SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Saffron Finance
Prepared by:	Sherlock
Lead Security Expert:	0x73696d616f
Dates Audited:	September 16 - September 21, 2024
Prepared on:	October 9, 2024

Introduction

Saffron is a DeFi primitive that turns a single source of yield into variable and fixed interest. The Saffron LIDO Vault makes native ETH staking into a fixed interest instrument.

Scope

Repository: saffron-finance/lido-fiv

Branch: main

Audited Commit: 7246b6651c8affffe17faa4d2984975102a65d81

Final Commit: f7f81583e8d0f251423c13dc2146e3fa8de5aefb

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
3	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x73696d616f
dobrevaleri
iamnmt
tobi0x18
0xAlix2

Issue H-1: `totalEarnings` is incorrect when withdrawing after ending which will withdraw too many funds leaving the `Vault` insolvent

Source: https://github.com/sherlock-audit/2024-08-saffron-finance-judging/issues/85

Found by

0x73696d616f

Summary

`totalEarnings` in LidoVault::vaultEndedWithdraw() is calculated as:

```
uint256 totalEarnings = vaultEndingETHBalance.mulDiv(
    withdrawnStakingEarningsInStakes, vaultEndingStakesAmount
) - totalProtocolFee + vaultEndedStakingEarnings;
```

The withdrawn shares are scaled to get the total earnings, along with vaultEndedStakingEarnings, which was aquired by getting the liquidity from the remaining shares when `LidoVault::finalizeVaultEndedWithdrawals()` was called.

However, `totalProtocolFee` is not scaled, which means that as the steth eth/shares ratio increases, the protocol fee increases with it, otherwise it will overestimate the `totalEarnings`, as can be confirmed in the calculations in the `POC`.

Root Cause

In `LidoVault.sol:775`, `protocolFee` is not scaled to the current steth eth/shares. It should be in shares and multiplied by the current exchange rate. In `LidoVault.sol:533`, `protocolFee` should be tracked as shares.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. Users withdraw via `LidoVault::vaultEndedWithdraw()` and withdraw more than they should due to the total earnings. Next users will not have enough funds to make their withdrawals, taking a loss for the profit of the earlier users.

Impact

The protocol becomes insolvent.

PoC

Consider that 2 users have 50% of the variable bearer token each and the protocol starts with 100 ETH and 100 Shares. After some time, the protocol accrues stEth and the holdings become 200 ETH and 100 shares. Before the vault ends, user A withdraws his share of the rewards via `LidoVault::withdraw()`, which is:

```
totalEarnings =  200 * (100 + 0) / 100 - 100 = 100
ethAmountOwed = totalEarnings / 2 = 100 / 2 = 50 // 50% of the bearer tokens, so
↪  divides by 2
protocolFee = 50 * 0.05 = 2.5
stakesAmountOwed = 50 / 2 = 25 // each share of stETH is worth 2 ETH, the same
↪  ratio as the protocol having 200 ETH and 100 shares.

// RESULT

totalProtocolFee = 2.5
variableToWithdrawnStakingEarningsInShares[userA] = 25
withdrawnStakingEarningsInStakes = 25
ETH = 150
shares = 75
```

Now, assume that the stETH shares double again its value, there are 300 ETH and 75 shares now in the protocol. Consider that the vault ended so the fixed users claimed their 100 ETH, which leaves the protocol with 200 ETH and 50 shares (25 shares were removed to pay the fixed users).

And lastly, userB withdraws his variable rewards by calling `LidoVault::vaultEndedWithdraw()`. The vault has earnings that were not withdrawn, so in the beginning of `LidoVault::vaultEndedWithdraw()` it requests the entire balance of the contract and registers the following:

```
vaultEndingStakesAmount = 50
vaultEndingETHBalance = 200
```

Then, `LidoVault::finalizeVaultEndedWithdrawals()` is called, which claims the withdrawals, getting 200 ETH and setting the following variables:

```
amountWithdrawn = 200 ETH
fixedETHDeposit = 0 // was withdrawn by the fixed users already
vaultEndedStakingEarnings = 200 - 200 * 0.05 = 200 - 10 = 190
```

And finally, it calls again `LidoVault::vaultEndedWithdraw()` at the end, which leads to the following `totalEarnings` calculations:

```
totalEarnings = 200 * 25 / 50 - 2.5 + 190 = 287.5
ethAmountOwed = totalEarnings / 2 = 287.5 / 2 = 143.75
```

At this point, there is 190 ETH in the contract, and userA has withdrawn 25 shares, but userB has not withdrawn any shares. In the first time the stETH shares price doubled, they both had the same shares, so they should get the same amount. However, by the end, the stETH shares price doubled again, but one user had already withdrawn. Intuitively, this means that userB, who has not withdrawn, should get 75% of the final earnings while user A should get 25%, but this is not what happens. `variableToWithdrawnStakingEarningsInShares[userB] == 0`, so userB will withdraw `143.75 / 190 ~= 0.757`.

To fix this, if we take the `totalProtocolFee` in shares instead of flat, when userA withdrew the first time, it withdraw 2.5 in fees, which at the time was 1.25 shares (it doubled). This yields:

```
totalEarnings = 200 * 25 / 50 - 1.25 * 200 / 50 + 190 = 285
ethAmountOwed = totalEarnings / 2 = 285 / 2 = 142.5
```

And lastly, `142.5 / 190 == 0.75`, which is the correct amount.

Additionally, if we calculate userA's amount, it will be wrong too and shows us how it does not add up. We just need to subtract the shares of userA worth in ETH to the `totalEarnings / 2` to get his part:

```
ethAmountOwed = 143.75 - 25 * 200 / 50 = 43.75
```

So summing up, they get `143.5 + 43.75 == 187.25`, which is less than 190 and some ETH is stuck. The amount is lower than 190, but it should be higher, the issue is that there is another bug, which is, the component that userA has already withdrawn should be discounted by the shares paid in fees (1.25). If we do this, it becomes

```
ethAmountOwed = 143.75 - (25 - 1.25) * 200 / 50 = 48.75
```

Now, summing both users' withdrawals, `143.75 + 48.75 == 192.5`, which is bigger than 190 and they will withdraw too much.

Mitigation

`totalProtocolFee` must be tracked as shares in `LidoVault.sol:533`.

```
totalProtocolFee += lido.getSharesByPooledEth(protocolFee);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/saffron-finance/lido-fiv/pull/29

Issue H-2: The incorrect accounting of protocol fee will cause double charging fee and wrong distribution of earnings for variable users

Source: https://github.com/sherlock-audit/2024-08-saffron-finance-judging/issues/129

Found by

0x73696d616f, dobrevaleri, iamnmt

Summary

The incorrect accounting of protocol fee will cause double charging fee and wrong distribution of earnings for variable users.

Root Cause

The calculation for a variable user's earnings, when they withdraw where `isStarted()` and `!isEnded()`

https://github.com/sherlock-audit/2024-08-saffron-finance/blob/38dd9c8436db341c331f1b14545770c1766fc0ee/lido-fiv/contracts/LidoVault.sol#L520-L545

```
          uint256 lidoStETHBalance = stakingBalance();
          uint256 fixedETHDeposits = fixedSidestETHOnStartCapacity;

        // staking earnings have accumulated on Lido
        if (lidoStETHBalance > fixedETHDeposits + minStETHWithdrawalAmount()) {
          uint256 currentStakes = stakingShares();
1>        (uint256 currentState, uint256 ethAmountOwed) =
↪   calculateVariableWithdrawState(
            (lidoStETHBalance.mulDiv(currentStakes +
↪   withdrawnStakingEarningsInStakes, currentStakes) - fixedETHDeposits),
            variableToWithdrawnStakingEarningsInShares[msg.sender].mulDiv(lidoSt
↪   ETHBalance, currentStakes)
          );
          if (ethAmountOwed >= minStETHWithdrawalAmount()) {
            // estimate protocol fee and update total - will actually be applied
↪   on withdraw finalization
2>          uint256 protocolFee = ethAmountOwed.mulDiv(protocolFeeBps, 10000);
            totalProtocolFee += protocolFee;
3>          uint256 stakesAmountOwed = lido.getSharesByPooledEth(ethAmountOwed);

            withdrawnStakingEarnings += ethAmountOwed - protocolFee;
```

```
4>          withdrawnStakingEarningsInStakes += stakesAmountOwed;

            variableToWithdrawnStakingEarnings[msg.sender] += ethAmountOwed -
↪ protocolFee;
5>          variableToWithdrawnStakingEarningsInShares[msg.sender] +=
↪ stakesAmountOwed;
            variableToWithdrawnProtocolFee[msg.sender] += protocolFee;
            variableToVaultOngoingWithdrawalRequestIds[msg.sender] =
↪ requestWithdrawViaETH(
              msg.sender,
              ethAmountOwed
            );
        ...
```

The variable user's earnings is the variable `ethAmountOwed` at `1>`. Note that, the earnings also includes the protocol fee (`2>`). Then `ethAmountOwed` is converted to shares `stakesAmountOwed` at `3>`. Then the shares is added to `withdrawnStakingEarningsInStakes` and `variableToWithdrawnStakingEarningsInShares[msg.sender]`

When the vault ends, the variable user's earnings is calculated at

https://github.com/sherlock-audit/2024-08-saffron-finance/blob/38dd9c8436db341c331f1b14545770c1766fc0ee/lido-fiv/contracts/LidoVault.sol#L773-L785

```
      uint256 stakingShareAmount = 0;

1>    uint256 totalEarnings = vaultEndingETHBalance.mulDiv(withdrawnStakingEarni
↪ ngsInStakes,vaultEndingStakesAmount) - totalProtocolFee  +
↪ vaultEndedStakingEarnings;

      if (totalEarnings > 0) {
2>      (uint256 currentState, uint256 stakingEarningsShare) =
↪ calculateVariableWithdrawState(
          totalEarnings,
3>        variableToWithdrawnStakingEarningsInShares[msg.sender].mulDiv(vaultEnd
↪ ingETHBalance, vaultEndingStakesAmount)
        );
        stakingShareAmount = stakingEarningsShare;
        variableToWithdrawnStakingEarningsInShares[msg.sender] =
↪ currentState.mulDiv(vaultEndingStakesAmount,vaultEndingETHBalance);
        variableToWithdrawnStakingEarnings[msg.sender] = currentState;
      }
```

The variable user's earnings is the variable `stakingEarningsShare` at `2>`. `stakingEarningsShare` is calculated basing on `withdrawnStakingEarningsInStakes`,

`variableToWithdrawnStakingEarningsInShares[msg.sender]` (1>, 3>).

We believe by including protocol fee shares in `withdrawnStakingEarningsInStakes`, `variableToWithdrawnStakingEarningsInShares[msg.sender]` will cause `stakingEarningsShare` to be wrongly calculated.

Refer to the attack path and the PoC for a concrete example.

Internal pre-conditions

A variable user withdraws when `isStarted()` and `!isEnded()`

External pre-conditions

No response

Attack Path

Let's have a vault with:

- `fixedSideCapacity = 100 ether`
- `variableSideCapacity = 10 ether`
- `protocolFeeBps = 2_000 (20%)`
- `stETHRate = 1` (1 shares equals to `1 stETH`)

On the variable side:

- Alice deposits `10 ether`

The expected behavior

1. `stETHRate = 1.1`. Vault's balance (stETH): `110 ether` (increased 10% since the beginning).
    - Alice withdraws `8 ether`. `totalProtocolFee = 2 ether`
    - New vault's balance: `100 ether`
2. Vault ends. `stETHRate = 1.21`. Vault's balance (stETH): `110 ether` (increased 10% since the `1.`).
    - Alice withdraws `8 ether`. Protocol fee: `2 ether`
    - New vault's balance: `100 ether`

This is the expected behavior of the vault.

The actual behavior

1. `stETHRate = 1.1`. Vault's balance (stETH): `110 ether` (increased 10% since the beginning).
    - Alice withdraws `8 ether`. `totalProtocolFee = 2 ether`
    - New vault's balance: `100 ether`
2. Vault ends. `stETHRate = 1.21`. Vault's balance (stETH): `110 ether` (increased 10% since the `1.`).
    - Alice withdraws.

```
uint256 stakingShareAmount = 0;

uint256 totalEarnings = vaultEndingETHBalance.mulDiv(withdrawnStakingEarningsInS
↪  takes,vaultEndingStakesAmount) - totalProtocolFee  +
↪  vaultEndedStakingEarnings;

if (totalEarnings > 0) {
  (uint256 currentState, uint256 stakingEarningsShare) =
↪  calculateVariableWithdrawState(
    totalEarnings,
    variableToWithdrawnStakingEarningsInShares[msg.sender].mulDiv(vaultEndingETH
↪  Balance, vaultEndingStakesAmount)
  );
  stakingShareAmount = stakingEarningsShare;
  variableToWithdrawnStakingEarningsInShares[msg.sender] =
↪  currentState.mulDiv(vaultEndingStakesAmount,vaultEndingETHBalance);
  variableToWithdrawnStakingEarnings[msg.sender] = currentState;
}
```

In this code

$$totalEarnings = 110 \times \frac{10/1.1}{100 - 10/1.1} - 2 + 10 \times 0.8 = 17$$

$$stakingEarningsShare = totalEarnings - \frac{10}{1.1} \times \frac{110}{100 - 10/1.1} = 6$$

She can only withdraw back `6 ether`, which is less than `2 ether` comparing to the expected behavior. Moreover, this `2 ether` is not credited to the `protocolFeeReceiver`, no one can claim it and it will stuck in the contract.

On the variable side:

- Alice deposits `5 ether`
- Bob deposits `5 ether`

The expected behavior

1. `stETHRate = 1.1`. Vault's balance (stETH): `110 ether` (increased 10% since the beginning).
    - Alice withdraws `4 ether`. `totalProtocolFee = 1 ether`
    - New vault's balance: `105 ether`
2. Vault ends. `stETHRate = 1.21`. Vault's balance (stETH): `115.5 ether` (increased 10% since the `1.`).
    - Alice withdraws `4 ether`. Protocol fee: `1 ether`
    - Bob withdraws `10.5 * 0.8 = 8.4 ether`. Protocol fee: `10.5 * 0.2 = 2.1 ether`

This is the expected behavior of the vault.

The actual behavior

1. `stETHRate = 1.1`. Vault's balance (stETH): `110 ether` (increased 10% since the beginning).
    - Alice withdraws `4 ether`. `totalProtocolFee = 1 ether`
    - New vault's balance: `105 ether`
2. Vault ends. `stETHRate = 1.21`. Vault's balance (stETH): `115.5 ether` (increased 10% since the `1.`).
    - Alice withdraws

$$totalEarnings = 115.5 \times \frac{5/1.1}{100 - 5/1.1} - 1 + 15.5 \times 0.8 = 16.9$$

$$stakingEarningsShare = totalEarnings/2 - \frac{5}{1.1} \times \frac{115.5}{100 - 5/1.1} = 2.95$$

    - Bob withdraws

$$stakingEarningsShare = totalEarnings/2 - 0 = 8.45$$

In the current logic comparing to the expected behavior, Alice is charged with `1 ether` more protocol fee. The `protocolFeeReceiver` can not claim this `1 ether` more fee, and it will stuck in the contract. Moreover, `0.05 ether` earnings of Alice is credited to Bob.

Impact

- The variable user, who withdraws when `isStarted()` and `!isEnded()`, will be charged more protocol fee when they withdraw when the vault ends
- The exceed protocol fee will be stuck in the contract
- Wrong distribution of the variable earnings

PoC

Add a setter in `LidoVault.sol` to set `lido` and `lidoWithdrawalQueue` to the mock version for easier debugging

https://github.com/sherlock-audit/2024-08-saffron-finance/blob/38dd9c8436db341c331f1b14545770c1766fc0ee/lido-fiv/contracts/LidoVault.sol#L1003-L1007

```
  /// @notice Lido contract
- ILido public constant lido = ILido(0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84);
+ ILido public lido = ILido(0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84);

  /// @notice Lido withdrawal queue contract
- ILidoWithdrawalQueueERC721 public constant lidoWithdrawalQueue =
    ILidoWithdrawalQueueERC721(0x889edC2eDab5f40e902b864aD4d7AdE8E412F9B1);
+ ILidoWithdrawalQueueERC721 public lidoWithdrawalQueue =
    ILidoWithdrawalQueueERC721(0x889edC2eDab5f40e902b864aD4d7AdE8E412F9B1);

+ function setLidoInfo(address _lido, address _withdrawalQueue) public {
+   lido = ILido(_lido);
+   lidoWithdrawalQueue = ILidoWithdrawalQueueERC721(_withdrawalQueue);
+ }
```

Run command: `forge test --match-path test/PoC.t.sol -vv`

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.18;

import { VaultFactory } from "contracts/VaultFactory.sol";
import { LidoVault } from "contracts/LidoVault.sol";

import { Test, console } from "forge-std/Test.sol";

contract MockLido {
    uint256 public rate;
    mapping(address => uint256) private _sharesOf;

    constructor() {
```

```
        rate = 1e27;
    }

    function submit(address _referral) external payable returns (uint256) {
        uint256 shares = msg.value * 1e27 / rate;
        _sharesOf[msg.sender] += shares;
        return shares;
    }

    function approve(address spender, uint256 amount) external returns (bool) {
        return true;
    }

    function getPooledEthByShares(uint256 _sharesAmount) external view returns
↪   (uint256) {
        return _sharesAmount * rate / 1e27;
    }

    function getSharesByPooledEth(uint256 _ethAmount) external view returns
↪   (uint256) {
        return _ethAmount * 1e27 / rate;
    }

    function balanceOf(address _account) external view returns (uint256) {
        return _sharesOf[_account] * rate / 1e27;
    }

    function sharesOf(address _account) external view returns (uint256) {
        return _sharesOf[_account];
    }

    function setRate(uint256 _rate) external {
        rate = _rate;
    }

    function burnShares(uint256 _amount, address _owner) external {
        uint256 shares = _amount * 1e27 / rate;
        _sharesOf[_owner] -= shares;
    }
}

contract MockLidoWithdrawalQueueERC721 {
    MockLido public lido;

    mapping(uint256 requestId => uint256 amount) public requestIdToAmount;
    uint256 public currentRequestId;
```

```
    constructor(address _lido) {
        lido = MockLido(_lido);
    }
    function claimWithdrawal(uint256 _requestId) external {
        payable(msg.sender).call{value: requestIdToAmount[_requestId]}("");
    }

    function requestWithdrawals(uint256[] calldata _amounts, address _owner)
↪   external returns (uint256[] memory requestIds) {
        lido.burnShares(_amounts[0], _owner);
        requestIdToAmount[currentRequestId] = _amounts[0];

        requestIds = new uint256[](1);
        requestIds[0] = currentRequestId++;
    }
    receive() external payable {
    }
}


contract PoC is Test {
    MockLido lido;
    MockLidoWithdrawalQueueERC721 lidoWithdrawalQueueERC721;

    VaultFactory factory;
    LidoVault vault;

    address fixedDepositor = makeAddr('fixedDepositor');
    address alice = makeAddr('alice');
    address bob = makeAddr('bob');
    address feeReceiver = makeAddr('feeReceiver');

    uint256 protocolFeeBps = 2_000;
    uint256 fixedCap = 100 ether;
    uint256 variableCap = 10 ether;

    function setUp() public {
        lido = new MockLido();
        lidoWithdrawalQueueERC721 = new
↪   MockLidoWithdrawalQueueERC721(address(lido));

        factory = new VaultFactory(protocolFeeBps, 0);
        factory.setProtocolFeeReceiver(feeReceiver);

        factory.createVault(fixedCap, 1 days, variableCap);
        (,address addr) = factory.vaultInfo(1);
        vault = LidoVault(payable(addr));
```

```
        vault.setLidoInfo(address(lido), address(lidoWithdrawalQueueERC721));

        vm.deal(address(lidoWithdrawalQueueERC721), 1000 ether);

        vm.deal(fixedDepositor, fixedCap);

        vm.prank(fixedDepositor);
        vault.deposit{value: fixedCap}(0);
    }

    function testFirstVulnerabilityPath() public {
        vm.deal(alice, variableCap);

        vm.prank(alice);
        vault.deposit{value: alice.balance}(1);

        lido.setRate(1.1e27);

        vm.startPrank(alice);
        vault.withdraw(1);
        vault.finalizeVaultOngoingVariableWithdrawals();
        vm.stopPrank();

        console.log("Alice's balance after first claim: %e", alice.balance);

        skip(1 days + 1);

        lido.setRate(1.21e27);

        vm.startPrank(alice);
        vault.withdraw(1);
        vault.finalizeVaultEndedWithdrawals(1);
        vm.stopPrank();

        console.log("Alice's balance at the end: %e", alice.balance);

        vm.prank(feeReceiver);
        vault.withdraw(1);

        vm.startPrank(fixedDepositor);
        vault.claimFixedPremium();
        vault.withdraw(0);
        vm.stopPrank();

        console.log("Vault's balance at the end: %e", address(vault).balance);
    }
```

```
    function testSecondVulnerabilityPath() public {
        vm.deal(alice, variableCap / 2);
        vm.deal(bob, variableCap / 2);

        vm.prank(alice);
        vault.deposit{value: alice.balance}(1);

        vm.prank(bob);
        vault.deposit{value: bob.balance}(1);

        lido.setRate(1.1e27);

        vm.startPrank(alice);
        vault.withdraw(1);
        vault.finalizeVaultOngoingVariableWithdrawals();
        vm.stopPrank();

        console.log("Alice's balance after first claim: %e", alice.balance);

        skip(1 days + 1);

        lido.setRate(1.21e27);

        vm.startPrank(alice);
        vault.withdraw(1);
        vault.finalizeVaultEndedWithdrawals(1);
        vm.stopPrank();

        vm.prank(bob);
        vault.withdraw(1);

        console.log("Alice's balance at the end: %e", alice.balance);
        console.log("Bob's balance at the end: %e", bob.balance);

        vm.prank(feeReceiver);
        vault.withdraw(1);

        vm.startPrank(fixedDepositor);
        vault.claimFixedPremium();
        vault.withdraw(0);
        vm.stopPrank();

        console.log("Vault's balance at the end: %e", address(vault).balance);
    }
}
```

```
testFirstVulnerabilityPath()
Logs:
  Alice's balance after first claim: 8e18
  Alice's balance at the end: 1.4e19
  Vault's balance at the end: 2e18
```

- Alice's final balance is only `8 ether + 6 ether = 14 ether`
- The `feeReceiver` and `fixedDepositor` have already withdrawn, but there is still ETH left in the contract.

```
testSecondVulnerabilityPath()
Logs:
  Alice's balance after first claim: 4e18
  Alice's balance at the end: 6.95e18
  Bob's balance at the end: 8.449999999999999999e18
  Vault's balance at the end: 1.000000000000000001e18
```

- Alice's final balance is only `4 ether + 2.95 ether = 6.95 ether`
- The `feeReceiver` and `fixedDepositor` have already withdrawn, but there is still ETH left in the contract.

Mitigation

`stakesAmountOwed` at `LidoVault.sol:534` should exclude the `protocolFee`

```
            uint256 protocolFee = ethAmountOwed.mulDiv(protocolFeeBps, 10000);
            totalProtocolFee += protocolFee;
-           uint256 stakesAmountOwed = lido.getSharesByPooledEth(ethAmountOwed);
+           uint256 stakesAmountOwed = lido.getSharesByPooledEth(ethAmountOwed -
 ↪  protocolFee );
```

`totalEarnings` at `LidoVault.sol:775` should exclude the deduction of `totalProtocolFee`

```
-     uint256 totalEarnings = vaultEndingETHBalance.mulDiv(withdrawnStakingEarni
 ↪  ngsInStakes,vaultEndingStakesAmount) - totalProtocolFee +
 ↪  vaultEndedStakingEarnings;
+     uint256 totalEarnings = vaultEndingETHBalance.mulDiv(withdrawnStakingEarni
 ↪  ngsInStakes,vaultEndingStakesAmount) + vaultEndedStakingEarnings;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/saffron-finance/lido-fiv/pull/29

Issue M-1: `LidoVault::vaultEndedWithdraw` doesn't take into consideration income withdrawals before slashing, blocking variable users from withdrwing their income

Source:
https://github.com/sherlock-audit/2024-08-saffron-finance-judging/issues/73

Found by

0x73696d616f, 0xAlix2, tobi0x18

Summary

When FIXED users deposit ETH, they are being deposited in Lido, and Lido might experience slashing. This is expected on the protocol's side, as the impact would be lower income, but it is expected for the protocol to keep functioning as expected, from the contest README:

> These incidents will decrease income from deposits to the Lido Liquid Staking protocol and could decrease the stETH balance. The contract must be operational after it, but it is acceptable for users to lose part of their income/deposit...

However, this isn't always preserved, let's take the following scenario. We have some FIXED value staked in Lido, some profit is accumulated, VARIABLE user withdraws his cut of that profit, by calling `LidoVault::withdraw`. When doing so, `withdrawnStakingEarningsInStakes` gets updated to reflect the amount of withdrawn profit shares, but, this value is calculated after some profit. No more profit comes in, and the vault ends, as soon as it ends, before any withdrawals, the vault gets slashed with some amount. Now, when variable users come to withdraw their profit (slashing didn't remove the whole profit), `totalEarnings` will be calculated wrongly, as the following:

```
uint256 totalEarnings =
↪   vaultEndingETHBalance.mulDiv(withdrawnStakingEarningsInStakes,
↪   vaultEndingStakesAmount) - totalProtocolFee + vaultEndedStakingEarnings;
```

As the used `vaultEndingETHBalance` and `vaultEndingStakesAmount` represent the amounts after slashing, while `withdrawnStakingEarningsInStakes` represents the withdrawn shares before slashing.

This results in wrong `totalEarnings` that also result in wrong `stakingEarningsShare` value for the VARIABLES users, `stakingEarningsShare` will be greater than the

contract's balance, forcing funds to be stuck forever, as `transferWithdrawnFunds` will revert.

Root Cause

When calculating the total earned ETH in `LidoVault::vaultEndedWithdraw`, the protocol doesn't take into consideration the slashing that happened after the vault ended, especially when some VARIABLE users withdrew part of their profit while the vault was still ongoing. `withdrawnStakingEarningsInStakes` will be a misleading value from the previous profit before being slashed. https://github.com/sherlock-audit/2024-08-saffron-finance/blob/main/lido-fiv/contracts/LidoVault.sol#L775

Impact

DOS, variable users can't withdraw their income from the FIXED amount staked.

PoC

Add the following test in `lido-fiv/test/1.LidoVault.test.ts`:

```
it("BUG - DOS, can't withdraw after Slashing", async () => {
  const { lidoVault, addr1, addr2, addr3 } = await
↪  loadFixture(deployLidoVaultFixture)
  const { lidoMock } = await setupMockLidoContracts(lidoVault)

  // Users deposit FIXED and VARIABLE
  await lidoVault.connect(addr1).deposit(SIDE.FIXED, { value: parseEther('1000')
↪  })
  await lidoVault.connect(addr2).deposit(SIDE.VARIABLE, { value:
↪  parseEther('15') })
  await lidoVault.connect(addr3).deposit(SIDE.VARIABLE, { value:
↪  parseEther('15') })

  // Vault has started
  expect(await lidoVault.isStarted()).to.equal(true)

  // User 1 claims FIXED premium
  await lidoVault.connect(addr1).claimFixedPremium()

  // Half time passes
  const { duration, endTime } = await getTimeState(lidoVault)
  await time.increaseTo(endTime - duration / BigInt(2))

  // Lido rebasing, vault earns 100 ETH
  await lidoMock.addStakingEarningsForTargetETH(
    parseEther('1100'),
```

```
    await lidoVault.getAddress()
  )

  // User 2 withdraws their income (part of the above rebasing)
  await lidoVault.connect(addr2).withdraw(SIDE.VARIABLE)

  // Withdrawal was sent to Lido
  expect(
    (await lidoVault.getVariableToVaultOngoingWithdrawalRequestIds(addr2.address
↪   )).length
  ).to.equal(1)
  // `withdrawnStakingEarningsInStakes` is now > 0
  expect(await lidoVault.withdrawnStakingEarningsInStakes()).to.be.greaterThan(0)

  // End time passes
  await time.increaseTo(endTime + BIG_INT_ONE)

  // Vault is ended
  expect(await lidoVault.isEnded()).to.equal(true)

  // Lido slashes the vault
  await lidoMock.subtractStakingEarnings(parseEther('50'))

  // User 1 withdraws their FIXED deposit
  await lidoVault.connect(addr1).withdraw(SIDE.FIXED)
  await lidoVault.connect(addr1).finalizeVaultEndedWithdrawals(SIDE.FIXED)

  // User 3 can't withdraw his income
  await expect(
    lidoVault.connect(addr3).finalizeVaultEndedWithdrawals(SIDE.VARIABLE)
  ).to.be.revertedWith('ETF')
})
```

Mitigation

In `LidoVault::vaultEndedWithdraw`, when calculating the `totalEarnings` when a variable user is withdrawing, consider the income that was withdrawn before Lido slashing happens. Maybe have something like the following?

```
totalEarnings = Math.min(totalEarnings, vaultEndingETHBalance);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/saffron-finance/lido-fiv/pull/29

Issue M-2: Withdrawing after a slash event before the vault has ended will decrease `fixedSidestETHOnStartCapacity` by less than it should, so following users will withdraw more their initial deposit

Source: https://github.com/sherlock-audit/2024-08-saffron-finance-judging/issues/92

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

In LidoVault::withdraw(), when the vault has started but not ended, it limits the value to withdraw if a slashing event occured and withdraws `lidoStETHBalance.mulDiv(fixedBearerToken[msg.sender], fixedLidoSharesTotalSupply());`. However, it also decreases `fixedSidestETHOnStartCapacity` by this same amount, which means that next users that withdraw will get more than their initial deposit in case the Lido ratio comes back up (likely during the vault's duration).

It's clear from the code users should get exactly their initial amount of funds or less, never more, as the comment indicates:

> since the vault has started only withdraw their initial deposit equivalent in stETH at the start of the vault- unless we are in a loss

Root Cause

In `LidoVault.sol:498`, `fixedSidestETHOnStartCapacity` is decreased by a lower amount than it should.

Internal pre-conditions

None.

External pre-conditions

Lido slash, which is in scope as per the readme.

> The Lido Liquid Staking protocol can experience slashing incidents (such as this https://blog.lido.fi/post-mortem-launchnodes-slashing-incident/). These incidents will decrease income from deposits to the Lido Liquid Staking protocol and could decrease the stETH balance. The contract must be operational after it

Attack Path

1. Lido slashes, decreasing the steth ETH / share ratio.
2. User withdraws, taking a loss and decreasing `fixedSidestETHOnStartCapacity` with the lossy amount.
3. Next user withdrawing will withdraw more because `fixedSidestETHOnStartCapacity` will be bigger than it should.

Impact

Fixed deposit users benefit from the slashing event at the expense of variable users who will take the loss.

PoC

Assume that there 100 ETH and 100 shares. A slashing event occurs and drops the ETH to 90 and shares remain 100. There are 2 fixed depositors, with 50% of the deposits each. User A withdraws, and should take `100 ETH * 50 / 100 == 50 ETH`, but takes `90 ETH * 50 / 100 == 45 ETH` instead due to the loss. `fixedSidestETHOnStartCapacity` is decreased by `45 ETH`, the withdrawn amount, so it becomes `55 ETH`. Now, when LIDO recovers from the slashing, the contract will hold more steth than `fixedSidestETHOnStartCapacity`, more specifically the remaining 45 ETH in the contract that were not withdrawn yet are worth 50 ETH now. So user B gets `fixedSidestETHOnStartCapacity * 50 / 50 == 55`.

As the fixed deposit user initially deposited 50, but claimed 55 now, it is getting much more than it should at the expense of the variable users who will take the loss.

Mitigation

The `fixedSidestETHOnStartCapacity` should be always reduced by `fixedETHDeposits.mulDiv(fixedBearerToken[msg.sender], fixedLidoSharesTotalSupply());`, such that users get their equivalent ETH from their initial deposit back and the variable users don't take losses.

Issue M-3: Attacker will DoS `LidoVault` up to 36 days which will ruin expected apr for all parties involved

Source: https://github.com/sherlock-audit/2024-08-saffron-finance-judging/issues/105

Found by

0x73696d616f

Summary

The parameters of each `LidoVault` are tuned such that fixed depositors get an upfront premium and variable depositors in return get all the yield produxed by the fixed depositors' deposits.

However, the protocol does not account for the fact that Lido may be DoSed for up to 36 days if it enters bunker mode. Assuming the return is 4% a year, users are losing approximately `4 * 36 / 365 == 0.4 %`, which goes against the intended returns of the protocol.

Additionally, an attacker may forcefully trigger this by transferring only up to 100 wei of steth, which will make the protocol request an withdrawal and be on hold for 36 days.

The protocol should allow users to withdraw by swapping or similar, taking a much lower amount such as 0.12%, described here.

Root Cause

In `LidoVault:712`, anyone may transfer just 100 wei of steth and DoS the protocol, so fixed, variable and the owner can not withdraw their funds for up to 36 days.

Internal pre-conditions

None.

External pre-conditions

Lido enters bunker mode, which is in scope as it happens when a mass slashing event happens, which is in scope

> The Lido Liquid Staking protocol can experience slashing incidents (such as this https://blog.lido.fi/post-mortem-launchnodes-slashing-incident/). These incidents will decrease income from deposits to the Lido Liquid

Staking protocol and could decrease the stETH balance. The contract must be operational after it

Attack Path

1. Vault has already requested all withdrawals, but they have not yet been claimed, so funds are in the protocol but it does not hold stEth anymore.
2. Attacker transfers 100 wei of steth, triggering the request of this 100 steth.
3. All funds are DoSed for 36 days.

Impact

36 days DoS, which means the protocol can not get the expected interest rate calculated.

PoC

Look at the function `LidoVault::vaultEndedWithdraw()` for confirmation.

Mitigation

Firstly, an attacker should not be able to transfer 100 wei of steth and initiate a request because of this. The threshold should be computed based on an estimated earnings left to withdraw for variable depositors and fixed depositors that have not claimed, not just 100.

Secondly, it would be best if there was an alternative way to withdraw in case requests are taking too much.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/saffron-finance/lido-fiv/pull/30

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.