Obol Audit Report

Obol Manager Contracts

Prepared by: Zach Obront
Sept 18 to 22, 2023

About *Obol*

The Obol Network is an ecosystem for trust minimized staking that enables people to create, test, run & co-ordinate distributed validators.

The Obol Manager contracts are responsible for distributing validator rewards and withdrawals among the validator and node operators involved in a distributed validator.

About *zachobront*

Zach Obront is an independent smart contract security researcher. He serves as a Lead Senior Watson at Sherlock, a Security Researcher at Spearbit, and has identified multiple critical severity bugs in the wild, including in a Top 5 Protocol on Immunefi. You can say hi on Twitter at @zachobront.

Summary & Scope

The ObolNetwork/obol-manager-contracts repository was audited at commit 50ce277919723c80b96f6353fa8d1f8facda6e0e.

The following contracts were in scope:

- src/controllers/ImmutableSplitController.sol
- src/controllers/ImmutableSplitControllerFactory.sol
- src/lido/LidoSplit.sol
- src/lido/LidoSplitFactory.sol
- src/owr/OptimisticWithdrawalReceiver.sol
- src/owr/OptimisticWithdrawalReceiverFactory.sol

After completion of the fixes, the 2f4f059bfd145f5f05d794948c918d65d222c3a9 commit was reviewed. After this review, the updated Lido fee share system in PR 96 was reviewed.

Summary of Findings

ID	Title	Severity	Fixed
M-01	Future fees may be skirted by setting a non-ETH reward token	Medium	✓
M-02	Splits with 256 or more node operators will not be able to switch on fees	Medium	✓
M-03	In a mass slashing event, node operators are incentivized to get slashed	Medium	
L-01	Obol fees will be applied retroactively to all non-distributed funds in the Splitter	Low	✓
L-02	If OWR is used with rebase tokens and there's a negative rebase, principal can be lost	Low	✓
L-03	LidoSplit can receive ETH, which will be locked in contract	Low	✓
L-04	Upgrade to latest version of Solady to fix LibClone bug	Low	✓
G-01	stETH and wstETH addresses can be saved on implementation to save gas	Gas	✓
G-02	OWR can be simplified and save gas by not tracking distributedFunds	Gas	✓
I-01	Strong trust assumptions between validators and node operators	Info	
I-02	Provide node operator checklist to validate setup	Info	

Detailed Findings

[M-01] Future fees may be skirted by setting a non-ETH reward token

Fees are planned to be implemented on the `rewardRecipient` splitter by updating to a new fee structure using the `ImmutableSplitController`.

It is assumed that all rewards will flow through the splitter, because (a) all distributed rewards less than 16 ETH are sent to the `rewardRecipient`, and (b) even if a team waited for rewards to be greater than 16 ETH, rewards sent to the `principalRecipient` are capped at the `amountOfPrincipalStake`.

This creates a fairly strong guarantee that reward funds will flow to the `rewardRecipient`. Even if a user were to set their `amountOfPrincipalStake` high enough that the `principalRecipient` could receive unlimited funds, the Obol team could call `distributeFunds()` when the balance got near 16 ETH to ensure fees were paid.

However, if the user selects a non-ETH token, all ETH will be withdrawable only through the `recoverFunds()` function. If they set up a split with their node operators as their `recoveryAddress`, all funds will be withdrawable via `recoverFunds()` without ever touching the `rewardRecipient` or paying a fee.

Recommendation

Remove the OWR's ability to use a non-ETH token.

Alternatively, if it feels like it may be a use case that is needed, it may make sense to always include ETH as a valid token, in addition to any `OWRToken` set.

Review

Fixed in PR 85 by removing the ability to use non-ETH tokens.

[M-02] Splits with 256+ node operators will not be able to switch on fees

0xSplits is used to distribute rewards across node operators. All Splits are deployed with an ImmutableSplitController, which is given permissions to update the split one time to add a fee for Obol at a future date.

The Factory deploys these controllers as Clones with Immutable Args, hard coding the `owner`, `accounts`, `percentAllocations`, and `distributorFee` for the future update. This data is packed as follows:

```
  function _packSplitControllerData(
    address owner,
    address[] calldata accounts,
    uint32[] calldata percentAllocations,
    uint32 distributorFee
  ) internal view returns (bytes memory data) {
    uint256 recipientsSize = accounts.length;
    uint256[] memory recipients = new uint[](recipientsSize);

    uint256 i = 0;
    for (; i < recipientsSize;) {
      recipients[i] = (uint256(percentAllocations[i]) << ADDRESS_BITS) |
uint256(uint160(accounts[i]));

      unchecked {
        i++;
      }
    }

    data = abi.encodePacked(splitMain, distributorFee, owner,
uint8(recipientsSize), recipients);
  }
```

In the process, `recipientsSize` is unsafely downcasted into a `uint8`, which has a maximum value of `256`. As a result, any values greater than 256 will overflow and result in a lower value of `recipients.length % 256` being passed as `recipientsSize`.

When the Controller is deployed, the full list of `percentAllocations` is passed to the `validSplit` check, which will pass as expected. However, later, when `updateSplit()` is called, the `getNewSplitConfiguation()` function will only return the first `recipientsSize` accounts, ignoring the rest.

```
  function getNewSplitConfiguration()
    public
    pure
    returns (address[] memory accounts, uint32[] memory percentAllocations)
  {
    // fetch the size first
    // then parse the data gradually
    uint256 size = _recipientsSize();
    accounts = new address[](size);
    percentAllocations = new uint32[](size);

    uint256 i = 0;
    for (; i < size;) {
      uint256 recipient = _getRecipient(i);
      accounts[i] = address(uint160(recipient));
      percentAllocations[i] = uint32(recipient >> ADDRESS_BITS);
      unchecked {
        i++;
      }
    }
  }
```

When `updateSplit()` is eventually called on `splitsMain` to turn on fees, the `validSplit()` check on that contract will revert because the sum of the percent allocations will no longer sum to `1e6`, and the update will not be possible.

Proof of Concept

The following test can be dropped into a file in `src/test` to demonstrate that passing 400 accounts will result in a `recipientSize` of `400 - 256 = 144`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import { Test } from "forge-std/Test.sol";
import { console } from "forge-std/console.sol";
import { ImmutableSplitControllerFactory } from
"src/controllers/ImmutableSplitControllerFactory.sol";
import { ImmutableSplitController } from
"src/controllers/ImmutableSplitController.sol";
```

```
interface ISplitsMain {
    function createSplit(address[] calldata accounts, uint32[] calldata
percentAllocations, uint32 distributorFee, address controller) external
returns (address);
}

contract ZachTest is Test {
    function testZach_RecipientSizeCappedAt256Accounts() public {
        vm.createSelectFork("INSERT_RPC_URL");

        ImmutableSplitControllerFactory factory = new
ImmutableSplitControllerFactory(address(9999));
        bytes32 deploymentSalt =
keccak256(abi.encodePacked(uint256(1102)));
        address owner = address(this);

        address[] memory bigAccounts = new address[](400);
        uint32[] memory bigPercentAllocations = new uint32[](400);

        for (uint i = 0; i < 400; i++) {
            bigAccounts[i] = address(uint160(i));
            bigPercentAllocations[i] = 2500;
        }

        // confirmation that 0xSplits will allow creating a split with this
many accounts
        // dummy acct passed as controller, but doesn't matter for these
purposes
        address split =
ISplitsMain(0x2ed6c4B5dA6378c7897AC67Ba9e43102Feb694EE).createSplit(bigAcco
unts, bigPercentAllocations, 0, address(8888));

        ImmutableSplitController controller =
factory.createController(split, owner, bigAccounts, bigPercentAllocations,
0, deploymentSalt);

        // added a public function to controller to read recipient size
directly
        uint savedRecipientSize = controller.ZachTest__recipientSize();
        assert(savedRecipientSize < 400);
        console.log(savedRecipientSize); // 144
    }
```

```
}
```

Recommendation

When packing the data in `_packSplitControllerData()`, check `recipientsSize` before downcasting to a uint8:

```
function _packSplitControllerData(
    address owner,
    address[] calldata accounts,
    uint32[] calldata percentAllocations,
    uint32 distributorFee
) internal view returns (bytes memory data) {
    uint256 recipientsSize = accounts.length;
+   if (recipientsSize > 256) revert
InvalidSplit__TooManyAccounts(recipientSize);
    ...
}
```

Review

Fixed as recommended in PR 86.

[M-03] Node operators incentivized to get slashed in mass slashing event

When the `OptimisticWithdrawalRecipient` receives funds from the beacon chain, it uses the following rule to determine the allocation:

"If the amount of funds to be distributed is greater than or equal to 16 ether, it is assumed that it is a withdrawal (to be returned to the principal, with a cap on principal withdrawals of the total amount they deposited). Otherwise, it is assumed that the funds are rewards."

This value being as low as 16 ether protects against any predictable attack the node operator could perform. For example, due to the effect of hysteresis in updating effective balances, it does not seem to be possible for node operators to predictably

bleed a withdrawal down to be below 16 ether (even if they timed a slashing perfectly).

However, in the event of a mass slashing event, slashing punishments can be much more severe than they otherwise would be. To calculate the size of a slash, we:

- take the total percentage of validator stake slashed in the 18 days preceding and following a user's slash
- multiply this percentage by 3 (capped at 100%)
- the full slashing penalty for a given validator equals 1/32 of their stake, plus the resulting percentage above applied to the remaining 31/32 of their stake

In order for such penalties to bring the withdrawal balance below 16 ether (assuming a full 32 ether to start), we would need the percentage taken to be greater than `15 / 31 = 48.3%`, which implies that `48.3 / 3 = 16.1%` of validators would need to be slashed.

Because the measurement is taken from the 18 days before and after the incident, node operators would have the opportunity to see a mass slashing event unfold, and later decide that they would like to be slashed along with it.

In the event that they observed that greater than 16.1% of validators were slashed, Obol node operators would be able to get themselves slashed, be exited with a withdrawal of less than 16 ether, and claim that withdrawal as rewards, effectively stealing from the principal recipient.

Recommendations

Find a solution that provides a higher level of guarantee that the funds withdrawn are actually rewards, and not a withdrawal.

Review

Acknowledged. We believe this is a black swan event. It would require a major ETH client to be compromised, and would be a betrayal of trust, so likely not EV+ for doxxed operators. Users of this contract with unknown operators should be wary of such a risk.

[L-01] Obol fees will be applied retroactively to all non-distributed funds in the Splitter

When Obol decides to turn on fees, a call will be made to `ImmutableSplitController::updateSplit()`, which will take the predefined split parameters (the original user specified split with Obol's fees added in) and call `updateSplit()` to implement the change.

```
function updateSplit() external payable {
    if (msg.sender != owner()) revert Unauthorized();

    (address[] memory accounts, uint32[] memory percentAllocations) =
getNewSplitConfiguration();

    ISplitMain(splitMain()).updateSplit(split, accounts,
percentAllocations, uint32(distributorFee()));
}
```

If we look at the code on `SplitsMain`, we can see that this `updateSplit()` function is applied retroactively to all funds that are already in the split, because it updates the parameters without performing a distribution first:

```
function updateSplit(
    address split,
    address[] calldata accounts,
    uint32[] calldata percentAllocations,
    uint32 distributorFee
)
    external
    override
    onlySplitController(split)
    validSplit(accounts, percentAllocations, distributorFee)
{
    _updateSplit(split, accounts, percentAllocations, distributorFee);
}
```

This means that any funds that have been sent to the split but have not yet be distributed will be subject to the Obol fee. Since these splitters will be accumulating all execution layer fees, it is possible that some of them may have received large MEV bribes, where this after-the-fact fee could be quite expensive.

Recommendation

The most strict solution would be for the `ImmutableSplitController` to store both the old split parameters and the new parameters. The old parameters could first be used to call `distributeETH()` on the split, and then `updateSplit()` could be called with the new parameters.

If storing both sets of values seems too complex, the alternative would be to require that `split.balance <= 1` to update the split. Then the Obol team could simply store the old parameters off chain to call `distributeETH()` on each split to "unlock" it to update the fees.

(Note that for the second solution, the ETH balance should be less than or equal to 1, not 0, because 0xSplits stores empty balances as `1` for gas savings.)

Review

Fixed as recommended in PR 86.

[L-02] If OWR is used with rebase tokens and there's a negative rebase, principal can be lost

The `OptimisticWithdrawalRecipient` is deployed with a specific token immutably set on the clone. It is presumed that that token will usually be ETH, but it can also be an ERC20 to account for future integrations with tokenized versions of ETH.

In the event that one of these integrations used a rebasing version of ETH (like `stETH`), the architecture would need to be set up as follows:

`OptimisticWithdrawalRecipient => rewards to something like LidoSplit.sol => Split Wallet`

In this case, the OWR would need to be able to handle rebasing tokens.

In the event that rebasing tokens are used, there is the risk that slashing or inactivity leads to a period with a negative rebase. In this case, the following chain of events could happen:

- `distribute(PULL)` is called, setting `fundsPendingWithdrawal == balance`
- rebasing causes the balance to decrease slightly

- `distribute(PULL)` is called again, so when `fundsToBeDistributed = balance - fundsPendingWithdrawal` is calculated in an unchecked block, it ends up being near `type(uint256).max`
- since this is more than `16 ether`, the first `amountOfPrincipalStake - _claimedPrincipalFunds` will be allocated to the principal recipient, and the rest to the reward recipient
- we check that `endingDistributedFunds <= type(uint128).max`, but unfortunately this check misses the issue, because only `fundsToBeDistributed` underflows, not `endingDistributedFunds`
- `_claimedPrincipalFunds` is set to `amountOfPrincipalStake`, so all future claims will go to the reward recipient
- the `pullBalances` for both recipients will be set higher than the balance of the contract, and so will be unusable

In this situation, the only way for the principal to get their funds back would be for the full `amountOfPrincipalStake` to hit the contract at once, and for them to call `withdraw()` before anyone called `distribute(PUSH)`. If anyone was to be able to call `distribute(PUSH)` before them, all principal would be sent to the reward recipient instead.

Recommendation

Similar to M-01, I would recommend removing the ability for the `OptimisticWithdrawalRecipient` to accept non-ETH tokens.

Otherwise, I would recommend two changes for redundant safety:

1) Do not allow the OWR to be used with rebasing tokens.

2) Move the `_fundsToBeDistributed = _endingDistributedFunds - _startingDistributedFunds;` out of the unchecked block. The case where `_endingDistributedFunds` underflows is already handled by a later check, so this one change should be sufficient to prevent any risk of this issue.

Review

Fixed in PR 85 by removing the ability to use non-ETH tokens.

[L-03] LidoSplit can receive ETH, which will be locked in contract

Each new `LidoSplit` is deployed as a clone, which comes with a `receive()` function for receiving ETH.

However, the only function on `LidoSplit` is `distribute()`, which converts `stETH` to `wstETH` and transfers it to the `splitWallet`.

While this contract should only be used for Lido to pay out rewards (which will come in `stETH`), it seems possible that users may accidentally use the same contract to receive other validator rewards (in ETH), or that Lido governance may introduce ETH payments in the future, which would cause the funds to be locked.

Proof of Concept

The following test can be dropped into `LidoSplit.t.sol` to confirm that the clones can currently receive ETH:

```
function testZach_CanReceiveEth() public {
    uint before = address(lidoSplit).balance;
    payable(address(lidoSplit)).transfer(1 ether);
    assertEq(address(lidoSplit).balance, before + 1 ether);
}
```

Recommendation

Introduce an additional function to `LidoSplit.sol` which wraps ETH into stETH before calling `distribute()`, in order to rescue any ETH accidentally sent to the contract.

Review

Fixed in PR 87 by adding a `rescueFunds()` function that can send ETH or any ERC20 (except `stETH` or `wstETH`) to the `splitWallet`.

[L-04] Upgrade to latest version of Solady to fix LibClone bug

In the recent Solady audit, an issue was found that effects LibClone.

In short, LibClone assumes that the length of the immutable arguments on the clone will fit in 2 bytes. If it's larger, it overlaps other op codes and can lead to strange behaviors, including causing the deployment to fail or causing the deployment to succeed with no resulting bytecode.

Because the `ImmutableSplitControllerFactory` allows the user to input arrays of any length that will be encoded as immutable arguments on the Clone, we can manipulate the length to accomplish these goals.

Fortunately, failed deployments or empty bytecode (which causes a revert when `init()` is called) are not problems in this case, as the transactions will fail, and it can only happen with unrealistically long arrays that would only be used by malicious users.

However, it is difficult to be sure how else this risk might be exploited by using the overflow to jump to later op codes, and it is recommended to update to a newer version of Solady where the issue has been resolved.

Proof of Concept

If we comment out the `init()` call in the `createController()` call, we can see that the following test "successfully" deploys the controller, but the result is that there is no bytecode:

```
function testZach__CreateControllerSoladyBug() public {
    ImmutableSplitControllerFactory factory = new
ImmutableSplitControllerFactory(address(9999));
    bytes32 deploymentSalt = keccak256(abi.encodePacked(uint256(1102)));
    address owner = address(this);

    address[] memory bigAccounts = new address[](28672);
    uint32[] memory bigPercentAllocations = new uint32[](28672);

    for (uint i = 0; i < 28672; i++) {
        bigAccounts[i] = address(uint160(i));
        if (i < 32) bigPercentAllocations[i] = 820;
        else bigPercentAllocations[i] = 34;
```

```
    }

    ImmutableSplitController controller =
factory.createController(address(8888), owner, bigAccounts,
bigPercentAllocations, 0, deploymentSalt);
    assert(address(controller) != address(0));
    assert(address(controller).code.length == 0);
}
```

Recommendation

Delete Solady and clone it from the most recent commit, or any commit after the fixes from PR 548 were merged.

Review

Solady has been updated to v.0.0.123 in PR 88.

[G-01] stETH and wstETH addresses can be saved on implementation to save gas

The `LidoSplitFactory` contract holds two immutable values for the addresses of the `stETH` and `wstETH` tokens.

When new clones are deployed, these values are encoded as immutable args. This adds the values to the contract code of the clone, so that each time a call is made, they are passed as calldata along to the implementation, which reads the values from the calldata for use.

Since these values will be consistent across all clones on the same chain, it would be more gas efficient to store them in the implementation directly, which can be done with `immutable` storage values, set in the constructor.

This would save 40 bytes of calldata on each call to the clone, which leads to a savings of approximately 640 gas on each call.

Recommendation

1) Add the following to `LidoSplit.sol`:

```
address immutable public stETH;
address immutable public wstETH;
```

2) Add a constructor to `LidoSplit.sol` which sets these immutable values. Solidity treats immutable values as constants and stores them directly in the contract bytecode, so they will be accessible from the clones.

3) Remove `stETH` and `wstETH` from `LidoSplitFactory.sol`, both as storage values, arguments to the constructor, and arguments to `clone()`.

4) Adjust the `distribute()` function in `LidoSplit.sol` to read the storage values for these two addresses, and remove the helper functions to read the clone's immutable arguments for these two values.

Review

Fixed as recommended in PR 87.

[G-02] OWR can be simplified and save gas by not tracking distributedFunds

Currently, the `OptimisticWithdrawalRecipient` contract tracks four variables:

- distributedFunds: total amount of the token distributed via push or pull
- fundsPendingWithdrawal: total balance distributed via pull that haven't been claimed yet
- claimedPrincipalFunds: total amount of funds claimed by the principal recipient
- pullBalances: individual pull balances that haven't been claimed yet

When `_distributeFunds()` is called, we perform the following math (simplified to only include relevant updates):

```
endingDistributedFunds = distributedFunds - fundsPendingWithdrawal +
currentBalance;
fundsToBeDistributed = endingDistributedFunds - distributedFunds;
distributedFunds = endingDistributedFunds;
```

As we can see, `distributedFunds` is added to the `endingDistributedFunds` variable and then removed when calculating `fundsToBeDistributed`, having no impact on the resulting `fundsToBeDistributed` value.

The `distributedFunds` variable is not read or used anywhere else on the contract.

Recommendation

We can simplify the math and save substantial gas (a storage write plus additional operations) by not tracking this value at all.

This would allow us to calculate `fundsToBeDistributed` directly, as follows:

```
fundsToBeDistributed = currentBalance - fundsPendingWithdrawal;
```

Review

Fixed as recommended in PR 85.

[I-01] Strong trust assumptions between validators and node operators

It is assumed that validators and node operators will always act in the best interest of the group, rather than in their selfish best interest.

It is important to make clear to users that there are strong trust assumptions between the various parties involved in the DVT.

Here are a select few examples of attacks that a malicious set of node operators could perform:

1) Since there is currently no mechanism for withdrawals besides the consensus of the node operators, a minority of them sufficient to withhold consensus could blackmail the principal for a payment of up to 16 ether in order to allow them to

withdraw. Otherwise, they could turn off their node operators and force the principal to bleed down to a final withdrawn balance of just over 16 ether.

2) Node operators are all able to propose blocks within the P2P network, which are then propagated out to the rest of the network. Node software is accustomed to signing for blocks built by block builders based on the metadata including quantity of fees and the address they'll be sent to. This is enforced by social consensus, with block builders not wanting to harm validators in order to have their blocks accepted in the future. However, node operators in a DVT are not concerned with the social consensus of the network, and could therefore build blocks that include large MEV payments to their personal address (instead of the DVT's 0xSplit), add fictitious metadata to the block header, have their fellow node operators accept the block, and take the MEV for themselves.

3) While the withdrawal address is immutably set on the beacon chain to the OWR, the fee address is added by the nodes to each block. Any majority of node operators sufficient to reach consensus could create a new 0xSplit with only themselves on it, and use that for all execution layer fees. The principal (and other node operators) would not be able to stop them or withdraw their principal, and would be stuck with staked funds paying fees to the malicious node operators.

Note that there are likely many other possible attacks that malicious node operators could perform. This report is intended to demonstrate some examples of the trust level that is needed between validators and node operators, and to emphasize the importance of making these assumptions clear to users.

Review

Acknowledged. We believe EIP 7002 will reduce this trust assumption as it would enable the validator exit via the execution layer withdrawal key.

[I-02] Provide node operator checklist to validate setup

There are a number of ways that the user setting up the DVT could plant backdoors to harm the other users involved in the DVT.

Each of these risks is possible to check before signing off on the setup, but some are rather hidden, so it would be useful for the protocol to provide a list of checks that node operators should do before signing off on the setup parameters (or, even better, provide these checks for them through the front end).

1) Confirm that `SplitsMain.getHash(split)` matches the hash of the parameters that the user is expecting to be used.

2) Confirm that the controller clone delegates to the correct implementation. If not, it could be pointed to delegate to `SplitMain` and then called to `transferControl()` to a user's own address, allowing them to update the split arbitrarily.

3) `OptimisticWithdrawalRecipient.getTranches()` should be called to check that `amountOfPrincipalStake` is equal to the amount that they will actually be providing.

4) The controller's `owner` and future split including Obol fees should be provided to the user. They should be able to check that `ImmutableSplitControllerFactory.predictSplitControllerAddress()`, with those parameters inputted, results in the controller that is actually listed on `SplitsMain.getController(split)`.

Review

Acknowledged. We do some of these already (will add the remainder) automatically in the launchpad UI during the cluster confirmation phase by the node operator. We will also add it in markdown to the repo.