Saffron Security Review

Pashov Audit Group

Conducted by: Peakbolt, immeas, sashik_eth

January 22nd 2024 - January 28th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **lido-fiv** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About lido-fiv

Saffron Lido Fixed Income Vaults enable fixed and variable yield exchange. Yield for Saffron Lido Vaults is produced from Lido ETH Staking. Fixed side participants are paid an up-front fixed amount for depositing ETH into the vault, which is then staked on Lido. The variable side likewise pays a fixed amount to earn all staking earnings generated over the duration of the vault's lifetime. Parties agree ahead of time on terms, including the lockup duration, total asset value, and fixed payment amount.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **-** **064791d13381af094a921970a0e280961c0049ad**

fixes review commit hash **-** **61a8273ec2b941651a444b645084b875ac1b741a**

Scope

The following smart contracts were in scope of the audit:

- `AdminLidoAdapter`
- `Initialized`
- `LidoAdapter`
- `LidoVault`
- `NonTransferrableVaultBearerToken`
- `VaultBearerToken`
- `VaultFactory`
- `interfaces/**`

7. Executive Summary

Over the course of the security review, Peakbolt, immeas, sashik_eth engaged with Saffron to review lido-fiv. In this period of time a total of **22** issues were uncovered.

Protocol Summary

Protocol Name	lido-fiv
Repository	https://github.com/saffron-finance/lido-fiv
Date	January 22nd 2024 - January 28th 2024
Protocol Type	Fixed Income Vaults

Findings Count

Severity	**Amount**
Critical	3
High	9
Medium	7
Low	3
Total Findings	**22**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect withdrawal amount computation for variable participants	Critical	Resolved
[C-02]	Fixed participants can claim more premium than expected	Critical	Resolved
[C-03]	Premium could be stolen by fixed side participant	Critical	Resolved
[H-01]	finalizeVaultEndedWithdrawals() will fail when last withdrawal request is less than 100 wei	High	Resolved
[H-02]	finalizeVaultEndedWithdrawals would be DOSed if at least one fixed participant would withdraw from the ongoing vault	High	Resolved
[H-03]	Fixed participants will be incorrectly penalized during adminSettleDebt timelock period	High	Resolved
[H-04]	Fixed participants will earn full premium even when vault ends early from debt settlement	High	Resolved
[H-05]	Funds in vault can be temporarily locked by spamming dust deposits and withdrawals	High	Resolved
[H-06]	Variable participants could lose their fee earnings	High	Resolved
[H-07]	Admin could incur a loss with debt settlement	High	Resolved
[H-08]	Protocol will miss part of it's fee	High	Resolved

[H-09]	Funds could stuck in lidoAdapter if the user requests withdrawal before adminSettleDebt() execution and finalized after	High	Resolved
[M-01]	Deposit function can be DOSed	Medium	Acknowledged
[M-02]	Race condition with initiatingAdminSettleDebt() could under-compensate participants	Medium	Resolved
[M-03]	adminSettleDebtLockPeriod might not be long enough	Medium	Resolved
[M-04]	adminSettleDebt() could be called multiple times leading to the loss of funds	Medium	Resolved
[M-05]	Admin could call adminSettleDebt anytime without previous call initiatingAdminSettleDebt	Medium	Resolved
[M-06]	Accrual before vault start can be used as honeypot	Medium	Acknowledged
[M-07]	ongoing withdrawals from variable side hurts variable earnings	Medium	Acknowledged
[L-01]	Unnecessary division before multiplication	Low	Resolved
[L-02]	early exit can lock funds in contract	Low	Resolved
[L-03]	Missing check for revoked adapter in createVault()	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect withdrawal amount computation for variable participants

Severity

Impact: High, wrong withdrawal amount can cause vault from servicing all withdrawals

Likelihood: High, always occur as variable participants will withdraw earnings

Description

When `vaultEndedWithdraw()` is called by variable participants, it determines the `sendAmount` to be withdrawn based on the participant's share of `totalEarnings` and `feeEarnings` using `calculateVariableWithdrawState()`.

Both `totalEarnings` and `feeEarnings` have a withdrawn component, `withdrawnStakingEarnings` and `withdrawnFeeEarnings`. These are used to keep track of staking/fee earnings withdrawn by variable participants when vault is on-going.

The issue is that both `withdrawnStakingEarnings` and `withdrawnFeeEarnings` are not reduced in `vaultEndedWithdraw()`, while `vaultEndedStakingEarnings` and `feeEarnings` are reduced. As they are used to determine `totalEarnings` and `feeEarnings`, the inconsistent behaviour will cause the share calculation to be incorrect.

As `variableBearerToken` is burned in `vaultEndedWithdraw()`, both `totalEarnings` and `feeEarnings` are supposed to be reduced accordingly. However, as `withdrawnStakingEarnings` and `withdrawnFeeEarnings` are not reduced, subsequent `vaultEndedWithdraw()` will end up with a `sendAmount`

that is higher than expected. That will lead to insufficient funds in the vault to service all withdrawals.

```
function vaultEndedWithdraw(uint256 side) internal {
      ...
      //@audit totalEarnings will not be reduced correctly as
      // withdrawnStakingEarnings is not reduced

            uint256 totalEarnings = withdrawnStakingEarnings + vaultEndedStakingEarni

      if (totalEarnings > 0) {
        //@audit stakingEarningsShare will be wrong as totalEarnings will be
        // incorrect
        (
          uint256currentState,
          uint256stakingEarningsShare
        ) = calculateVariableWithdrawState(
          totalEarnings,
          variableToWithdrawnStakingEarnings[msg.sender]
        );
        stakingShareAmount = stakingEarningsShare;
        variableToWithdrawnStakingEarnings[msg.sender] = currentState;
      }

      uint256 feeShareAmount = 0;
      //@audit totalFees will not be reduced correctly as withdrawnFeeEarnings
      // is not reduced
      uint256 totalFees = withdrawnFeeEarnings + feeEarnings;

      if (totalFees > 0) {
        //@audit feesShare will be wrong as totalFees will be incorrect
        (
          uint256currentState,
          uint256feesShare
        ) = calculateVariableWithdrawState(
          totalFees,
          variableToWithdrawnFees[msg.sender]
        );
        feeShareAmount = feesShare;
        variableToWithdrawnFees[msg.sender] = currentState;
      }

      //@audit only vaultEndedStakingEarnings and feeEarnings are reduced
      vaultEndedStakingEarnings -= stakingShareAmount;
      feeEarnings -= feeShareAmount;

      variableBearerToken.burn(msg.sender, bearerBalance);

      uint256 sendAmount = stakingShareAmount + feeShareAmount;
      sendFunds(sendAmount);

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

Recommendations

Ideally, we should have a consistent share calculation for simplicity, so it is recommended to use the same share amount computation as vault-on-going withdrawal.

That means for vault-end withdrawals, the `totalEarnings` and `totalFees` should not be reduced on withdrawals. And `variableSideCapacity` can be used instead of `variableBearerToken.totalSupply()` for the share amount computation, as `variableBearerToken` is still required to be burned.

[C-02] Fixed participants can claim more premium than expected

Severity

Impact: High, wrong premium amount will allow some participants to withdraw more than others

Likelihood: High, always occur as fixed participants will claim premiums

Description

`claimFixedPremium()` is called to allow fixed participants to claim their fixed premium when vault is started. And `sendAmount`, the premium amount to be claimed, is determined by the participant's share of `variableSideCapacity`, using their balance of `fixedClaimTokens` over `fixedLidoSharesTotalSupply()`.

As `fixedLidoSharesTotalSupply()` is the sum of `fixedBearerToken.totalSupply()` and `fixedClaimTokens.totalSupply()`, it will be reduced when fixed participants performs on-going vault withdrawals due to the fact that the participant's balance of `fixedBearerToken` will be burned.

This will cause an issue with the `sendAmount` calculation within `claimFixedPremium()`, giving a higher than expected share of the `variableSideCapacity` after vault withdrawal by fixed participants. This allows fixed participants to claim more premium than allowed by doing it after certain amount of withdrawals.

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

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

    //@audit fixedLidoSharesTotalSupply
    //() will reduces upon withdrawal, but variableSideCapacity is not.
    //       this will cause sendAmount to be higher for subsequent claims
    // Send a proportional share of the total variable side deposits
    //(premium) to the fixed side depositor
    uint256 sendAmount = claimBal.mulDiv(1e18, fixedLidoSharesTotalSupply
      ()).mulDiv(variableSideCapacity, 1e18);

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

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

    // Mint bearer token
    fixedBearerToken.mint(msg.sender, claimBal);

    // Burn claim tokens
    fixedClaimToken.burn(msg.sender, claimBal);

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

Recommendations

In `claimFixedPremium()`, use `fixedSideCapacity` instead of `fixedLidoSharesTotalSupply()` for L439 as follows:

```
uint256 sendAmount = fixedETHDepositToken.balanceOf(msg.sender).mulDiv
  (variableSideCapacity, fixedSideCapacity);
```

[C-03] Premium could be stolen by fixed side participant

Severity

Impact: High, since variable side participants could lose their deposits

Likelihood: High, since vector could be easily done

Description

The `vaultEndedWithdraw` function allows users to request withdrawal of all staking balance after the vault duration is ended (L754). In case if vault balance is less than Lido's minimum withdrawal amount - the function just puts the vault into the state of `vaultEndedWithdrawalsFinalized` (L752) since there is no need to create a new withdrawal request and now users are allowed to withdraw their earnings if there are any:

```
File: LidoVault.sol
746:   function vaultEndedWithdraw(uint256 side) internal {
747:     if
  (vaultEndedWithdrawalRequestIds.length == 0 && !vaultEndedWithdrawalsFinalized) {
748:       emit VaultEnded(block.timestamp, msg.sender);
749:
750:       if (lidoAdapter.stakingBalance
  () < lidoAdapter.minStETHWithdrawalAmount()) {
751:         // not enough staking ETH to withdraw just override vault ended
// state and continue the withdraw
752:         vaultEndedWithdrawalsFinalized = true;
753:       } else {

755:
756:         emit LidoWithdrawalRequested
  (msg.sender, vaultEndedWithdrawalRequestIds, side, isStarted(), isEnded());
757:         // need to call finalizeVaultEndedWithdrawals once request is
// processed
758:         return;
759:       }
760:     }
...
```

However, in the last case function failed to claim all requests that were created during an ongoing stage in the same way as `finalizeVaultEndedWithdrawals` do at L673-L676:

```
File: LidoVault.sol
665:   function finalizeVaultEndedWithdrawals
  (uint256 side) external nonReentrant {
666:     require(side == FIXED || side == VARIABLE, "IS");
667:     require(
  vaultEndedWithdrawalRequestIds.length!=0&&!vaultEndedWithdrawalsFinalized,
  "WNR"
);
668:
669:     vaultEndedWithdrawalsFinalized = true;
670:
671:     // claim any ongoing fixed withdrawals too
672:     uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
673:     for (uint i = 0; i < arrayLength; i++) {
674:       address fixedUser = fixedOngoingWithdrawalUsers[i];

676:     }
...
```

This could lead to the next scenario:

1. A new vault was created. Alice puts in it 90% of variable side capacity.

2. Bob put 100% of fixed side capacity and the last 10% of a variable. Vault starts.
3. Bob claims all premiums and requests ongoing withdrawal of his fixed deposit.
4. Vault ends, Alice calls the `withdraw` function that puts the vault into the `vaultEndedWithdrawalsFinalized` state and since earnings equal 0 - Alice also receives 0 for her 90% supply of `variableBearerToken` that is now burned.
5. Bob calls `finalizeVaultOngoingFixedWithdrawals` receiving his fixed side deposit minus premium and early exit fee.
6. Bob calls `withdraw` and since he is now the only variable side participant - he receives all premiums and early exit fees that are equal in sum to Alice's initial deposit.

Recommendations

Even if the `lidoAdapter.stakingBalance()` is less than the `lidoAdapter.minStETHWithdrawalAmount()` vault still should claim all fixed ongoing withdrawal requests in the `vaultEndedWithdraw` function.

8.2. High Findings

[H-01] `finalizeVaultEndedWithdrawals()` will fail when last withdrawal request is less than 100 wei

Severity

Impact: High, the issue will prevent withdrawal of stETH

Likelihood: Medium, when last withdrawal request is < 100 wei

Description

When LidoVault ends, `LidoVault.finalizeVaultEndedWithdrawals()` can be called to withdraw its entire stETH balance from Lido. That will trigger `LidoAdapter.requestEntireBalanceWithdraw()`, which will then calls `_requestWithdraw()` to perform the withdrawals. As the max withdrawal is 1000 stETH (MAX_STETH_WITHDRAWAL_AMOUNT), the withdrawal will be splitted into multiple withdrawal requests of 1000 stETH as shown in the code below. Note that the min withdrawal amount is 100 wei (in stETH), as shown by the require statement.

```
/// @notice Request a withdrawal on Lido to exchange stETH for ETH
  /// @param stETHAmount amount of stETH to withdraw
  /// @return requestIds Ids of the withdrawal requests
  function _requestWithdraw
    (address user, uint256 stETHAmount) internal returns (uint256[] memory) {
    require(stETHAmount >= MIN_STETH_WITHDRAWAL_AMOUNT, "WM");

    // Approve the withdrawal queue contract to pull the stETH tokens
    bool approved = lido.approve(address(lidoWithdrawalQueue), stETHAmount);
    require(approved, "AF");

    uint256[] memory amounts = new uint256[](calculateWithdrawals(stETHAmount));
    if (stETHAmount > MAX_STETH_WITHDRAWAL_AMOUNT) {
      uint256 amountLeft = stETHAmount;
      uint256 i = 0;
      while (amountLeft > 0) {
        if (amountLeft >= MAX_STETH_WITHDRAWAL_AMOUNT) {
          amounts[i] = MAX_STETH_WITHDRAWAL_AMOUNT;
          amountLeft -= MAX_STETH_WITHDRAWAL_AMOUNT;
        } else {
          amounts[i] = amountLeft;
          amountLeft = 0;
        }
        i++;
      }
    } else {
      amounts[0] = stETHAmount;
    }

    //@audit - the following will revert if last withdrawal request is less than
    // 100 wei
    // Submit the stETH to the withdrawal queue
    uint256[] memory requestIds = lidoWithdrawalQueue.requestWithdrawals
      (amounts, address(this));
    require(requestIds.length > 0, "IWR");

    emit WithdrawalRequested(stETHAmount, requestIds, user);

    return requestIds;
  }
```

The issue is the call `lidoWithdrawalQueue.requestWithdrawals(amounts, address(this))` will revert if the last withdrawal request is less than 100 wei (in stETH).

This will prevent `LidoVault` from completing the vault end withdrawals, causing all the stETH to be locked within the `LidoAdapter`.

Recommendations

Ensure that all withdrawal requests are at least 100 wei.

[H-02] `finalizeVaultEndedWithdrawals` would be DOSed if at least one fixed

participant would withdraw from the ongoing vault

Severity

Impact: Medium, since withdrawing from the vault would be blocked temporarily

Likelihood: High, since withdraws from ongoing vault could be expected as regular user flow

Description

Fixed participants can request a withdrawal from the ongoing vault using the `withdraw` function. It would invoke a withdrawal request on Lido on behalf of the user and push the address of such user to the `fixedOngoingWithdrawalUsers` array at L549:

```
File: LidoVault.sol
458:   function withdraw(uint256 side) external nonReentrant {
..
516:       // Vault started and in progress
517:     } else if (!isEnded()) {
518:       if (side == FIXED) {
519:         require(
520:
             fixedToVaultOngoingWithdrawalRequestIds[msg.sender].requestIds.length ==
521:
               fixedToVaultNotStartedWithdrawalRequestIds[msg.sender].length == 0,
522:           "WAR"
523:         );
524:
525:         // require that they have claimed their upfront premium to simplify
// this flow
526:         uint256 bearerBalance = fixedBearerToken.balanceOf(msg.sender);
527:         require(bearerBalance > 0, "NBT");
528:
529:         uint256 initialDepositAmount = fixedETHDepositToken.balanceOf
  (msg.sender);
530:         require(initialDepositAmount > 0, "NET");
531:
532:         // since the vault has started only withdraw their initial fixed
// deposit - unless we are in a loss
533:         uint256 withdrawAmount = initialDepositAmount;
534:         uint256 lidoStETHBalance = lidoAdapter.stakingBalance();
535:         uint256 fixedETHDeposits = fixedETHDepositToken.totalSupply();
536:
537:         if (fixedETHDeposits > lidoStETHBalance) {
538:           // our staking balance if less than our initial ETH deposits only
// return a proportional amount of the balance to the fixed user
539:           withdrawAmount = lidoStETHBalance.mulDiv
  (initialDepositAmount, fixedETHDeposits);
540:         }
541:
542:         fixedBearerToken.burn(msg.sender, bearerBalance);
543:         fixedETHDepositToken.burn(msg.sender, initialDepositAmount);
544:

546:           requestIds: lidoAdapter.requestWithdrawViaETH
  (msg.sender, withdrawAmount),
547:           timestamp: block.timestamp
548:         });
549:         fixedOngoingWithdrawalUsers.push(msg.sender);
...
```

Later user can finalize their withdrawal using the `finalizeVaultOngoingFixedWithdrawals` function that would call the `claimFixedVaultOngoingWithdrawal` helper function. The last one would claim the user's request, calculate the corresponding amount of fee, etc. Most importantly this helper function would delete the user's address from the `fixedOngoingWithdrawalUsers` array at L876:

```
File: LidoVault.sol
637:   function finalizeVaultOngoingFixedWithdrawals() external nonReentrant {
638:     uint256 sendAmount = claimFixedVaultOngoingWithdrawal(msg.sender);
639:
640:     sendFunds(sendAmount);
641:
642:     emit FixedFundsWithdrawn(sendAmount, msg.sender, isStarted(), isEnded
  ());
643:   }
...
863:   function claimFixedVaultOngoingWithdrawal
  (address user) internal returns (uint256) {
864:
       WithdrawalRequest memory request = fixedToVaultOngoingWithdrawalRequestIds[user
865:     uint256[] memory requestIds = request.requestIds;
866:     require(requestIds.length != 0, "WNR");
867:
868:     uint256 upfrontPremium = userToFixedUpfrontPremium[user];
869:
870:     delete userToFixedUpfrontPremium[user];
871:     delete fixedToVaultOngoingWithdrawalRequestIds[user];
872:
873:     uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
874:     for (uint i = 0; i < arrayLength; i++) {
875:       if (fixedOngoingWithdrawalUsers[i] == user) {
876:         delete fixedOngoingWithdrawalUsers[i];
877:       }
878:     }
..
```

Solidity `delete` keyword only assigns the array element to zero value, while the length of the array remains the same. So this deleting will cause the DOS in a later call to `finalizeVaultEndedWithdrawals`. At L675 contract would try to call the helper function for all addresses in the `fixedOngoingWithdrawalUsers` array, one of which now is `address(0)` that stays in the array after deleting the user address previously:

```
File: LidoVault.sol
665:   function finalizeVaultEndedWithdrawals
  (uint256 side) external nonReentrant {
...
671:     // claim any ongoing fixed withdrawals too
672:     uint256 arrayLength = fixedOngoingWithdrawalUsers.length;
673:     for (uint i = 0; i < arrayLength; i++) {
674:       address fixedUser = fixedOngoingWithdrawalUsers[i];

676:     }
...
```

`claimFixedVaultOngoingWithdrawal` on its turn would revert to the `address(0)` parameter since this address has no withdrawal requests associated with it (L866). Only the admin would be able to restore the vault withdrawing flow using the `adminSettleDebt` function.

Recommendations

`claimFixedVaultOngoingWithdrawal` should return 0 in case if `address(0)` is provided as `user` parameter.

[H-03] Fixed participants will be incorrectly penalized during adminSettleDebt timelock period

Severity

Impact: High, fixed participants will incur losses

Likelihood: Medium, occurs when admin settles debt

Description

When `initiatingAdminSettleDebt()` is called, the settle debt process is initialized. That will start the timelock of 3 days (based on `adminSettleDebtLockPeriod`), before the admin can call `adminSettleDebt()` and transfer the stETH balance to `AdminLidoAdapter`. I believe the timelock is designed to allow fixed/variable participants to withdraw before `adminSettleDebt()`, when the `settleDebtAmount` set during initialization does not properly compensate them (e.g. a rogue admin try to settle debt with a heavy loss).

However, when fixed participants withdraw their deposit after `initiatingAdminSettleDebt()` and before vault ends, they will be penalized with the early withdrawal fees as calculated in `calculateFixedEarlyExitFees()`. That is incorrect as it defeats the purpose of the timelock, causing fixed participants to be under-compensated regardless of the timelock.

```
function calculateFixedEarlyExitFees(
    uint256 upfrontPremium,
    uint256 timestampRequested
  ) internal view returns (uint256) {
    uint256 remainingProportion =
      (endTime > timestampRequested ? endTime - timestampRequested : 0).mulDiv(
      1e18,
      duration
    );


    // Calculate the scaling fee based on the quadratic scaling factor and
    // earlyExitFeeBps
    uint256 earlyExitFees = upfrontPremium.mulDiv(
      earlyExitFeeBps.mulDiv(remainingProportion.mulDiv
        (remainingProportion, 1e18), 1e18),
      10000
    );

    // Calculate the amount to be paid back of their original upfront claimed
    // premium, not influenced by quadratic scaling
    earlyExitFees += upfrontPremium - upfrontPremium.mulDiv
      (timestampRequested - startTime, duration);

    return earlyExitFees;
  }
```

Recommendations

After `initiatingAdminSettleDebt()` is called, `calculateFixedEarlyExitFees()` should not apply the scaling fee.

[H-04] Fixed participants will earn full premium even when vault ends early from debt settlement

Severity

Impact: High, premium can be stolen from variable participants

Likelihood: Medium, occurs when admin settles debt

Description

When `adminSettleDebt()` is called before the full vault duration, it will end the vault early, allowing all participants to withdraw their deposits.

In that scenario, the debt will be settled with a positive staking earnings, to ensure that variable participants can withdraw both earnings and part of the fixed premium based on the early vault end time.

The problem is, fixed participants do not pay back the partial premium based on remaining time (not in vault), allowing them to withdraw both full initial deposits and full fixed premium even when vault ended early. This is due to missing calculation in `vaultEndedWithdraw()` to return the partial premium, as shown below.

So variable participants will incur a loss as they are not paid back the partial premium.

Furthermore, a malicious admin can exploit this and earn the full premium by depositing and withdrawing as a fixed participant.

```
function vaultEndedWithdraw(uint256 side) internal {
    ...

    // have to call finalizeVaultEndedWithdrawals first
    require(vaultEndedWithdrawalsFinalized, "WNF");

    if (side == FIXED) {
      require(

                fixedToVaultOngoingWithdrawalRequestIds[msg.sender].requestIds.length
          fixedToVaultNotStartedWithdrawalRequestIds[msg.sender].length == 0,
        "WAR"
      );

      uint256 sendAmount = fixedToPendingWithdrawalAmount[msg.sender];

      // they submitted a withdraw before the vault had ended and the vault
      // ending should have claimed it
      if (sendAmount > 0) {
        delete fixedToPendingWithdrawalAmount[msg.sender];
      } else {
        uint256 bearerBalance = fixedBearerToken.balanceOf(msg.sender);
        require(bearerBalance > 0, "NBT");

        //@audit fixed participants will obtain the full deposit as
        // vaultEndedFixedDepositsFunds
             will be equal to fixedETHDepositToken.totalSupply
        //() when debt is settled with positive earnings
        sendAmount = fixedBearerToken.balanceOf(msg.sender).mulDiv(
          vaultEndedFixedDepositsFunds,
          fixedLidoSharesTotalSupply()
        );

        fixedBearerToken.burn(msg.sender, bearerBalance);
        fixedETHDepositToken.burn(msg.sender, fixedETHDepositToken.balanceOf
          (msg.sender));
        vaultEndedFixedDepositsFunds -= sendAmount;
      }

      sendFunds(sendAmount);

      emit FixedFundsWithdrawn(sendAmount, msg.sender, isStarted(), true);
      return;
    } else {
```

Recommendations

When admin settle debt ends the vault early, deduct `vaultEndedFixedDepositsFunds` by the amount of premium to be returned to variable participants as part of `feeEarnings`.

[H-05] Funds in vault can be temporarily locked by spamming dust deposits and withdrawals

Severity

Impact: Medium, funds in vault will be temporarily locked

Likelihood: High, can always occur

Description

`finalizeVaultEndedWithdrawals()` is required to be called to finalize all fixed withdrawals, including on-going fixed withdrawals.

The issue is that it iterates through `fixedOngoingWithdrawalUsers`, which can be manipulated and become unbounded. An attacker can make `fixedOngoingWithdrawalUsers` extremely large by spamming dust fixed deposits of 100 wei (with multiple EOA) and then withdraw them when vault is on-going. This will cause `finalizeVaultEndedWithdrawals()` to be DoS, which prevents withdrawals when vault ends, resulting in the vault funds locked.

I have classified this as Medium impact as admin can recover the issue with settle debt function.

Another issue that can occur with dust deposits and withdrawals is that an attacker can prevent vault from starting. The attack can conducted by spamming multiple 1 wei variable deposits and then withdraw one of them whenever vault is going to start by frontrunning the last depositor.

Recommendations

One possible mitigation is to increase the attack cost by setting a higher minimum deposit amount (e.g. 0.1 ETH) for fixed/variable participants.

Take note to ensure deposits do not cause unfilled capacity to be less than the minimum deposit amount, otherwise it will prevent vault from starting.

[H-06] Variable participants could lose their fee earnings

Severity

Impact: High, affected variable participants will lose their fee earnings

Likelihood: Medium, occurs when final total staking earnings is lower than on-going period

Description

`calculateVariableWithdrawState()` is used to calculate the `ethAmountOwed`, which is the balance of unwithdrawn earnings. This is computed using the variable particpant share of the `totalEarnings`, subtracted by `previousWithdrawnAmount` (see code below).

```
function calculateVariableWithdrawState(
    uint256 totalEarnings,
    uint256 previousWithdrawnAmount
  ) internal view returns (uint256, uint256) {
    uint256 bearerBalance = variableBearerToken.balanceOf(msg.sender);
    require(bearerBalance > 0, "NBT");

    //@audit this can revert if participant share of earnings is less than
    // previousWithdrawnAmount
    uint256 ethAmountOwed = bearerBalance.mulDiv
      (totalEarnings, variableBearerToken.totalSupply()) -
      previousWithdrawnAmount;

    return (ethAmountOwed + previousWithdrawnAmount, ethAmountOwed);
  }
```

The issue is that the computation assumes that `bearerBalance.mulDiv(totalEarnings, variableBearerToken.totalSupply())` will always be greater than `previousWithdrawnAmount`.

However, there are edge cases that breaks that assumption,

1. When a negative rebase occurs, causing `totalEarnings` during vault-end withdrawal to be much lower than earlier withdrawals.
2. when admin settle debt with a negative earning, ending the vault with a lower `totalEarnings` than earlier.

When these occur, it will cause `calculateVariableWithdrawState()` to revert in `vaultEndedWithdraw()`, preventing the affected variable participant from withdrawing his share of `feeEarnings` (see code below).

```
function vaultEndedWithdraw(uint256 side) internal {
    ...
      if (totalEarnings > 0) {
        //@audit this could reverts due to edge cases where participant share of
        // totalEarnings < previous withdrawn earnings
            that will prevent the affected participants from withdrawing
        // fee earnings, which is executed below
        (
          uint256currentState,
          uint256stakingEarningsShare
        ) = calculateVariableWithdrawState(
          totalEarnings,
          variableToWithdrawnStakingEarnings[msg.sender]
        );
        stakingShareAmount = stakingEarningsShare;
        variableToWithdrawnStakingEarnings[msg.sender] = currentState;
      }

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

      if (totalFees > 0) {
        (
          uint256currentState,
          uint256feesShare
        ) = calculateVariableWithdrawState(
          totalFees,
          variableToWithdrawnFees[msg.sender]
        );
        feeShareAmount = feesShare;
        variableToWithdrawnFees[msg.sender] = currentState;
      }

       vaultEndedStakingEarnings -= stakingShareAmount;
      feeEarnings -= feeShareAmount;

      variableBearerToken.burn(msg.sender, bearerBalance);

      uint256 sendAmount = stakingShareAmount + feeShareAmount;
      sendFunds(sendAmount);

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

Recommendations

Set `ethAmountOwed` to zero when `previousWithdrawnAmount >= bearerBalance.mulDiv(totalEarnings, variableBearerToken.totalSupply())`.

[H-07] Admin could incur a loss with debt settlement

Severity

Impact: High, admin will incur a loss as he will pay more than required to settle debts

Likelihood: Medium, occurs when admin settles debt

Description

An admin of LidoVault can perform a debt settlement to end the vault early due to unexpected issues with Lido. The process is initiated using `initiatingAdminSettleDebt()` with the `adminSettleDebtAmount` paid by admin in ETH. After the timelock, the debt is settled upon calling `adminSettleDebt()`, which will transfer the stETH balance to admin, in exchange for what was paid with `adminSettleDebtAmount`.

Basically with the above exchange, the correct `adminSettleDebtAmount` paid should be equivalent to the unwithdrawn fixed deposit and staking/fee earnings in the `LidoAdapter`.

However, the issue is that after payment of `adminSettleDebtAmount`, the participants could still withdraw the fixed deposits and staking/fee earnings, reducing the stETH balance in `LidoAdapter`. When that occur, admin will actually receive a lower amount of stETH than what was expected during `initiatingAdminSettleDebt()`.

Furthermore, the calculation of `vaultEndedStakingEarnings` and `vaultEndedFixedDepositsFunds` will be incorrect as they are based on the incorrect `adminSettleDebtAmount`.

```
function adminSettleDebt(address adminLidoAdapterAddress) external onlyAdmin {
    require(block.timestamp > adminSettleDebtLockPeriodEnd, "ANM");

    vaultEndedWithdrawalsFinalized = true;
    adminSettledDebt = true;

    //@audit adminSettleDebtAmount is set during initiatingAdminSettleDebt(),
         which could be higher than required when withdrawals are made
    // during the timelock period
    vaultEndedStakingEarnings = fixedETHDepositToken.totalSupply
      () < adminSettleDebtAmount
      ? adminSettleDebtAmount - fixedETHDepositToken.totalSupply()
      : 0;

         vaultEndedFixedDepositsFunds = adminSettleDebtAmount - vaultEndedStakingEarni

    //@audit the stETH transferred to admin will be lesser than
    // adminSettleDebtAmount due to withdrawals
    if (vaultEndedWithdrawalRequestIds.length > 0) {
      IAdminLidoAdapter adminLidoAdapter = IAdminLidoAdapter
        (adminLidoAdapterAddress);
      adminLidoAdapter.setLidoWithdrawalRequestIds
        (vaultEndedWithdrawalRequestIds);
      for (uint i = 0; i < vaultEndedWithdrawalRequestIds.length; i++) {
        lidoAdapter.transferWithdrawalERC721
          (adminLidoAdapterAddress, vaultEndedWithdrawalRequestIds[i]);
      }
    } else {
      lidoAdapter.transferStETH
        (adminLidoAdapterAddress, lidoAdapter.stakingBalance());
    }

    emit AdminSettledDebt(msg.sender);
  }
```

Recommendations

In `adminSettleDebt()`, leave an amount of ETH equivalent to LidoAdapter's sETH balance in the vault and refund the balance amount of `adminSettleDebtAmount` to `AdminLidoAdapter`.

[H-08] Protocol will miss part of it's fee

Severity

Impact: Medium, part of the protocol fee would missed

Likelihood: High, each user that will claim his earnings in ongoing vault more than ones would pay less fees

Description

Vault allows variable stakers to claim earnings during the ongoing stage using the `withdraw` function:

```
File: LidoVault.sol
458:   function withdraw(uint256 side) external nonReentrant {
...
516:       // Vault started and in progress
517:     } else if (!isEnded()) {
518:       if (side == FIXED) {
...
559:       } else {
560:         require
  (variableToVaultOngoingWithdrawalRequestIds[msg.sender].length == 0, "WAR");
561:
562:         if (msg.sender == protocolFeeReceiver && appliedProtocolFee > 0) {
563:           return protocolFeeReceiverWithdraw();
564:         }
565:
566:         uint256 lidoStETHBalance = lidoAdapter.stakingBalance();
567:         uint256 fixedETHDeposits = fixedETHDepositToken.totalSupply();
568:
569:         // staking earnings have accumulated on Lido
570:         if (lidoStETHBalance > fixedETHDeposits) {
571:           (
  uint256currentState,
  uint256ethAmountOwed
) = calculateVariableWithdrawState(
572:             (
  lidoStETHBalance-fixedETHDeposits
) + withdrawnStakingEarnings + totalProtocolFee,
573:             variableToWithdrawnStakingEarnings[msg.sender]
574:           );
575:
576:           if (ethAmountOwed >= lidoAdapter.minStETHWithdrawalAmount()) {
577:             // estimate protocol fee and update total - will actually be
// applied on withdraw finalization
578:             uint256 protocolFee = ethAmountOwed.mulDiv
  (protocolFeeBps, 10000);
579:             totalProtocolFee += protocolFee;
580:
581:             withdrawnStakingEarnings += ethAmountOwed - protocolFee;
582:
              variableToWithdrawnStakingEarnings[msg.sender] = currentState - protoco
583:

585:               msg.sender,
586:               ethAmountOwed
587:             );
588:
589:             emit LidoWithdrawalRequested(
590:               msg.sender,
591:               variableToVaultOngoingWithdrawalRequestIds[msg.sender],
592:               VARIABLE,
593:               isStarted(),
594:               isEnded()
595:             );
596:             return;
597:           }
598:         }
...
```

The amount for claiming is calculated at L571 based on the current staking earnings accumulated on Lido and the already withdrawn amount for the caller

using the `calculateVariableWithdrawState` helper function:

```
File: LidoVault.sol
907:   function calculateVariableWithdrawState(
908:     uint256 totalEarnings,
909:     uint256 previousWithdrawnAmount
910:   ) internal view returns (uint256, uint256) {
911:     uint256 bearerBalance = variableBearerToken.balanceOf(msg.sender);
912:     require(bearerBalance > 0, "NBT");
913:
914:     uint256 ethAmountOwed = bearerBalance.mulDiv
  (totalEarnings, variableBearerToken.totalSupply()) -
915:       previousWithdrawnAmount;
916:
917:     return (ethAmountOwed + previousWithdrawnAmount, ethAmountOwed);
918:   }
```

However, at L582 we can see that the protocol fee is substructed from the withdrawn amount that is saved in the `variableToWithdrawnStakingEarnings`. On the next call after the first earning withdrawal, this value would be used inside `calculateVariableWithdrawState` at L915, resulting in `ethAmountOwed` bigger than it should be, it would include the protocol fee amount from the previous earning withdrawal, since `previousWithdrawnAmount` would correspond to user's only withdrawn amount while protocol fee from previous withdraw call would be encountered as user's earnings.

This would result in receiving bigger amounts for users who would withdraw their staking earnings more than once on an ongoing vault.

Recommendations

At L573 use `variableToWithdrawnStakingEarnings[msg.sender].mulDiv(10000, 10000 - protocolFeeBps)`, this way `calculateVariableWithdrawState` function would calculate the correct earnings for the user since the protocol fee would be included in the `previousWithdrawnAmount` parameter value. Also, L582 should be updated to store the correct amount of the user's withdrawn earnings:

```

```

[H-09] Funds could stuck in `lidoAdapter` if the user requests withdrawal before

`adminSettleDebt()` execution and finalized after

Severity

Impact: High, users could lose their deposits

Likelihood: Medium, scenario requires admin settle debt execution and specific order of transactions

Description

When the admin calls `initiatingAdminSettleDebt` users have a timelock period to withdraw their deposits. They will use the `withdraw` function for this. Depending on what stage the vault is, the contact would burn the appropriate user's tokens balance, request a withdrawal from Lido using `lidoAdapter`, and save request IDs in the associate array slot:

```
File: LidoVault.sol
458:   function withdraw(uint256 side) external nonReentrant {
...
461:     // Vault has not started
462:     if (!isStarted()) {
463:       if (side == FIXED) {
...

490:             msg.sender,
491:             claimBalance
492:           );
...
516:       // Vault started and in progress
517:     } else if (!isEnded()) {
518:       if (side == FIXED) {
...

546:           requestIds: lidoAdapter.requestWithdrawViaETH
  (msg.sender, withdrawAmount),
547:           timestamp: block.timestamp
548:         });
549:         fixedOngoingWithdrawalUsers.push(msg.sender);
...
558:         return;
559:       } else {
...

585:               msg.sender,
586:               ethAmountOwed
587:             );
...
```

After admin calls `adminSettleDebt` all `stEth` balances or `vaultEndedWithdrawalRequestIds` would be transferred from the `lidoAdapter` to the `adminLidoAdapterAddress`:

```
File: LidoVault.sol
712:   function adminSettleDebt
  (address adminLidoAdapterAddress) external onlyAdmin {
...
723:     if (vaultEndedWithdrawalRequestIds.length > 0) {
724:       IAdminLidoAdapter adminLidoAdapter = IAdminLidoAdapter
  (adminLidoAdapterAddress);
725:       adminLidoAdapter.setLidoWithdrawalRequestIds
  (vaultEndedWithdrawalRequestIds);
726:       for (uint i = 0; i < vaultEndedWithdrawalRequestIds.length; i++) {
727:         lidoAdapter.transferWithdrawalERC721
  (adminLidoAdapterAddress, vaultEndedWithdrawalRequestIds[i]);
728:       }
729:     } else {
730:       lidoAdapter.transferStETH
  (adminLidoAdapterAddress, lidoAdapter.stakingBalance());
731:     }
...
```

At the same time since `adminSettledDebt` is now equal to `true`, all calls to the `sendFunds` function would be executed using vault balance, but not `lidoAdapter` as previously:

```
File: LidoVault.sol
845:   function sendFunds(uint256 ethSendAmount) internal {
846:     if (adminSettledDebt) {
847:       (bool sent, ) = msg.sender.call{value: ethSendAmount}("");
848:       require(sent, "ETF");
849:     } else {
850:       lidoAdapter.transferWithdrawnFunds(msg.sender, ethSendAmount);
851:     }
852:   }
```

This means that withdrawal requests that would be finalized after `adminSettleDebt()` execution would be claimed to the `lidoAdapter` and stay there, while the vault would try to repay the user's withdrawals with its balance.

Consider the next scenario:

1. There is a vault with 10e18 fixed side capacity and 0.5e18 variable side capacity. Alice deposits 4 eth on the fixed side, Bob deposits 6 eth, Charlie provides 0.5 eth on the variable side, and the vault starts.
2. Alice and Bob claim their premiums.
3. Admin decides to call `initiatingAdminSettleDebt` with `msg.value` equal to 10.5 eth.

4. Bob requested withdrawal with the `withdraw` function. Since `adminSettleDebt` is not `true` yet, his request is executed using `lidoAdapter.requestWithdrawViaETH()`, and the request id is saved in the `fixedToVaultOngoingWithdrawalRequestIds` array.
5. Admin calls the `adminSettleDebt` function. Since Bob burned his `fixedETHDepositToken` balance (6 eth) on previous step - now `vaultEndedStakingEarnings` is equal to `10.5e18 - 4e18 = 6.5e18` and would be fully claimed by Charlie as only variable side participant, at the same time `vaultEndedFixedDepositsFunds` is equal to 4e18, equivalent to Alice `fixedETHDepositToken` balance and total supply of it's token.
6. Alice and Charlie withdraw their funds, leaving the vault with zero balance.
7. Bob tries to finalize his withdrawal, however, it fails because of the `OutOfFund` error. Withdrawal request firstly successfully increases the `lideAdapter` balance, but since the `sendFunds` function now sends funds from the vault balance instead of the adapter - the function would revert.

In this scenario, Admin loses a portion of funds in the same way described in [H-07] by overpaying Charlie, but at the same time - Bob's balance is now completely stuck.

Recommendations

`LidoAdapter#_claimWithdrawals` function should send all withdrawn eth to the `msg.sender` (vault in this case). This way all previously requested withdrawals would guaranteed to end on vault balance.

8.3. Medium Findings

[M-01] Deposit function can be DOSed

Severity

Impact: Medium, while core contract functionality would be DOSed, it's still possible to bypass attack using a private relayer

Likelihood: Medium, the attack vector is cheap, but at the same time malicious actor has no incentive except griefing users

Description

The deposit function requires that the sum of fixed and variable deposits be strictly equal to the `fixedSideCapacity` and `variableSideCapacity` correspondingly. This is done by checking the supply of deposit tokens and current `msg.value` (L395 and L409). But if the `msg.value` is bigger than the difference between the total supply of deposit tokens and side capacity - the whole deposit tx would be reverted.

This opens a griefing attack vector when malicious actors could DOS deposit function. Since the vault is permissionless - an attacker could spot users' tx that would start the vault and front-run it with a dust amount deposit, causing reverting of the first one. This could be repeated many times, effectively preventing the vault from being started.

```
File: LidoVault.sol
383:   function deposit
  (uint256 side) external payable isInitialized nonReentrant {
384:     require(!isStarted(), "DAS");
385:     // don't allow deposits once settle debt process has been initialized
// to prevent vault from starting
386:     require(!isAdminSettleDebtInitialized(), "AAI");
387:     require(side == FIXED || side == VARIABLE, "IS");
388:     require(msg.value > 0, "NZV");
389:
390:     uint256 amount = msg.value;
391:     if (side == FIXED) {
392:       // Fixed side deposits
393:
394:       // no refunds allowed
395:       require
  (amount <= fixedSideCapacity - fixedETHDepositToken.totalSupply(), "OED");
396:
397:       // Stake on Lido
398:       uint256 shares = lidoAdapter.stakeFunds{value: amount}(msg.sender);
399:
400:       // Mint claim tokens
401:       fixedClaimToken.mint(msg.sender, shares);
402:       fixedETHDepositToken.mint(msg.sender, amount);
403:
404:       emit FixedFundsDeposited(amount, shares, msg.sender);
405:     } else {
406:       // Variable side deposits
407:
408:       // no refunds allowed
409:       require
  (amount <= variableSideCapacity - variableBearerToken.totalSupply(), "OED");
410:
411:       // Mint bearer tokens
412:       variableBearerToken.mint(msg.sender, amount);
413:
414:       emit VariableFundsDeposited(amount, msg.sender);
415:     }
```

Recommendations

Deposit function should refund user in case if `msg.value` is greater than the current vault capacity.

[M-02] Race condition with `initiatingAdminSettleDebt()` could under-compensate participants

Severity

Impact: Medium, participants will incur loss on withdrawal as admin underpaid for debt settlement. But admin (trusted) can make up the loss by

refunding them the difference separately.

Likelihood: Medium, occurs when admin settles debt

Description

Within `LidoVault.deposit()`, there is a check `require(!isAdminSettleDebtInitialized(), "AAI")` that prevents fixed/variable participants from starting the vault with the last deposit when the admin settle debt process has been initialized.

```
function deposit(uint256 side) external payable isInitialized nonReentrant {
    require(!isStarted(), "DAS");

    // don't allow deposits once settle debt process has been initialized to
    // prevent vault from starting
    require(!isAdminSettleDebtInitialized(), "AAI");
```

However, the check can be bypassed due to a race condition, where an unexpected vault-starting `deposit()` occurs before `initiatingAdminSettleDebt()`, forcing the vault to start.

The race condition could occur as follows,

1. Variable participants has deposited into LidoVault, filling up the premiums.
2. Admin decides to cancel the vault and settle the debt with zero earnings by calling `initiatingAdminSettleDebt()` with `adminSettleDebtAmount [msg.value] = fixedETHDepositToken.totalSupply()`. Note that `adminSettleDebtAmount` will be less than `fixedSideCapacity`, since vault is not started yet.
3. A fixed participant happen to perform the last `deposit()` call, which is somehow executed before `initiatingAdminSettleDebt()` due to MEV/re-ordering.
4. The vault is unintentionally started, causing `fixedETHDepositToken.totalSupply() == fixedSideCapacity`. However as `adminSettleDebtAmount < fixedSideCapacity` due to step 3, the admin will cause the participants to be under-compensated as the ETH in the vault does not equate to LidoAdapter stETH balance.
5. When the vault ended after timelock with zero earnings, the variable participants will suffer a loss as there are no earned staked earnings to withdraw.

6. There is also a possibility that fixed participants can only claim a portion of their initial deposit even after claiming premium, if `vaultEndedFixedDepositsFund` is significantly lower than expected.

Recommendations

Add an `expectedVaultStarted` (bool) as a parameter for `initiatingAdminSettleDebt()` with a check `require(expectedVaultStarted == isStarted());` to ensure the vault start status is what the admin expected.

If the check fails, `initiatingAdminSettleDebt()` should revert, which will then allow admin to repeat `initiatingAdminSettleDebt()` with a correct `msg.value` that compensates the participants.

[M-03] `adminSettleDebtLockPeriod` might not be long enough

Severity

Impact: High as users might not be able to withdraw in time before their `stETH` is transferred to `AdminLidoAdapter`

Likelihood: Low, as it needs to be a delay in the queue for lido withdrawals and an event that causes admin to trigger settle debt

Description

`admin` has the ability to end the vault early in case of unforeseen circumstances. This is done by first calling `initiatingAdminSettleDebt` which triggers a timelock before `adminSettleDebt` can be called. This so that users can chose to withdraw before all `stETH` and pending withdraw requests are transferred to `AdminLidoAdapter`.

The timelock, `adminSettleDebtLockPeriod`, is set in `VaultFactory` to `3 days`.

This might however not be enough. Looking at what lido says the withdrawal requests can take anything between 1-5 days: https://blog.lido.fi/ethereum-withdrawals-overview-faq/#:~:text=How%20does%20the,1%2D5%20days.

How does the withdrawal process work? The withdrawal process is simple and has two steps:

Request: Lock your stETH/wstETH by issuing a withdrawal request. ETH is sourced to fulfill the request, and then locked stETH is > burned, which marks the withdrawal request as claimable. **Under normal circumstances, this can take anywhere between 1-5 days.**

Recommendations

Consider increasing the timelock to 5 days, or 6 days to give stakers a day to react as well. As well as addressing [H-09].

[M-04] `adminSettleDebt()` could be called multiple times leading to the loss of funds

Severity

Impact: High, since some users could lose their funds by others users claiming these funds

Likelihood: Low, since multiple call of `adminSettleDebt()` could be done only accidently or by malicious admin

Description

`adminSettleDebt` lack of check if it was already called:

```
File: LidoVault.sol
712:   function adminSettleDebt
  (address adminLidoAdapterAddress) external onlyAdmin {
713:     require(block.timestamp > adminSettleDebtLockPeriodEnd, "ANM");
714:
715:     vaultEndedWithdrawalsFinalized = true;
716:     adminSettledDebt = true;
717:
718:     vaultEndedStakingEarnings = fixedETHDepositToken.totalSupply
  () < adminSettleDebtAmount
719:       ? adminSettleDebtAmount - fixedETHDepositToken.totalSupply()
720:       : 0;
721:
       vaultEndedFixedDepositsFunds = adminSettleDebtAmount - vaultEndedStakingEarning
722:
723:     if (vaultEndedWithdrawalRequestIds.length > 0) {
724:       IAdminLidoAdapter adminLidoAdapter = IAdminLidoAdapter
  (adminLidoAdapterAddress);
725:       adminLidoAdapter.setLidoWithdrawalRequestIds
  (vaultEndedWithdrawalRequestIds);
726:       for (uint i = 0; i < vaultEndedWithdrawalRequestIds.length; i++) {
727:         lidoAdapter.transferWithdrawalERC721
  (adminLidoAdapterAddress, vaultEndedWithdrawalRequestIds[i]);
728:       }
729:     } else {
730:       lidoAdapter.transferStETH
  (adminLidoAdapterAddress, lidoAdapter.stakingBalance());
731:     }
...
```

This could lead to a scenario when:

1. Admin calls `adminSettleDebt()` and sets correct values for `vaultEndedStakingEarnings` and `vaultEndedFixedDepositsFunds` variables.
2. Users withdraw their positions and change `fixedETHDepositToken.totalSupply()` this way.
3. Admin calls `adminSettleDebt()` again, now variables from step 1 would end with totally wrong values, leading to loss of funds for users that haven't withdrawn their positions yet.

Recommendations

Add a check that would prevent multiple calls to the `adminSettleDebt()` function.

[M-05] Admin could call `adminSettleDebt` anytime without previous call `initiatingAdminSettleDebt`

Severity

Impact: High, core functionality related to timelock would be bypassed

Likelihood: Low, since could be done by admin only accidentally or in case of malicious action

Description

The `adminSettleDebt` function is designed to allow the admin to settle debt after finishing timelock. The last one should be initiated during `initiatingAdminSettleDebt`. While `adminSettleDebt` successfully checks if timelock is ended, it fails to check if timelock was ever initiated, this way check at L713 would compare the current timestamp with the default value for `uint256` type - `0`, meaning `adminSettleDebt` could be executed by admin anytime if timelock wasn't yet initiated. Also, `vaultEndedStakingEarnings` and `vaultEndedFixedDepositsFunds` would be set to 0 values, breaking withdrawal flow for users.

```
File: LidoVault.sol
712:   function adminSettleDebt
  (address adminLidoAdapterAddress) external onlyAdmin {
713:     require(block.timestamp > adminSettleDebtLockPeriodEnd, "ANM");
714:
715:     vaultEndedWithdrawalsFinalized = true;
716:     adminSettledDebt = true;
717:
718:     vaultEndedStakingEarnings = fixedETHDepositToken.totalSupply
  () < adminSettleDebtAmount
719:       ? adminSettleDebtAmount - fixedETHDepositToken.totalSupply()
720:       : 0;
721:
      vaultEndedFixedDepositsFunds = adminSettleDebtAmount - vaultEndedStakingEarning
...
```

Recommendations

Add condition to L713 require statement that `adminSettleDebtLockPeriodEnd > 0`.

[M-06] Accrual before vault start can be used as honeypot

Severity

Impact: : low, as you can argue that the victim should know better

Likelihood: : high, as it is very easy to perform at low risk for the attacker

Description

As soon as a fixed side staker deposits, their funds are sent to lido and start accruing rewards:

```
397:       // Stake on Lido
398:       uint256 shares = lidoAdapter.stakeFunds{value: amount}(msg.sender);
```

Anyone on the fixed side can chose to unstake before the vault starts and receive their share *and* the accrued rewards:

```
490:             msg.sender,
491:             claimBalance
492:           );
```

```
627:     // give fixed depositor all of their principal + any staking earnings
628:     uint256 sendAmount = lidoAdapter.claimWithdrawals
  (msg.sender, requestIds);
629:
630:     sendFunds(sendAmount);
```

Hence, joining the vault before it beings is the same as just staking in lido directly.

Thus, an attacker can create a vault with a high premium, deposit the full fixed premium side which will slowly accrue.

At one point a victim sees that there is a vault with large earnings already built up and choses to join. Even though the premium is high, there are already earnings covering the high premium so they join on the variable side.

The attacker then front runs the variable deposit and withdraws their deposit and staking earnings. They can then in the same tx rejoin the vault with a fresh deposit, starting over from scratch.

Thus they have lured a victim to join a vault at an unfair premium with little to no risk for the fixed side attacker.

Recommendations

There is no real good on chain mitigation for this so our suggestion is to make it clear in the documentation that earnings before the vault start is not guaranteed.

[M-07] ongoing withdrawals from variable side hurts variable earnings

Severity

Impact: Medium, as it can be argued that his is part of the risk on the variable side

Likelihood: Medium, as it also hurts the one withdrawing

Description

When the vault is active, a variable side staker can chose to withdraw their share of the, so far, accrued earnings:

```
571:           (
  uint256currentState,
  uint256ethAmountOwed
) = calculateVariableWithdrawState(
572:             (
  lidoStETHBalance-fixedETHDeposits
) + withdrawnStakingEarnings + totalProtocolFee,
573:               variableToWithdrawnStakingEarnings[msg.sender]
574:           );
575:
576:           if (ethAmountOwed >= lidoAdapter.minStETHWithdrawalAmount()) {
...

585:               msg.sender,
586:               ethAmountOwed
587:             );
```

On L584 this unstakes from lido, meaning it will lessen the rewards gained by other variable stakers (and themselves) going forward. This mainly makes it difficult for variable side stakers to calculate expected earnings as accounting for how many early withdrawals might be made is hard.

Recommendations

There are a couple ways of mitigating this, either an early withdrawal fee on the variable side, similar to early exit fee on the fixed side. Another way is to

have the vault creator chose if its allowed or not by adding a flag, ongoing variable withdrawals true/false.

8.4. Low Findings

[L-01] Unnecessary division before multiplication

When fixed claimers decide their premium they take that as a share of the variable side:

```
438:     // Send a proportional share of the total variable side deposits
//(premium) to the fixed side depositor
439:     uint256 sendAmount = claimBal.mulDiv(1e18, fixedLidoSharesTotalSupply
  ()).mulDiv(variableSideCapacity, 1e18);
```

This calculation does division before multiplication which results in a small loss of precision (the smaller the balance of the claimee, the greater the loss of precision). The calculation could be simplified to:

```
claimBal.mulDiv(variableSideCapacity, fixedLidoSharesTotalSupply());
```

to avoid this.

However see [C-02] about the issues with `fixedLidoSharesTotalSupply()`

[L-02] early exit can lock funds in contract

Description

When a fixed staker exists early in the staking duration they are penalized with an `earlyExitFee`.

The `earlyExitFee` is decided by this calculation in `calculateFixedEarlyExitFees`:

```
// @audit-info max value 1e18
945:     uint256 remainingProportion =
  (endTime > timestampRequested ? endTime - timestampRequested : 0).mulDiv(
946:       1e18,
947:       duration
948:     );
949:
950:     // Calculate the scaling fee based on the quadratic scaling factor and
// earlyExitFeeBps
        // @audit-info max value upfrontPremium * 0.1 (1_000/10_000)
951:     uint256 earlyExitFees = upfrontPremium.mulDiv(
952:       earlyExitFeeBps.mulDiv(remainingProportion.mulDiv
  (remainingProportion, 1e18), 1e18),
953:       10000
954:     );
955:
956:     // Calculate the amount to be paid back of their original upfront
// claimed premium, not influenced by quadratic scaling
        // @audit-info max value upfrontPremium * 0.1 + upfrontPremium = 1.1 *
        // upfrontPremium
957:     earlyExitFees += upfrontPremium - upfrontPremium.mulDiv
  (timestampRequested - startTime, duration);
```

`earlyExitFeeBps` is 10% (set at construction from `VaultFactory`).

The fee is then applied in `claimFixedVaultOngoingWithdrawal`:

```
880:     uint256 amountWithdrawn = lidoAdapter.claimWithdrawals
  (msg.sender, requestIds);
881:
         // @audit-info max value 1.1 * upfrontPremium
882:     uint256 earlyExitFees = calculateFixedEarlyExitFees
  (upfrontPremium, request.timestamp);
...
889:     return amountWithdrawn - earlyExitFees;
```

The issue is the latest subtraction `amountWithdrawn - earlyExitFees`. If the vault has a large enough variable to fixed capacity ratio this can underflow if a staker withdraws early enough. Thus making this withdrawal impossible. As the entry in `fixedToVaultOngoingWithdrawalRequestIds` will never be removed this will lock these funds in the contract.

Imagine a vault where the ration is 1-1 fixed vs variable. Then the `upfrontPremium` would be equal to the `amountWithdrawn`. Then unstaking early enough would cause `amountWithdrawn - earlyExitFees` to underflow.

Recommendations

Consider using `min(earlyExitFee,amountWithdrawn)` as `earlyExitFee`, i.e saying that `earlyExitFee` can max be what you are withdrawing.

[L-03] Missing check for revoked adapter in `createVault()`

`revokeAdapterType()` will set `adapterTypeByteCode = ""` with the intention that new vault cannot be deployed with revoked adapters. However, in `createVault()` there is no check for revoked adapters and it's possible to deploy a vault with revoked adapters.

To fix this, add a check `require(adapterTypeByteCode[_adapterInfo.adapterTypeId] != "");` in `createVault()` L174.