Zellic

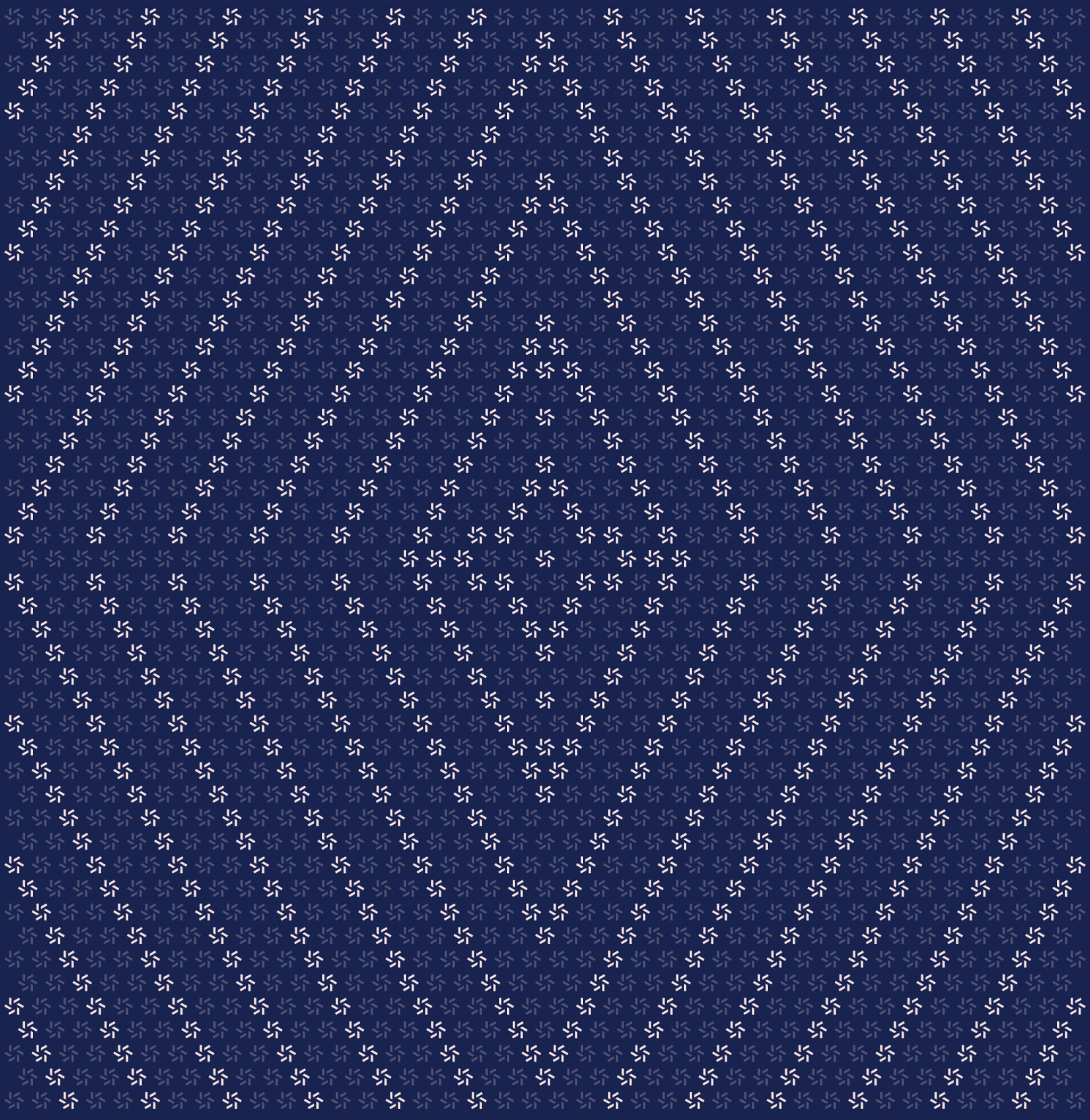
Prepared for
Joey
Saffron

Prepared by
Seungjun Kim
Kuilin Li
Zellic

September 4, 2024

Lido Fixed Income

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 Saffron from April 1st to April 8th, 2024. During this engagement, Zellic reviewed Lido Fixed Income'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:

- Could an on-chain attacker drain from the vault?
- Could an on-chain attacker cause a DOS on the contract?
- Is there a risk of withdrawing funds belonging to another user?

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 Lido Fixed Income contracts, we discovered 14 findings. Two critical issues were found. Three were of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	3
Medium	2
Low	5
Informational	2

2. Introduction

2.1. About Lido Fixed Income

Saffron contributed the following description of Lido Fixed Income:

> Saffron LIDO Fixed Income Vaults provides fixed income from ETH staking.

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:

Lido Fixed Income Contracts

Repository	https://github.com/saffron-finance/lido-fiv ↗
Version	lido-fiv: `598dce27175b0d26e3e3d78632732a4e45afef57`
Programs	• VaultFactory • LidoVault
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 one calendar week.

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:

Seungjun Kim
Engineer
seungjun@zellic.io

Kuilin Li
Engineer
kuilin@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

April 1, 2024	Start of primary review period
April 8, 2024	End of primary review period

3. Detailed Findings

3.1. Fixed depositor reentrancy can take all the ETH

Target	LidoVault		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

After a vault begins, fixed depositors can call the `claimFixedPremium()` function to claim their ETH fixed premium:

```
function claimFixedPremium() external {
  require(isStarted(), "CBS");

  // Check and cache balance for gas savings
  uint256 claimBal = fixedClaimToken[msg.sender];
  require(claimBal > 0, "NCT");

  // Send a proportional share of the total variable side deposits (premium) to
    the fixed side depositor
  uint256 sendAmount
    = fixedETHDepositToken[msg.sender].mulDiv(variableSideCapacity,
    fixedSideCapacity);

  // Track premiums
  userToFixedUpfrontPremium[msg.sender] = sendAmount;

  (bool sent, ) = msg.sender.call{value: sendAmount}("");
  require(sent, "ETF");

  // Mint bearer token
  fixedBearerToken[msg.sender] += claimBal;
  fixedBearerTokenTotalSupply += claimBal;

  // Burn claim tokens
  fixedClaimToken[msg.sender] = 0;
  fixedClaimTokenTotalSupply -= claimBal;

  emit FixedPremiumClaimed(sendAmount, claimBal, msg.sender);
}
```

However, since the ETH is sent before the burning of the claim tokens, if the call to `msg.sender` reenters the same function, the user can claim their fixed premium again and again.

Impact

Any fixed depositor can reenter `claimFixedPremium()` as many times as the gas and stack-size limit permit. By ensuring that their initial deposit is an even multiple of their claim, they can drain the entire balance using this reentrancy attack.

Additionally, this inflates the user's `fixedBearerToken`, which causes them to be able to claim more of the returned fixed principal than they deserve after the vault ends.

Recommendations

Send the `sendAmount` to the caller after the claim tokens are burned.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit `0b1fbc2d` ↗.

3.2. Unsafe handling of over-100% early exit fees

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

Description

During a vault's earning period, a fixed depositor can call `withdraw()` to withdraw their stake and back out of the vault. If they do so, after the withdraw is finalized and they call `finalizeVaultOngoingFixedWithdrawals()` to claim their ETH, they are charged an early exit fee that depends on the timestamp they exited.

This early exit fee is calculated in the function `claimFixedVaultOngoingWithdrawal`, which returns the ETH amount ultimately transferred to the `user`:

```
function claimFixedVaultOngoingWithdrawal(address user)
    internal returns (uint256) {
  if (user == address(0)) return 0;

  WithdrawalRequest memory request
    = fixedToVaultOngoingWithdrawalRequestIds[user];
  uint256[] memory requestIds = request.requestIds;
  require(requestIds.length != 0, "WNR");

  uint256 upfrontPremium = userToFixedUpfrontPremium[user];

  delete userToFixedUpfrontPremium[user];
  delete fixedToVaultOngoingWithdrawalRequestIds[user];

  uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
  for (uint i = 0; i < arrayLength; i++) {
    if (fixedOngoingWithdrawalUsers[i] == user) {
      delete fixedOngoingWithdrawalUsers[i];
    }
  }

  uint256 amountWithdrawn = claimWithdrawals(msg.sender, requestIds);

  uint256 earlyExitFees = calculateFixedEarlyExitFees(upfrontPremium,
    request.timestamp);
```

```
    // add earlyExitFees to earnings for variable side
    feeEarnings += earlyExitFees

    emit LidoWithdrawalFinalized(user, requestIds, FIXED, isStarted(),
      isEnded());

    return amountWithdrawn - Math.min(earlyExitFees, amountWithdrawn);
  }
```

Note that according to the documentation, the early exit fee starts at 110% of their deposited stake.

The `feeEarnings` variable, when increased, directly represents fees able to be withdrawn by the variable-side depositors. Because of the `Math.min(earlyExitFees, amountWithdrawn)`, this fee is given to the variable-side depositors even when the fee cannot be removed, due to lack of funds, from the amount withdrawn by the fixed-side caller.

Impact

If the `Math.min` call above happens while `earlyExitFees > amountWithdrawn`, the difference can be claimed by the variable side even though no portion of the contract's actual balance includes it. In normal operation, this means that the last few variable-side users to claim the fee will not be able to claim it until more ETH enters the contract.

This issue creates a significant perverse economic exploit. If the first user to deposit into the vault is a fixed depositor who does not fill up most of the fixed space, then an attacker can fill up the entire remaining fixed and variable space and then immediately withdraw all of their fixed deposits. Since the attacker owns the entire variable space, they are entitled to all the fees that their fixed holdings were charged, and since zero time has passed, there will be no protocol fees for their fixed withdrawal. So, if the attacker claims their fixed ongoing withdrawal immediately after the vault ends, the extra 10% of the attacker's large fixed position is paid for by the victim's smaller fixed position, and the victim cannot withdraw due to the contract running out of native ETH.

Recommendations

If the early exit fees exceed the amount withdrawn, then the entire amount withdrawn should be added to `feeEarnings` instead of an amount that exceeds it.

Alternatively, if the suggestion in Discussion 4. ↗ is implemented, then this would not be an issue since the early exit fees would always be less than the amount withdrawn, both denominated in stETH.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit `a9cc64b4` ↗.

3.3. Clone construction leaves constants uninitialized

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

Description

The LidoVault contract contains state variables that are implicitly initialized to constant values:

```
contract LidoVault is ILidoVaultInitializer, ILidoVault {
  // [...]

  /// @notice Minimum amount of ETH that can be deposited for variable or fixed
    side users
  uint256 public minimumDepositAmount = 0.01 ether;

  /// @notice Minimum amount of the fixed capacity that can be deposited for
    fixed side users on a single deposit in basis points
  uint256 public minimumFixedDepositBps = 500; // default 5%

  // [...]
}
```

However, in VaultFactory, a new LidoVault is deployed using `Clones.clone`:

```
function createVault(
  uint256 _fixedSideCapacity,
  uint256 _duration,
  uint256 _variableSideCapacity
) public virtual {
  // Deploy vault (Note: this does not run constructor)
  address vaultAddress = Clones.clone(vaultContract);

  // [...]
}
```

This means that the state variables mentioned above are uninitialized in the clone.

Impact

For all vaults deployed by this factory, the minimum deposit amount and minimum fixed basis points are uninitialized and zero, effectively turning off those two limits. This can lead to vaults being created with incorrect incentives, dusting attacks on vaults, and everything else those limits were intended to guard against.

Recommendations

Make these constants `constant` or `immutable` so that they are not assigned a storage slot, so that they cannot be erroneously uninitialized in clones.

Additionally, we recommend having the test fixtures deploy the LidoVault that is tested through the LidoVaultFactory, rather than directly. That is why this issue was missed in the tests - in `deployVault` in the file `1.LidoVault.test.ts`, even though it calls the variable `LidoVaultFactory`, it is actually a `LidoVault`:

```
let LidoVaultFactory = await ethers.getContractFactory('LidoVault')
```

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit `eb7fbd37` ↗.

3.4. Quadratic-complexity logic risks gas-limit attacks

Target	LidoVault		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

In the LidoVault, the variable-length array `fixedOngoingWithdrawalUsers` is used to store a list of fixed-side users that requested a withdrawal while the vault has not ended.

During the vault-finalization process, the internal function `claimOngoingFixedWithdrawals()` is called to finish all of the ongoing fixed-side deposits so that fees can be properly accounted for even if the fixed-side user never completes their withdrawal themselves.

This function loops through the `fixedOngoingWithdrawalUsers` array:

```
function claimOngoingFixedWithdrawals() internal {
  uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
  for (uint i = 0; i < arrayLength; i++) {
    address fixedUser = fixedOngoingWithdrawalUsers[i];
    fixedToPendingWithdrawalAmount[fixedUser]
    = claimFixedVaultOngoingWithdrawal(fixedUser);
  }
}
```

The function that it calls inside the loop, `claimFixedVaultOngoingWithdrawal`, however, also loops through the array to set the user in question to zero:

```
function claimFixedVaultOngoingWithdrawal(address user)
    internal returns (uint256) {
  // [...]

  uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
  for (uint i = 0; i < arrayLength; i++) {
    if (fixedOngoingWithdrawalUsers[i] == user) {
      delete fixedOngoingWithdrawalUsers[i];
    }
  }

  // [...]
```

```
}
```

This means the overall vault finalization process runs in $O(n^2)$ time, if n is the number of users. Note that the `delete` statement does not reduce the size of the array, it sets the array entry at index `i` to zero.

Impact

The only limit on the number of fixed-side depositors is the fixed-side capacity divided by the minimum deposit. If this implicit limit is too large, then it is possible that a vault with many fixed-side depositors cannot be finalized because it would take too much gas.

Recommendations

We recommend reworking the withdrawal logic to not require traversing a variable-length array.

Alternatively, adding an optional parameter to `claimFixedVaultOngoingWithdrawal` that provides a hint for the index of the user to be removed will make the overall logic linear-time in all cases — either the caller or the callee takes $O(n)$ to find the index.

Alternatively, if the suggestion in Discussion 4. ↗ is implemented, then an ongoing fixed withdrawal would be settled immediately in stETH (with optional use of the withdrawal queue happening in a different contract), and so this would not be an issue.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit `a75cf521` ↗.

3.5. Inconsistent division of fixed-side withdrawals is unfair

Target	LidoVault		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

When fixed-side depositors deposit into the LidoVault, both the amount of stETH shares minted due to their native ETH deposit and the quantity of native ETH deposited are recorded:

```
function deposit(uint256 side) external payable {
  // [...]

  if (side == FIXED) {
    // [...]

    // Stake on Lido
    /// returns stETH, and returns amount of Lido shares issued for the staked
    ETH
    uint256 stETHBalanceBefore = stakingBalance();
    uint256 shares = lido.submit{value: amount}(address(0)); // _referral
    address argument is optional use zero address
    require(shares > 0, "ISS");
    // stETH transfered from Lido != ETH deposited to Lido - some rounding error
    uint256 stETHReceived = (stakingBalance() - stETHBalanceBefore);
    require((stETHReceived >= amount) || (amount - stETHReceived
    <= LIDO_ERROR_TOLERANCE_ETH), "ULD");
    emit FundsStaked(amount, shares, msg.sender);

    // Mint claim tokens
    fixedClaimToken[msg.sender] += shares;
    fixedClaimTokenTotalSupply += shares;
    fixedETHDepositToken[msg.sender] += amount;
```

Next, when they claim their fixed premium, the amount of native ETH claimed from the variable side's deposit is dependent on the amount of ETH they deposited, which is `fixedETHDepositToken`:

```
function claimFixedPremium() external {
  // [...]
```

```
  uint256 claimBal = fixedClaimToken[msg.sender];
  require(claimBal > 0, "NCT");

  // Send a proportional share of the total variable side deposits (premium) to
    the fixed side depositor
  uint256 sendAmount
    = fixedETHDepositToken[msg.sender].mulDiv(variableSideCapacity,
    fixedSideCapacity);

  // Track premiums
  userToFixedUpfrontPremium[msg.sender] = sendAmount;

  (bool sent, ) = msg.sender.call{value: sendAmount}("");
```

Then, before the vault ends, if they withdraw, the amount of fixed deposit returned to them is also proportional to their `fixedETHDepositToken`:

```
function withdraw(uint256 side) external {
  require(side == FIXED || side == VARIABLE, "IS");

  // Vault has not started
  if (!isStarted()) {

    // [...]

    // Vault started and in progress
  } else if (!isEnded()) {
    if (side == FIXED) {
      // [...]

      uint256 initialDepositAmount = fixedETHDepositToken[msg.sender];
      require(initialDepositAmount > 0, "NET");

      // since the vault has started only withdraw their initial fixed deposit
    - unless we are in a loss
      uint256 withdrawAmount = initialDepositAmount;
      uint256 lidoStETHBalance = stakingBalance();
      uint256 fixedETHDeposits = fixedETHDepositTokenTotalSupply;

      // [...] [assume stETH did not default]

      fixedToVaultOngoingWithdrawalRequestIds[msg.sender]
    = WithdrawalRequest({
        requestIds: requestWithdrawViaETH(msg.sender, withdrawAmount),
        timestamp: block.timestamp
      });
```

But, if they instead withdraw after the vault ends, the amount of fixed deposit returned to them is proportional to the shares they originally deposited, a quantity now tracked in `fixedBearerToken`:

```
function vaultEndedWithdraw(uint256 side) internal {
  // [...]

  uint256 bearerBalance = fixedBearerToken[msg.sender];

  require(bearerBalance > 0, "NBT");
  sendAmount = fixedBearerToken[msg.sender].mulDiv(
    vaultEndedFixedDepositsFunds,
    fixedLidoSharesTotalSupply()
  );

  // [...]

  transferWithdrawnFunds(msg.sender, sendAmount);
```

Impact

This inconsistent division of assets causes discontinuities across the time periods of the vault. For example, consider the case where Alice and Bob each deposit 1 ETH when the share price was one share = 1 ETH, then time passes and yields are earned, and then Charlie deposits 2 ETH when the share price is at one share = 2 ETH. Based on the shares deposited, Charlie is entitled to a third of the value of the total principal, but based on ETH value deposited, Charlie is entitled to half.

So, Charlie will want to do an ongoing withdraw close to the end of the vault, instead of letting his share be finalized, because then he gets more of the fixed premium returned to him.

This effect is even worse if Lido defaults after some fixed depositors deposit but before the vault starts. If this happens, the vault essentially becomes a trap, because future fixed depositors will deposit many more shares due to the lower ETH price of a share, and therefore pay for the value the pre-default fixed depositors already lost.

Recommendations

We recommend always dividing the fixed principal by shares.

Additionally, consider allowing fixed depositors to hold their fixed principal in ETH rather than in stETH before the vault begins and having the option for those depositors to withdraw ETH immediately if they want to back out before the vault starts.

This is reasonable because, before a vault begins, if the partial-fixed premium gains yields or loses value, it is not clear who those yields belong to and who should pay for any losses, since anyone can permissionlessly start the vault.

For instance, if the vault fixed-side capacity is 5 ETH, and 4 ETH has been deposited but that has grown to 6 ETH, and there are not yet any depositors on the variable side, then anyone can come along and deposit 1 ETH to the fixed side and fill up the variable side and instantly claim the entire 2 ETH yield that has already been gained from the variable side. Depending on the size of the variable side, this may be an instantaneous profit even if Lido defaults the very next block.

On the other hand, if the fixed side sends in withdrawals first, before the vault has a chance to start, then the fixed side gets all of the yield returned to them. In essence, yield is created by subjecting that native ETH to the counterparty risk, but whether or not the vault starts yet is unknown. Instead of making the claiming of that excess yield a gas auction, it makes more sense to not subject the native ETH to that risk yet, before the vault starts, and consequentially also allow fixed-side depositors to immediately withdraw their ETH if they wish.

Remediation

This issue has been acknowledged by Saffron.

3.6. Variable-side yield on yield is unfairly split

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

Description

During a vault's lifetime, the staked stETH grows due to Lido yield, and that stETH belongs to the variable-side depositors. At any time, a variable-side depositor can request the disbursement of their share of the gains from that yield:

```
function withdraw(uint256 side) external {
  // [...] [ case !isEnded() && side == VARIABLE ]

  uint256 lidoStETHBalance = stakingBalance();
  uint256 fixedETHDeposits = fixedETHDepositTokenTotalSupply;

  // staking earnings have accumulated on Lido
  if (lidoStETHBalance > fixedETHDeposits) {
    (uint256 currentState, uint256 ethAmountOwed)
    = calculateVariableWithdrawState(
      (lidoStETHBalance - fixedETHDeposits) + withdrawnStakingEarnings
    + totalProtocolFee,
      variableToWithdrawnStakingEarnings[msg.sender].mulDiv(10000,
    10000 - protocolFeeBps)
    );

    if (ethAmountOwed >= minStETHWithdrawalAmount()) {
      // estimate protocol fee and update total - will actually be applied on
    withdraw finalization
      uint256 protocolFee = ethAmountOwed.mulDiv(protocolFeeBps, 10000);
      totalProtocolFee += protocolFee;

      withdrawnStakingEarnings += ethAmountOwed - protocolFee;
      variableToWithdrawnStakingEarnings[msg.sender] = currentState
    - currentState.mulDiv(protocolFeeBps, 10000);

      variableToVaultOngoingWithdrawalRequestIds[msg.sender]
    = requestWithdrawViaETH(
        msg.sender,
        ethAmountOwed
```

```
        );
```

Consider this early partial-withdrawal decision from a variable-side depositor's perspective. If Alice, a variable-side depositor, is allowed to withdraw 1 ETH of value currently held in stETH but chooses not to, then that 1 stETH continues to earn yield from Lido. The longer the duration of the vault, the more this compounding effect increases total yield percentage.

However, if Alice does not withdraw from the vault, then the compounded value is not actually given to her; it is currently not distinguished from yields earned on the original principal, which means it is evenly (based on deposit) distributed across all variable depositors. So, Alice will favor withdrawing the partial value immediately whenever she can (notwithstanding the Lido minimum withdrawal and the price of gas), in order to restake that in Lido herself and get the full amount of yield on that yield.

Impact

This means that all profit-maximizing active variable-side investors should withdraw as often as possible, which ultimately decreases net yield due to stETH not earning interest while in the withdrawal queue. Additionally, this effect is unfriendly to passive investors and causes users to have to spend more gas.

Recommendations

A more succinct description of the underlying mathematical miscalculation is that `withdrawnStakingEarnings` is a fixed quantity denominated in past stETH, a quantity that does not change as the Lido share price changes. If Lido earns yield and does not default, then the share price should gradually be increasing, so past stETH should be worth more than present stETH, so keeping `withdrawnStakingEarnings` constant results in more value being allocated to any early withdrawer.

Instead of denominating `withdrawnStakingEarnings` in ETH, it should be denominated in Lido shares to better represent the permanent withdrawal of future earnings that a present variable-side withdrawal causes. Whenever the withdraw function is called, it should calculate the total amount of Lido shares, originally entirely owned by the fixed side, that the entire variable side is entitled to using the current share price, and allow a withdrawal of the rest of the Lido shares that a particular variable depositor is entitled to (subtracting out the Lido shares they have already withdrawn).

Remediation

This issue has been acknowledged by Saffron.

3.7. Variable side cannot always withdraw fee share

Target	LidoVault		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

During a vault's lifetime, a variable-side depositor can withdraw both their share of the yields and their share of the early exit fees charged by withdrawing fixed-side depositors. This is implemented in the `withdraw` function:

```
function withdraw(uint256 side) external {
  // [...] [ case !isEnded() && side == VARIABLE ]

  require(variableToVaultOngoingWithdrawalRequestIds[msg.sender].length == 0,
    "WAR");

  // [...]

  if (lidoStETHBalance > fixedETHDeposits) {

    // [...]

    if (ethAmountOwed >= minStETHWithdrawalAmount()) {
      // [...]

      variableToVaultOngoingWithdrawalRequestIds[msg.sender]
    = requestWithdrawViaETH(
        msg.sender,
        ethAmountOwed
      );

      // [...]

      return;
    }
  }

  // there are no staking earnings that can be withdrawn but there are fixed
    side early withdrawal fees
  if (feeEarnings > 0) {
```

```
    uint256 feeEarningsShare = calculateVariableFeeEarningsShare();
    require(feeEarningsShare > 0, "NZW");

    transferWithdrawnFunds(msg.sender, feeEarningsShare);

    emit VariableFundsWithdrawn(feeEarningsShare, msg.sender, isStarted(),
    isEnded());
    return;
  }
```

However, note that the first type of withdrawal is tried first and, if it succeeds in queueing a withdrawal, the fee earnings are not withdrawn. Also, if the user has an ongoing withdrawal in the queue, they also cannot withdraw due to the require.

Impact

Even though the `feeEarnings` consists of ETH held in the contract, variable-side depositors inconveniently cannot withdraw it unless they also withdraw their yield earnings and wait for that withdrawal to complete.

Recommendations

We recommend processing the fee earnings withdrawal even if there is a pending yield withdrawal and also even if the withdrawal call successfully caused the queueing of a new yield withdrawal.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit ee03623d ↗.

3.8. Open receive function can lock native ETH

Target	LidoVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The LidoVault has a public payable receive function:

```
/// @notice Function to receive Ether. msg.data must be empty
receive() external payable {}
```

This is so that it can receive native ETH sent by the Lido withdrawal queue.

Impact

However, users may accidentally send ETH to the contract, which results in the ETH being stuck and effectively burned.

Recommendations

We recommend at least whitelisting the payable receive function to the Lido withdrawal value sender so that users cannot accidentally transfer value to the contract.

Another solution is to have a state variable that is temporarily set before any call to Lido's `claimWithdrawal` and unset afterwards, and only allow the receive hook to not revert if that variable is set. This would prevent ETH from being locked by users calling `claimWithdrawalsTo` with the recipient being the LidoVault. However, that situation is not likely something that a user accidentally does.

Note that it is not possible to completely prevent value from being stuck in the LidoVault without a permissioned emergency withdrawal function that sends it back out, because of self destructs, coinbase transfers, and transfers of value before deployment.

Remediation

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

- 59bd5003 ↗
- 0d98df9a ↗

- d41395af ↗

3.9. Implementation contract can be used

Target	VaultFactory		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

During the construction of the VaultFactory, the implementation LidoVault is constructed:

```
constructor(uint256 _protocolFeeBps, uint256 _earlyExitFeeBps) {
  // [...]

  vaultContract = address(new LidoVault());
}
```

However, this implementation contract is fully usable by the first to call `initialize` on it, due to a lack of a constructor that disables the initializer.

Impact

The only impact is that if that contract is initialized maliciously, it can be used once to charge higher-than-expected fees and send the fees to an incorrect protocol-fee receiver. Although the view function `wasDeployedByFactory` will return false for this vault, on Etherscan and similar chain explorers, the contract will still say it was deployed by the vault, which can mislead users.

Recommendations

Even though the impact is small, we recommend adding a constructor to the LidoVault that explicitly disables the implementation contract from being used.

Remediation

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

- 6c498b48 ↗
- 4877d7fa ↗
- 80930eeb ↗

3.10. Incorrect `withdrawnFeeEarnings` after finalization

Target	LidoVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

While the vault is in progress, the helper function `calculateVariableFeeEarningsShare` is used to calculate the current amount owed to the sender as a variable-side depositor:

```
function calculateVariableFeeEarningsShare() internal returns (uint256) {
  (uint256 currentState, uint256 feeEarningsShare)
    = calculateVariableWithdrawState(
    feeEarnings + withdrawnFeeEarnings,
    variableToWithdrawnFees[msg.sender]
  );

  variableToWithdrawnFees[msg.sender] = currentState;
  withdrawnFeeEarnings += feeEarningsShare;
  feeEarnings -= feeEarningsShare;

  return feeEarningsShare;
}
```

However, after the vault is finalized, the calculation is done without this helper function but with similar logic:

```
function vaultEndedWithdraw(uint256 side) internal {
  // [...]

  uint256 feeShareAmount = 0;
  uint256 totalFees = withdrawnFeeEarnings + feeEarnings;

  if (totalFees > 0) {
    (uint256 currentState, uint256 feesShare)
    = calculateVariableWithdrawState(
      totalFees,
      variableToWithdrawnFees[msg.sender]
    );
    feeShareAmount = feesShare;
```

```
    variableToWithdrawnFees[msg.sender] = currentState;
  }
```

Note that this logic is identical to the logic in the helper function, except that `withdrawnFeeEarnings` is no longer updated.

Impact

The public view function for `withdrawnFeeEarnings` will stop tracking the amount of withdrawn fee earnings after the vault is finalized.

There is no other impact because everywhere it is internally read, `withdrawnFeeEarnings` is added to `feeEarnings`, and this oversight does not affect the correctness of the sum of the two.

Recommendations

We recommend using the helper function `calculateVariableFeeEarningsShare` in `vaultEndedWithdraw`, instead of repeating the logic, to increase code maintainability and ensure that `withdrawnFeeEarnings` is always correct when read by users and third-party contracts.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit `fddfa3b1` ↗.

3.11. Reentrancy can falsify `isStarted` in emitted event

Target	LidoVault		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In multiple places, `isStarted()` and `isEnded()` are called in the same line of code that an event is emitted. For example, when a variable-side user withdraws before the vault starts:

```
function withdraw(uint256 side) external {
  // [...] [ case !isStarted() && side == VARIABLE ]

  uint256 sendAmount = variableBearerToken[msg.sender];
  require(sendAmount > 0, "NBT");

  variableBearerToken[msg.sender] -= sendAmount;
  variableBearerTokenTotalSupply -= sendAmount;

  (bool sent, ) = msg.sender.call{value: sendAmount}("");
  require(sent, "ETF");

  emit VariableFundsWithdrawn(sendAmount, msg.sender, isStarted(), isEnded());
  return;
```

However, the call to the sender to send `sendAmount` of native ETH can reenter the LidoVault. If during this reentrancy, a deposit is issued that completes the vault despite the withdrawal, then `isStarted()` called after the reentrancy returns will incorrectly be true.

Impact

Reentrancy attacks can cause the `isStarted` field of emitted events to be falsified, confusing external subscribers to these events.

Recommendations

For `emit` lines where an event is emitted and where `isStarted()` and `isEnded()` at the beginning of the call are known because of conditionals around that line of code, such as the above example where clearly the vault has not started yet, replacing them with constants would save gas and fix this

issue.

For cases like `finalizeVaultNotStartedFixedWithdrawals`, where reentrancy is possible through `transferWithdrawnFunds` but it is unknown based on context whether the vault has started or not, consider emitting the event first or saving the values of `isStarted()` and `isEnded()` prior to the call to sender.

Remediation

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

- `4c971a17` ↗
- `44a82869` ↗
- `c77079b3` ↗

3.12. Inactive variable depositor locks protocol fees

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

Description

The variable `appliedProtocolFee` is increased whenever protocol fees are charged, and the `protocolFeeReceiver` can withdraw them using various withdraw procedures throughout the vault life cycle.

However, if a variable-side depositor requests a withdraw during the vault's lifetime, the protocol fee is only applied when the request is finalized and they call `finalizeVaultOngoingVariableWithdrawals`:

```
function finalizeVaultOngoingVariableWithdrawals() external {
  // [...]

  uint256 protocolFee = applyProtocolFee(amountWithdrawn);

  // [...]
}

function applyProtocolFee(uint256 stakingEarnings) internal returns (uint256)
    {
  uint256 protocolFee = stakingEarnings.mulDiv(protocolFeeBps, 10000);
  appliedProtocolFee += protocolFee;
  return protocolFee;
}
```

This means that if a variable-side depositor disappears and never finalizes their withdrawal, the protocol fee is never claimable by the protocol-fee receiver even if the vault completely ends.

Impact

Protocol fees that should have been claimable can become lost if a variable-side depositor goes missing or loses their private key. For vaults with many depositors, it is likely that the vault must be monitored by the protocol-fee receiver for a long time to withdraw the protocol fee if a depositor ever comes back and claims their funds. Over time, this unboundedly builds up the number of contracts the protocol-fee receiver must watch for events.

Recommendations

Allow the protocol-fee receiver to finalize ongoing variable withdrawals on behalf of the withdrawer. Similar to `fixedToPendingWithdrawalAmount`, the value should be kept in ETH and the withdrawer should be able to withdraw their share later.

Remediation

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

- a9c6506e ↗
- ccd1c0e3 ↗
- ebcb06a4 ↗

3.13. Coordination hazard causes reverts on finalization

Target	LidoVault		
Category	Protocol Risks	**Severity**	Informational
Likelihood	Medium	**Impact**	Informational

Description

If a vault has many depositors, when a vault's time period has ended and then its withdrawal is finalized, it is likely that multiple depositors want to withdraw. To do this, the first such withdrawer must call `finalizeVaultEndedWithdrawals`:

```
function finalizeVaultEndedWithdrawals(uint256 side) external {
  require(side == FIXED || side == VARIABLE, "IS");
  require(vaultEndedWithdrawalRequestIds.length != 0 &&
    !vaultEndedWithdrawalsFinalized, "WNR");

  vaultEndedWithdrawalsFinalized = true;

  // [...] [actually finalize the withdrawal]

  return vaultEndedWithdraw(side);
}
```

And then after the first withdrawer, the rest of them must call `withdraw`, which directly calls `vaultEndedWithdraw`.

However, if the withdrawers do not coordinate off chain, then multiple withdrawers could issue a transaction for `finalizeVaultEndedWithdrawals` thinking that they are the first one in the next block to request withdrawal. In this case, all of the calls will revert except the first one.

Impact

The design of `finalizeVaultEndedWithdrawals` and `withdraw` causes a coordination problem when many depositors are anticipating withdrawing after the finalizing of a vault that has ended that may waste gas and time from withdrawers.

Recommendations

Instead of reverting if the vault-end withdrawal has already been finalized, call `vaultEndedWithdraw` so that the user's withdrawal still goes through.

Alternatively, instead of having this be a separate function that needs to be specifically called on the first withdrawal, integrate this logic with the `vaultEndedWithdraw` flow such that users need to only call `withdraw` and the first such transaction that ends up in the block sorts out the withdrawal.

Or, if the suggestion in Discussion 4. ↗ is implemented, then this would not be an issue since the ending of a vault would not require an asynchronous use of the withdrawal queue.

Remediation

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

- `1e5ae48b` ↗
- `13b8a030` ↗

3.14. Unnecessary precision loss in protocol fee

Target	LidoVault		
Category	Protocol Risks	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

When a variable-ongoing withdrawal is requested, the variable `variableToWithdrawnStakingEarnings` is multiplied by the inverse of the protocol fee and then set to the current state minus the protocol fee applied to the current state:

```
if (lidoStETHBalance > fixedETHDeposits) {
  (uint256 currentState, uint256 ethAmountOwed)
    = calculateVariableWithdrawState(
    (lidoStETHBalance - fixedETHDeposits) + withdrawnStakingEarnings
    + totalProtocolFee,
    variableToWithdrawnStakingEarnings[msg.sender].mulDiv(10000,
    10000 - protocolFeeBps) // rounds down!
  );

  if (ethAmountOwed >= minStETHWithdrawalAmount()) {
    // estimate protocol fee and update total - will actually be applied on
    withdraw finalization
    uint256 protocolFee = ethAmountOwed.mulDiv(protocolFeeBps, 10000);
    totalProtocolFee += protocolFee;

    withdrawnStakingEarnings += ethAmountOwed - protocolFee;
    variableToWithdrawnStakingEarnings[msg.sender] = currentState
    - currentState.mulDiv(protocolFeeBps, 10000);
```

However, precision loss happens when a quantity is multiplied by `10000/(10000 - protocolFeeBps)`, then an amount is added to it, and then the amount is subtracted from itself multiplied again by `protocolFeeBps / 10000`.

This precision loss is entirely unnecessary, because more exact quantities already exist in memory in the `ethAmountOwed` and `protocolFee` variables.

Impact

Unnecessary precision loss affects the estimated protocol fee. These extra multiplications and divisions cost gas while reducing the accuracy of the estimation.

Recommendations

Note that because of the implementation of `calculateVariableWithdrawState`, the return value will have the relation that `ethAmountOwed = currentState - previousWithdrawnAmount`.

So, what the new `variableToWithdrawnStakingEarnings[msg.sender]` is set to is equal to this:

```
= currentState - currentState.mulDiv(protocolFeeBps, 10000);
= currentState.mulDiv(10000 - protocolFeeBps, 10000)
= (ethAmountOwed + previousWithdrawnAmount).mulDiv(10000 - protocolFeeBps,
    10000)
= ethAmountOwed.mulDiv(10000 - protocolFeeBps, 10000)
    + previousWithdrawnAmount.mulDiv(10000 - protocolFeeBps, 10000)
= ethAmountOwed.mulDiv(10000 - protocolFeeBps, 10000) +
      variableToWithdrawnStakingEarnings[msg.sender].mulDiv(10000,
    10000 - protocolFeeBps).mulDiv(10000 - protocolFeeBps, 10000)
// (assuming no precision loss)
= ethAmountOwed.mulDiv(10000 - protocolFeeBps, 10000)
    + variableToWithdrawnStakingEarnings[msg.sender]
= variableToWithdrawnStakingEarnings[msg.sender]
    + ethAmountOwed.mulDiv(10000 - protocolFeeBps, 10000)
= variableToWithdrawnStakingEarnings[msg.sender] + ethAmountOwed
    - ethAmountOwed.mulDiv(protocolFeeBps, 10000)
= variableToWithdrawnStakingEarnings[msg.sender] + ethAmountOwed - protocolFee
```

Therefore, assuming no precision loss, that line of code is equivalent to this:

```
variableToWithdrawnStakingEarnings[msg.sender] += ethAmountOwed - protocolFee;
```

Additionally, this matches the line of code before it.

Remediation

This issue has been acknowledged by Saffron, and a fix was implemented in commit 4828dd90 ↗.

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. Consider transferring stETH instead of ETH

At a very high level, the LidoVault contract implements a way for fixed-side and variable-side depositors to trustlessly coordinate and execute a mutually beneficial interest-rate swap on underlying stETH held by the contract.

The way this contract is implemented draws heavily from the assumption that the users of the contract will only handle native ETH, and so the contract is the only party interacting with Lido to stake the ETH into stETH, to hopefully generate the yield, and then unstake the stETH via the withdrawal queue to pay users back in ETH.

Contract complexity

However, we would like to point out that much of this contract's complexity come from this hard requirement. Specifically,

- The withdrawal process from stETH to native ETH requires using the Lido withdrawal queue, which incurs a time delay of one or more days, typically.
- This time delay means withdrawals must be finalized asynchronously, and this happens independently of the vault's start-end timeline, which adds considerable business-logic complexity.
- During the withdrawal process, Lido can potentially default, and withdrawals are not guaranteed to return all of the value transferred when the withdrawal was initiated. Moreover, since stETH does not earn interest while it is in the withdrawal queue, any use of the withdrawal queue means counterparty risk is incurred without being paid for.
- All fixed-side early withdrawals must be finalized before any variable-side depositor can withdraw due to the early exit fee being calculated after the withdrawal. This means the vault-finalization process cannot be constant-time without even more complex logic.

Economical inefficiencies

On the other hand, we would like to note that this requirement is actually not fundamentally necessary to implement the financial instrument the vault is supposed to implement. Specifically,

- While stETH is in the withdrawal queue, it does not earn interest, which means that whenever an amount of stETH would be transferred to the withdrawal queue, it is not any more risky to just transfer it away to a user.
- Users, even nontechnical users, are likely familiar with stETH as an ERC-20 token and probably have no issues manipulating it. If user-friendliness is a concern, we would rec-

ommend implementing a second, less-trusted forwarder contract whose role is to either stake or implement the unstaking of stETH (handling later asynchronous claim calls) on behalf of users who need this functionality.
- Users who receive stETH from the contract for any reason can choose to swap it on the open market for ETH, instead of using the withdrawal queue, which saves them the opportunity cost of interest during the time it spends in the withdrawal queue.
- If the vault both sent users stETH and took deposits in stETH, there would be no downtime for any user, variable or fixed, who wishes to continuously use the LidoVault. In the current implementation, a user who wants to continuously use Saffron's product would have to wait for stETH to be withdrawn only to immediately cause it to be deposited again by their next LidoVault. This is inefficient and also makes the implementation of perpetuals, mentioned later in the Saffron roadmap, easier.

Specific recommendations

More specifically, for all of the above reasons, we recommend the following:

1. Allow initial fixed deposits in stETH.

During the deposit process, fixed-side depositors should have the option of depositing stETH instead of submitting native ETH and having the contract call `lido.submit`. This does not add any risk to anybody, because the contract currently unconditionally calls `lido.submit` and only increases flexibility.

2. Allow initial fixed withdrawals in stETH.

Before the vault starts, if a fixed depositor wishes to back out, they should be able to immediately withdraw stETH. In the current implementation, these users are entitled to the entire amount of native ETH obtained after the withdrawal of a fixed amount of stETH completes, so directly transferring that stETH to the user makes no difference risk-wise to any other party. Meanwhile, it decreases the risk for the caller since they gain the option of externally swapping the stETH for ETH.

3. Allow ongoing fixed withdrawals in stETH by charging the early exit fee in stETH.

During the vault's active period, if a fixed depositor wishes to back out, if the early exit fee is charged in stETH, the caller can immediately be given stETH corresponding to their original deposit, minus the early exit fees.

This does change the economic design, because in the current design, if a fixed depositor requests withdrawal, and then before the withdrawal queue cycles once, Lido defaults, then the fixed depositor pays for that default out of value they would have obtained from the withdrawal due to the fee being denominated in ETH now being a larger proportion of the withdrawn native ETH.

However, this risk overlaps with risk that the fixed depositor was already compensated for. Since variable depositors can immediately withdraw their share of any value in the contract as a whole that

is above the fixed deposit capacity, assuming that variable depositors always withdraw whenever there is yield to withdraw, this is not actually bad.

Moreover, if the next recommendation is implemented, and variable depositors can withdraw their share of the early exit fee in stETH, then any variable depositor who becomes concerned about a Lido default can themselves withdraw that value and put it through the Lido withdrawal queue if they wish.

4. Allow ongoing and after-ending variable withdrawals in stETH by charging protocol fees in stETH.

During the vault's active period and after the vault finishes, protocol fees should be charged in stETH.

The protocol fee is charged as a percentage of variable-side rewards. Therefore, no matter what the Lido share price actually is, because multiplication is commutative, charging it in stETH is equivalent to the current design of charging it in native ETH.

So, variable depositors gain the flexibility of externally swapping stETH for ETH instead of having to lock it up. The protocol-fee receiver also gains this flexibility.

5. Allow after-ending fixed withdrawals in stETH.

After the vault period has ended, fixed-side depositors should be given their deposit back in stETH rather than ETH.

This significantly changes the economic design, because in the current design, if Lido defaults while the vault finalization withdrawal is in the withdrawal queue, fixed-side users are still entitled to their original deposit in ETH after it is all finalized.

However, this does not actually change the economic design under the assumption that variable-side depositors continuously withdraw from the vault whenever there is value to be withdrawn.

Therefore, the only major benefit that fixed-side depositors get out of the current design, which they would not get if the vault were settled in stETH without using the withdrawal queue, is the benefit of insurance during the extra risk during the finalization process. This extra risk is something that variable-side depositors can freely opt out of donating to the fixed side, if they notice this economic discontinuity and make sure to send in a withdrawal right before the vault ends.

Summary

In summary, currently, the design of the LidoVault mandates the depositing of native ETH into Lido at the start, and it mandates the use of the withdrawal queue before any value can be taken out. However, we believe it is a better design to have the vault transact in stETH as much as possible, except for the fixed upfront premium.

If all of the above suggestions are taken, then the vault's business logic can be simplified considerably, because the vault no longer needs to deposit or withdraw Lido shares at all.

In order to maintain user-friendliness, if having users manipulate stETH is a significant concern, then a separate multicall-like contract can be deployed that handles holding a balance of Lido shares on behalf of users, depositing ETH into stETH, and claiming native ETH from the withdrawal queue after the owner of those stETH shares call for their withdrawal. This allows the withdrawal timeline to be managed completely independently from the LidoVault timelines and lifecycles, which improves usability and reduces the risk of implementation errors.

4.2. Confusing math in function `finalizeVaultEndedWithdrawals`

The function `finalizeVaultEndedWithdrawals` finalizes the withdrawals for the vault ending. In it, there is some math:

```
function finalizeVaultEndedWithdrawals(uint256 side) external {
  // [...]

  vaultEndedFixedDepositsFunds = claimWithdrawals(msg.sender,
    vaultEndedWithdrawalRequestIds);
  uint256 fixedETHDeposit = fixedETHDepositTokenTotalSupply;

  if (fixedETHDeposit < vaultEndedFixedDepositsFunds) {
    vaultEndedStakingEarnings = vaultEndedFixedDepositsFunds
    - fixedETHDeposit;
    vaultEndedFixedDepositsFunds -= vaultEndedStakingEarnings;
  }

  // [...]
}
```

This math is confusing and needlessly circuitous. In the case where `fixedETHDeposit < vaultEndedFixedDepositsFunds`, the second line subtracts `vaultEndedStakingEarnings` from `vaultEndedFixedDepositsFunds`, but the former quantity is the latter quantity minus `fixedETHDeposit`, which means it is equivalent to just setting it to `fixedEthDeposit`.

Moreover, reading and writing storage costs more gas than local variables. We recommend refactoring this math to an equivalent calculation that is more clear and manipulates storage less, such as this:

```
uint256 amountWithdrawn = claimWithdrawals(msg.sender,
    vaultEndedWithdrawalRequestIds);
uint256 fixedETHDeposit = fixedETHDepositTokenTotalSupply;

if (amountWithdrawn > fixedETHDeposit) {
  vaultEndedStakingEarnings = amountWithdrawn - fixedETHDeposit;
```

```
    vaultEndedFixedDepositsFunds = fixedETHDeposit;
  } else {
    vaultEndedFixedDepositsFunds = amountWithdrawn;
  }
```

This issue has been acknowledged by Saffron, and a fix was implemented in commit 3552ae7e ↗.

4.3. Premium claim requirement ensures safety too

In the `withdraw` function, for the case where a vault has started and a fixed-side depositor is withdrawing, there is an interesting comment:

```
function withdraw(uint256 side) external {
  require(side == FIXED || side == VARIABLE, "IS");

  // Vault has not started
  if (!isStarted()) {
    // [...]

    // Vault started and in progress
  } else if (!isEnded()) {
    if (side == FIXED) {
      // [...]

      // require that they have claimed their upfront premium to simplify this
    flow
      uint256 bearerBalance = fixedBearerToken[msg.sender];
      require(bearerBalance > 0, "NBT");
```

The comment indicates that the `"NBT"` requirement is to simplify the workflow, so that the exit fee can be charged correctly. However, we would like to highlight that this check is an important safety check too, for a nonobvious reason.

Consider a vault that has not started yet, from which an existing fixed depositor wishes to make a withdrawal before it starts. If they naively sign a `withdraw` call and broadcast it, a malicious front-runner can front-run the withdrawal with their own transactions that complete the vault. This causes the vault to start, and therefore the front-runner charges them a very large early exit fee that they were not expecting to pay.

Normally, this would be mitigated by having a different `withdraw` function, or an additional argument to the function, that includes in the transaction calldata the expected state of the vault — so that a user cannot sign a `withdraw` call expecting a vault state in which their withdraw does not incur fees,

but it is applied to a state in a way that assumes they assented to the fee. However, the requirement that the premium be collected effectively accidentally does this, because a user who never thought the vault had started of course would not have collected the premium yet.

This issue has been acknowledged by Saffron, and a fix was implemented in commit 4f1668a0 ↗.

4.4. Comment implies more centralization than exists

In the preamble to the `isEnded()` view function, there is a notice:

```
/// @notice True if the vault has ended
/// or the admin has settled all debts thus effectively ending the vault
function isEnded() public view returns (bool) {
  return (endTime > 0 && block.timestamp > endTime);
}
```

However, this likely refers to an older version of the contract as there is no admin. The current version is less centralized, so removing this comment would reduce alarm from casual readers of the contract source.

This issue has been acknowledged by Saffron, and a fix was implemented in commit 32b5db84 ↗.

4.5. Use batch withdrawals for gas savings

When the Lido withdrawal queue is called to claim withdrawals, `claimWithdrawal` is called in a loop:

```
function _claimWithdrawals(address user, uint256[] memory requestIds)
    internal returns (uint256) {
  uint256 beforeBalance = address(this).balance;

  // Claim Ether for the burned stETH positions
  // this will fail if the request is not finalized
  for (uint i = 0; i < requestIds.length; i++) {
    lidoWithdrawalQueue.claimWithdrawal(requestIds[i]);
  }

  uint256 withdrawnAmount = address(this).balance - beforeBalance;
  require(withdrawnAmount > 0, "IWA");
```

```
    emit WithdrawalClaimed(withdrawnAmount, requestIds, user);

    return withdrawnAmount;
  }
```

However, according to the Lido documentation ↗, if a caller wishes to claim multiple withdrawals, the recommended way to do this is to call `claimWithdrawals(uint256[] _requestIDs, uint256[] _hints)`.

This saves gas by not having to emit multiple external calls. The hints can be passed from the caller all the way into this function after having been computed off chain.

4.6. Unused event `VaultCodeSet`

In VaultFactory, the `VaultCodeSet` event is defined; however, it is never emitted.

```
contract VaultFactory is ILidoVaultInitializer {
  // [...]

  /// @notice Emitted when the vault bytecode is set
  /// @param creator Address of creator
  event VaultCodeSet(address indexed creator);
```

We recommend emitting this event at the appropriate time or removing this event definition.

This issue has been acknowledged by Saffron, and a fix was implemented in commit `0cdbc3d1` ↗.

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: LidoVault.sol

Function: `claimFixedPremium()`

This function is to change claim token to bearer token and transfer to the user premium from variable deposits.

Branches and code coverage

Intended branches

- Send `sendAmount` to `msg.sender`.
  - ☑ Test coverage

Negative behavior

- Reverts if the `isStarted()` is not set.
  - ☑ Negative test
- Reverts if the `claimBal` is smaller than one.
  - ☑ Negative test
- Reverts if the `claimBal` is false.
  - ☑ Negative test

Function call analysis

- `Math.mulDiv(this.fixedETHDepositToken[msg.sender], this.variableSideCapacity, this.fixedSideCapacity)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `deposit(uint256 side)`

This function divides into `FIXED` and `VARIABLE` to deposit. In `FIXED`, mint claim tokens and ETHDepositTokens. In `VARIABLE`, mint bearer tokens.

Inputs

- `side`
  - **Control**: Arbitrary.
  - **Constraints**: It is expected `FIXED` or `VARIABLE` value.
  - **Impact**: The value of side.

Branches and code coverage

Intended branches

- Amount is not zero.
  - ☑ Test coverage
- `isStarted()` is not set.
  - ☑ Test coverage
- Side is divided into `FIXED` and `VARIABLE`.
  - ☑ Test coverage

Negative behavior

- Reverts if the `fixedSideCapacity` is zero.
  - ☑ Negative test
- Reverts if the `!isStarted()` is not zero.
  - ☑ Negative test
- Reverts if the `side` is not `FIXED` or `VARIABLE`.
  - ☑ Negative test
- Reverts if the `msg.value` is smaller than `minimumDepositAmount`.
  - ☑ Negative test
- Reverts if the `amount` is smaller than `minimumFixedDeposit`.
  - ☑ Negative test
- Reverts if the `amount` is larger than `fixedSideCapacity - fixedETHDepositTokenTotalSupply`.
  - ☑ Negative test
- Reverts if the `remainingCapacity` is smaller than `minimumFixedDeposit` and not zero.
  - ☑ Negative test
- Reverts if the `shares` is smaller than one.
  - ☑ Negative test
- Reverts if the `stETHReceived` is smaller than `amount` and `amount - stETHReceived` is larger than `LIDO_ERROR_TOLERANCE_ETH`.
  - ☑ Negative test
- Reverts if the `amount` is larger than `variableSideCapacity - variableBearerTokenTotalSupply`.
  - ☑ Negative test
- Reverts if the `remainingCapacity` is smaller than `minimumDepositAmount` and not zero.
  - ☑ Negative test

Function call analysis

- `Math.mulDiv(this.fixedSideCapacity, this.minimumFixedDepositBps, 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.stakingBalance() -> LidoVault.lido.balanceOf(address(this))`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `LidoVault.lido.submit{value: amount}`
  - **What is controllable?** `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** It can be misused. It is caused reentrancy.

Function: `finalizeVaultEndedWithdrawals(uint256 side)`

This function is to finailize withdrawals when the vault is ended.

Inputs

- `side`
  - **Control**: Arbitrary.
  - **Constraints**: It is expected `FIXED` or `VARIABLE` value.
  - **Impact**: The value of side.

Function call analysis

- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> this.claimWithdrawals(msg.sender, requestIds) - > this._claimWithdrawals(user, requestIds) -> LidoVault.lidoWithdrawalQueue.claimWithdrawal(requestIds[i])`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(this.endTime > timestampRequested ? this.endTime - timestam-`

`pRequested : 0, 1000000000000000000, this.duration)`

- **What is controllable?** None.
- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000), 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, timestampRequested - this.startTime, this.duration)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fix -> Math.min(earlyExitFees, amountWithdrawn)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.applyProtocolFee(this.vaultEndedStakingEarnings) -> Math.mulDiv(stakingEarnings, this.protocolFeeBps, 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.stakingBalance() -> LidoVault.lido.balanceOf(address(this))`
  - **What is controllable?** `side`.

- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.vaultEndedWithdraw(side) -> this.requestEntireBalanceWithdraw(msg.sender) -> this.requestWithdrawViaETH(user, stETHAmount) -> this._requestWithdraw(user, stETHAmount) -> LidoVault.lido.approve(address(LidoVault.lidoWithdrawalQueue), stETHAmount)`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.requestEntireBalanceWithdraw(msg.sender) -> this.requestWithdrawViaETH(user, stETHAmount) -> this._requestWithdraw(user, stETHAmount) -> LidoVault.lidoWithdrawalQueue.requestWithdrawals(amounts, address(this))`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> Math.mulDiv(this.fixedBearerToken[msg.sender], this.vaultEndedFixedDepositsFunds, this.fixedLidoSharesTotalSupply())`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.calculateVariableWithdrawState(totalEarning this.variableToWithdrawnStakingEarnings[msg.sender]) -> Math.mulDiv(bearerBalance, totalEarnings, this.variableSideCapacity)`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `finalizeVaultNotStartedFixedWithdrawals()`

This function is to finalize the fixed withdrawal with `requestIds` if not started.

Function call analysis

- `this.claimWithdrawals(msg.sender, requestIds) -> this._claimWithdrawals(user, requestIds) -> LidoVault.lidoWithdrawalQueue.claimWithdrawal(requestIds[i])`
  - **What is controllable?** None.

- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `finalizeVaultOngoingFixedWithdrawals()`

This function is to finalize the fixed withdrawal with `requestIds`.

Function call analysis

- `this.claimFixedVaultOngoingWithdrawal(msg.sender) -> this.claimWithdrawals(msg.sender, requestIds) -> this._claimWithdrawals(user, requestIds) -> LidoVault.lidoWithdrawalQueue.claimWithdrawal(requestIds[i])`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimFixedVaultOngoingWithdrawal(msg.sender) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(this.endTime > timestampRequested ? this.endTime - timestampRequested : 0, 1000000000000000000, this.duration)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimFixedVaultOngoingWithdrawal(msg.sender) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000), 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimFixedVaultOngoingWithdrawal(msg.sender) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.claimFixedVaultOngoingWithdrawal(msg.sender) ->`

`this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, timestampRequested - this.startTime, this.duration)`

- **What is controllable?** None.
- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.claimFixedVaultOngoingWithdrawal(msg.sender) -> Math.min(earlyExitFees, amountWithdrawn)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `finalizeVaultOngoingVariableWithdrawals()`

This function is to finalize the variable withdrawal with `requestIds`.

Function call analysis

- `this.claimWithdrawals(msg.sender, requestIds) -> this._claimWithdrawals(user, requestIds) -> LidoVault.lidoWithdrawalQueue.claimWithdrawal(requestIds[i])`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.applyProtocolFee(amountWithdrawn) -> Math.mulDiv(stakingEarnings, this.protocolFeeBps, 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.calculateVariableFeeEarningsShare() -> this.calculateVariableWithdrawState( + this.withdrawnFeeEarnings, this.variableToWithdrawnFees[msg.sender]) -> Math.mulDiv(bearerBalance, totalEarnings, this.variableSideCapacity)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `getFixedOngoingWithdrawalRequestIds(address user)`

This function is to get `RequestIds` of fixed-side withdraw requests after vault ended.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The value of `RequestIds` from fixed-side withdraw requests for user.

Function: `getFixedOngoingWithdrawalRequestTimestamp(address user)`

This function is to get the timestamp of fixed-side withdraw requests after vault ended.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The value of the timestamp from fixed-side withdraw requests for the user.

Function: `initialize(InitializationParams params)`

This function is to initialize new LidoVault.

Inputs

- `params`
  - **Control**: Arbitrary.
  - **Constraints**: The values in params is not zero.
  - **Impact**: The values required to initialize.

Branches and code coverage

Intended branches

- Initialize if the params are not zero.
  - ☑ Test coverage

Negative behavior

- Reverts if the `params.protocolFeeReceiver` address is zero.
  - ☑ Negative test
- Reverts if the `params.earlyExitFeeBps` is zero.
  - ☑ Negative test
- Reverts if the `params.vaultId` is zero.
  - ☑ Negative test
- Reverts if the `params.duration` is zero.
  - ☑ Negative test
- Reverts if the `params.fixedSideCapacity` is zero.
  - ☑ Negative test
- Reverts if the `params.variableSideCapacity` is zero.
  - ☑ Negative test
- Reverts if the `fixedSideCapacity.mulDiv(minimumFixedDepositBps, 10_000)` is smaller than `minimumDepositAmount`.
  - ☑ Negative test
- Reverts if the `fixedSideCapacity` is not zero.
  - ☑ Negative test

Function call analysis

- `Math.mulDiv(params.fixedSideCapacity, this.minimumFixedDepositBps, 10000)`
  - **What is controllable?** `fixedSideCapacity`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `isEnded()`

This function returns true if the vault has ended.

Function: `isStarted()`

This function returns true if the vault has started.

Function: `maxStETHWithdrawalAmount()`

This function is to return constant maximum amount of stETH.

Function: `minStETHWithdrawalAmount()`

This function is to return constant minimum amount of stETH.

Function: `stakingBalance()`

This function is to return the balance of Lido from `address(this)`.

Function: `withdraw(uint256 side)`

This function is to withdraw that withdraw is divided into `FIXED` or `VARIABLE` and vault conditions.

Inputs

- `side`
  - **Control**: Arbitrary.
  - **Constraints**: It is expected `FIXED` or `VARIABLE` value.
  - **Impact**: The value of side.

Branches and code coverage

Intended branches

- Before vault starts, during, and after it ends, execute withdraw divided into each `side`.
  - ☑ Test coverage

Negative behavior

- Reverts if the `side` is not `FIXED` or `VARIABLE`.
  - ☑ Negative test
- Reverts if all kinds of `WithdrawalRequestIds`s' length is not zero.
  - ☑ Negative test
- Reverts if the `claimBalance` is smaller than one.
  - ☑ Negative test
- Reverts if the `sendAmount` is smaller than one.
  - ☑ Negative test
- Reverts if the `feeEarningsShare` is smaller than one.
  - ☑ Negative test

Function call analysis

- `this.requestWithdrawViaShares(msg.sender, claimBalance) -> LidoVault.lido.getPooledEthByShares(shares)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.requestWithdrawViaShares(msg.sender, claimBalance) -> this.requestWithdrawViaETH(user, stETHAmount) -> this._requestWithdraw(user, stETHAmount) -> LidoVault.lido.approve(address(LidoVault.lidoWithdrawalQueue), stETHAmount)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.requestWithdrawViaShares(msg.sender, claimBalance) -> this.requestWithdrawViaETH(user, stETHAmount) -> this._requestWithdraw(user, stETHAmount) -> LidoVault.lidoWithdrawalQueue.requestWithdrawals(amounts, address(this))`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.stakingBalance() -> LidoVault.lido.balanceOf(address(this))`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `Math.mulDiv(lidoStETHBalance, initialDepositAmount, fixedETHDeposits)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.calculateVariableWithdrawState(lidoStETHBalance - fixedETHDeposits + this.withdrawnStakingEarnings + this.totalProtocolFee, Math.mulDiv(this.variableToWithdrawnStakingEarnings[msg.sender], 10000, 10000 - this.protocolFeeBps)) -> Math.mulDiv(bearerBalance, totalEarnings, this.variableSideCapacity)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `Math.mulDiv(this.variableToWithdrawnStakingEarnings[msg.sender], 10000, 10000 - this.protocolFeeBps)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `Math.mulDiv(ethAmountOwed, this.protocolFeeBps, 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?**

None.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `Math.mulDiv(currentState, this.protocolFeeBps, 10000)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> this.claimWithdrawals(msg.sender, requestIds) -> this._claimWithdrawals(user, requestIds) -> LidoVault.lidoWithdrawalQueue.claimWithdrawal(requestIds[i])`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(this.endTime > timestampRequested ? this.endTime - timestampRequested : 0, 1000000000000000000, this.duration)`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000), 10000)`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(1 + this.earlyExitFeeBps, remainingProportion, 1000000000000000000)`
  - **What is controllable?** `side`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals()`

`-> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> this.calculateFixedEarlyExitFees(upfrontPremium, request.timestamp) -> Math.mulDiv(upfrontPremium, timestampRequested - this.startTime, this.duration)`
    - **What is controllable?** `side`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> this.claimOngoingFixedWithdrawals() -> this.claimFixedVaultOngoingWithdrawal(fixedUser) -> Math.min(earlyExitFees, amountWithdrawn)`
    - **What is controllable?** `side`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.vaultEndedWithdraw(side) -> Math.mulDiv(this.fixedBearerToken[msg.sender], this.vaultEndedFixedDepositsFunds, this.fixedLidoSharesTotalSupply())`
    - **What is controllable?** `side`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.2. Module: VaultFactory.sol

Function: `createVault(uint256 _fixedSideCapacity, uint256 _duration, uint256 _variableSideCapacity)`

This function is to create a new LidoVault with initialize value.

Inputs

- `_fixedSideCapacity`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The value of fixed-side capacity.
- `_duration`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The value of staking duration.
- `_variableSideCapacity`
    - **Control**: Arbitrary.
    - **Constraints**: None.

- **Impact**: The value of variable-side capacity.

Branches and code coverage

Intended branches

- Create vault with initialized params.

Negative behavior

- Reverts if the `vaultAddress` is zero.

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 Lido Fixed Income contracts, we discovered 14 findings. Two critical issues were found. Three were of high impact, two were of medium impact, five were of low impact, and the remaining findings were informational in nature.

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.