateBnftMode` should not be called while a deposit registration is in progress. Gadze Finance SEZC is aware of this and has expressed the intention to use `updateBfntMode` after pausing the contract and verifying that no deposits are pending. While pausing the contract to verify that no deposits are pending is not ideal, this design does not fundamentally change the trust users have to put in the contract owners, because the contracts are upgradable.

To create the new flow, the previous `batchDepositAsBnftHolder` and `batchRegisterAsBnftHolder` functions were renamed to `_batchDeposit` and `_batchRegister`, respectively, and changed to internal. The logic of the functions was also adjusted to account for the new requirements. The now internal functions are invoked by two public functions acting as entry points for the two flows.

`_batchDeposit`

The logic for batch depositing was changed to now take a parameter that specifies how much ETH must be included with the calling message. With the regular flow, the caller is required to provide 2 ETH per validator. With the new flow, the caller is not required to provide any ETH.

`_batchRegister`

The batch-registration function was updated to take the recipient of the B-NFT as an argument. When the function is invoked using the previous flow, the B-NFT recipient is `msg.sender`. When it is invoked using the new flow where the LP acts as B-NFT holder, the address of the pool is used instead.

Notable unrelated changes

When comparing commit 912a4d5a ↗ with commit c40bd506 ↗ (the HEAD of PR #1568 at the time of writing), we noted the following unrelated changes, which we document for completeness.

Transfer-flow changes. Transfers to the staking manager contract are now performed at the time of the deposit actions instead of when a validator is registered. We were informed that this change is part of a work-in-progress modification to the protocol.

Replacement of `_sendFund`. The `_sendFund` function used to transfer ETH was replaced by a manual low-level call (with a check on the call success). The `_sendFund` function performed an additional check that ensured the balance *after* the call equals the balance *before* the call minus the transferred amount. The removal of this check technically allows the recipient of the transfer to transfer any amount of ETH back into the LiquidityPool contract. We do not believe this creates a security issue as the contract balance can only be increased.

Initialization of `fundStatistics`. The contract initializer now sets `fundStatistics[...].numberOfValidators` to 1 for keys `SourceOfFunds.EETH` and `SourceOfFunds.ETHER_FAN`. We believe this was done to simplify `allocateSourceOfFunds`, which no longer handles the special case of `numberOfValidators == 0`.

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: Liquifier.sol

Function: `depositWithERC20WithPermit(address _token, uint256 _amount, address _referral, PermitInput _permit)`

Mint eETH by transferring stEth, cbEth, or wbEth, to be liquidated by this contract, using a signed permit rather than a previously-submitted token allowance.

Inputs

- `_token`
  - **Control**: Arbitrary.
  - **Constraints**: Must be whitelisted, and also be stEth, cbEth, or wbEth.
  - **Impact**: Token to deposit.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Call to `_token.transferFrom` with amount must succeed.
  - **Impact**: Amount of token to deposit.
- `_referral`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to liquidity pool contract to be emitted in eETH mint event.
- `_permit`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Call to `_token.permit` with permit must succeed.

Branches and code coverage

Intended branches

- Deposit with permit successfully mints eETH and adds to running deposit cap and statistics.
  - ☑ Test coverage

Negative behavior

- Reverts when paused.
  - ☐ Negative test
- Reverts when `depositWithERC20` would revert.
  - ☑ Negative test

Function: `depositWithERC20(address _token, uint256 _amount, address _referral)`

Mint eETH by transferring stEth, cbEth, or wbEth, to be liquidated by this contract.

Inputs

- `_token`
  - **Control**: Arbitrary.
  - **Constraints**: Must be whitelisted, and also be stEth, cbEth, or wbEth.
  - **Impact**: Token to deposit.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: Call to `_token.transferFrom` with amount must succeed.
  - **Impact**: Amount of token to deposit.
- `_referral`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to liquidity pool contract to be emitted in eETH mint event.

Branches and code coverage

Intended branches

- stEth deposit successfully mints eETH and adds to running deposit cap and statistics.
  - ☑ Test coverage
- cbEth deposit successfully mints eETH and adds to running deposit cap and statistics.
  - ☑ Test coverage
- wbEth deposit successfully mints eETH and adds to running deposit cap and statistics.
  - ☑ Test coverage
- ERC20s owned by contract can be liquidated later by admin.
  - ☑ Test coverage

Negative behavior

- Reverts when paused.

- ☐ Negative test
- Reverts when reentrancy is attempted.
  - ☐ Negative test
- Reverts on unwhitelisted token.
  - ☐ Negative test
- Reverts if `transferFrom` fails.
  - ☑ Negative test
- Reverts if deposit cap is reached.
  - ☑ Negative test
- Reverts if amount of eETH to be minted is insufficient to pay fees.
  - ☐ Negative test

Function: `depositWithQueuedWithdrawal(IStrategyManager.QueuedWithdrawal _queuedWithdrawal, address _referral)`

Mint eETH by supplying proof of a queued EigenLayer withdrawal whose `withdrawer` is the Liquifier.

Inputs

- `_queuedWithdrawal`
  - **Control**: Arbitrary.
  - **Constraints**: Hash is checked. It must not have been deposited before, and EigenLayer must confirm it is a queued withdrawal. Also, tokens are checked against a token/strategy whitelist.
  - **Impact**: Value determines value of eETH to mint.
- `_referral`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Passed to liquidity pool contract to be emitted in eETH mint event.

Branches and code coverage

Intended branches

- Queued withdrawal successfully mints eETH and adds to running deposit cap and statistics.
  - ☑ Test coverage
- Queued withdrawal can be completed by admin.
  - ☑ Test coverage

Negative behavior

- Reverts when paused.
  - ☐ Negative test
- Reverts when reentrancy is attempted.
  - ☐ Negative test
- Reverts if sender is not withdrawal depositor.
  - ☐ Negative test
- Reverts if this contract is not withdrawal withdrawer.
  - ☑ Negative test
- Reverts on unwhitelisted token or strategy.
  - ☐ Negative test
- Reverts if withdrawal is not queued in EigenLayer.
  - ☐ Negative test
- Reverts if withdrawal has already been deposited.
  - ☑ Negative test
- Reverts if deposit cap is reached.
  - ☐ Negative test
- Reverts if amount of eETH to be minted is insufficient to pay fees.
  - ☐ Negative test

Function: `receive()`

Receive ether without calldata. Whitelisted to Lido withdrawal queue, cbEth pool, and wbEth pool.

Branches and code coverage

Intended branches

- Ether transfer from allowed source succeeds.
  - ☑ Test coverage

Negative behavior

- Ether transfer from other sources fails.
  - ☑ Negative test

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 ether.fi contracts, we discovered nine findings. No critical issues were found. Three findings were of high impact, two were of medium impact, two were of low impact, and the remaining findings were informational in nature. Gadze Finance SEZC 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.