stela labs

Security review

Vault strategy

From:
Stela Labs
Info@stelalabs.com

To:
Composable finance

Vault strategy review

The following report is a result of a security review from **Stela Labs** (Info@stelalabs.com), requested by **Composable finance** team to conduct a security check of their smart contracts. 12 days were spent on the review.

The review started on a codebase with a following commit hash: `af8a2afdbe2a1b137443f8eb41e50efa9c5f2257` . The scope was narrowed down to only `HarvestSC.sol` contract and it's dependencies. The code was changing multiple times during the review, so some issues are based on a different commit hash.

Summary

The code has a highly complex structure and a lot of external calls and dependencies. That fact makes it very hard to ensure that there are no bugs in the codebase and that nothing can go wrong. As a result of the review, many potential problems are partially mitigated by upgradability; that solution saves from funds lock-up but increases trust in the system's owners. During the review, we also noticed the lack of tests as many of the found issues should have been covered by the tests. It is highly recommended to increase the coverage and use more tools to test all the scenarios properly. The issues and recommendations are listed below.

Issues

1. **CRITICAL. Reward distribution does not match the distribution of the underlying Harvest's reward function.**

   The rewards from staking to the Harvest's contract are distributed to the users only when they withdraw their funds from the system (i.e., from the `HarvestSC.sol` contract). It's intended that if you withdraw `x%` of your funds, you'll receive `x%` of the earned rewards. Because of that, all the earned staking rewards are stored in the `HarvestSC.sol` contract before they are redeemed. So the amount of reward tokens in the contract is accumulating over time.

   When distributing the rewards, in the _getPendingRewards function, the amount of rewards are calculated without taking into account the duration of staking, i.e. any deposit is processed as if it was there from the very beginning. So new users with new deposits are stealing staking rewards from the old users:

   ```
   uint256 rewardPerBlock = 0;
           uint256 balance = IERC20(farmToken).balanceOf(address(this));
           if (balance == 0) {
               return 0;
           }
           uint256 diff = block.timestamp.sub(firstDepositTimestamp);
           if (diff == 0) {
               rewardPerBlock = balance;
           } else {
               rewardPerBlock = balance.div(diff);
           }
           uint256 rewardPerBlockUser =
               rewardPerBlock.mul(block.timestamp.sub(user.timestamp));
           uint256 ratio = _getRatio(toCalculateForAmount, totalDeposits, 18);
           return (rewardPerBlockUser.mul(ratio)).div(10**18);
   ```

   Recommendations. Duplicate Harvest's staking reward distribution.

   ***Update**: The issue is fixed by copying Harvest's staking reward distribution.*

2. **CRITICAL. `user.amountToken` should be decreased during the withdrawal according to the current `user.underlyingRatio` .**

   The `UserInfo` structure keeps three closely interconnected variables: `User.amountToken` - the number of deposited tokens. `User.amountfToken` - the amount of received ftokens during the deposits. `User.underlyingRatio` - the average underlying ratio of the deposited tokens and the ftokens received during the deposits. So the `user.underlyingRatio` should always be equal to `user.amountfToken / user.amountToken` . Any of these variables can even be removed as redundant because they can be received from the other two.

The `user.underlyingRatio` is needed because all the tokens earned from the yield farming are subject to fees. So if the withdrawal ratio is lower than the `user.underlyingRatio`, the fees are implied to the profit. Logically, `user.underlyingRatio` should remain the same after the withdrawal as it represents the average ratio of the deposits.

In the codebase, the `user.underlyingRatio` is recalculated and usually significantly changed after each withdrawal, leading to either lowering or increasing fees. The users that know about the bug can trick the system into reducing the fee-able token amount substantially; other users may unintentionally even increase their fees.

Recommendations. This all happens because during the withdrawal, the `user.amountfToken` and the `user.amountToken` are decreased in a different proportion(current ratio in the Harvest's contracts) from the `user.underlyingRatio`:

```
if (user.amountfToken == 0) //full exit
        {
            user.amountToken = 0;
        } else {
            if (user.amountToken > results.totalToken) {
                user.amountToken = user.amountToken.sub(results.totalToken);
            } else {
                user.amountToken = 0;
            }
        }
        user.underlyingRatio = _getRatio(
            user.amountfToken,
            user.amountToken,
            18
        );
```

One of the suggestions would be to decrease the `user.amountToken` in a way, so the `user.underlyingRatio` will remain the same.

***Update**: the issue is fixed by properly decresaing the `user.amountToken` to maintain the `user.underlyingRatio`:*

```
results.calculatedTokenAmount = (amount.mul(10**18)).div(
            user.underlyingRatio
        );
        if (user.amountfToken == 0) {
            user.amountToken = 0;
        } else {
            if (results.calculatedTokenAmount <= user.amountToken) {
                user.amountToken = user.amountToken.sub(
                    results.calculatedTokenAmount
                );
            } else {
                user.amountToken = 0;
            }
        }
```

3. **CRITICAL. The receipt tokens are not burned if the full amount is burned.** In the _calculatefTokenRemainings function, the `burnAmount` is going to be zero if all the tokens are withdrawn:

   ```
   amountfToken = 0;
   if (!wasUserBlacklisted) {
         burnAmount = amountfToken;
   }
   ```

   Because of that, receipt tokens won't be burned, and they can be infinitely minted by repeating the deposit-withdraw cycle.

   ***Update**: the issue was fixed by calculating the `burnAmount` first and removing the blacklist.*

4. **HIGH. The TimelockOwner is not doing what intended and is overcomplicated.**

   The `TimelockOwner` contract is supposed to provide the timelock functionality to avoid instant unpredictable changes in the system. The contract is overcomplicated because it contains all the possible functions that the owner can call. The contract will grow and will be manually changed as the system evolves. It also does not include all the data when announcing the transaction because it does not specify which function will be called:

```
function startUpdate(
     address _strategy,
     address _newAddress,
     uint256 _newValue
 ) external onlyOwner {
     Info storage details = timestamps[_strategy];
     details.newAddress = _newAddress;
     details.newValue = _newValue;
     details.moment = block.timestamp;
 }
```

Recommendations. It's highly recommended to switch to a general-purpose timelock contract.

Update: *fixed by switching to Compound's timelock contract.*

5. **HIGH. The `results.auctionedEth` value is not changed when it should**.

   The following lines should increase the `results.auctionedEth` :

   ```
   results.auctionedEth.add(swapAuctionedTokenResult);
   ```

   Update: *fixed*.

6. **HIGH.Many dependencies and future changes in Ethereum can potentially lock the funds.**

   The `withdraw` function is performing a lot of operations and external calls during the execution. All the external contracts can revert, fail, be upgraded, or potentially be hacked. If any of that happens, the `withdraw` function will probably revert, and there will be no way to retrieve the funds. In addition to the dependencies, any future changes in gas cost/gas limits may make this function run out of gas.

   Recommendations. It is impossible to predict everything that may happen in the external calls. If they all are needed and can't be avoided, it makes sense to make the contract upgradable to handle any unexpected situation. The upgrade should be executed in a more or less trustless manner by using a reasonably long timelock. Ideally, the upgrade should only happen in case of major failure and if the funds are locked.

   Update: *fixed by making the contract upgradable.*

7. **MEDIUM. The `totalToken` should also change like `user.amountToken` .**

   The `totalToken` variable is needed to cap the number of tokens deposited. So when the previous issue with `user.amountToken` is fixed, the `totalToken` amount should be decreased by the same amount as `user.amountToken` to represent the total amount of the deposited tokens properly.

   Update: *fixed as recommended.*

8. **MEDIUM. Harvest has a greylist.**

   Harvest's vault contract may ban arbitrary contracts from using their system. If that happens, the funds can be stuck in case of greylisting.

9. **MEDIUM. Harvest's staking contract misbehavior is handled inconsistently**.

   Harvest's staking contract is required to return exactly the same amount of ftokens submitted during the deposit. But in the `HarvestSC.sol` contract, it is assumed that the number of withdrawn tokens can be different:

   ```
   results.obtainedfToken = _unstakefTokenFromHarvestPool(amount);
   ```

   According to the staking contract's logic, the `results.obtainedfToken` and the `amount` values should be the same. Therefore, it's impossible to predict what went wrong if they are not equal, but if that happens, the logic that handles this error should be consistent. The problem is that in the `HarvestSC.sol` contract, both values are used afterward, and every choice behind using `amount` or `results.obtainedfToken` looks random. **Recommendations**. Come up with a consistent logic on how to handle the situation when the `results.obtainedfToken` and the `amount` are not equal.

__Update__: fixed by using mentioned values more consistently. If the staking contract returns a different amount of tokens, it's assumed in the accounting that the intended value(`amount`) was withdrawn.

10. **MEDIUM. The `user.timestamp` is only submitted once.**.

    The `user.timestamp is needed to make sure that the user can't withdraw immediately after the deposit.

    ```
    function isWithdrawalAvailable(address user) public view returns (bool) {
       if (lockTime > 0) {
             return userInfo[user].timestamp.add(lockTime) <= block.timestamp;
       }
       return true;
    }
    ```

    The `user.timestamp` is only changed if the current timestamp is zero, which only happens during the first deposit. So it won't prevent a user from doing instant deposit-withdrawal afterward.

    ```
    if (user.timestamp == 0) {
             user.timestamp = block.timestamp;
       }
    ```

    Recommendations. The fix depends on the intention of the `lockTime` mechanism. The `user.timestamp` should be either updated after each deposit or stored per each user's separate deposit.

    __Update__: the `user.timestamp` is now updated on every deposit.

11. **Optimization**. **Gas cost and code complexity**.

    The code structure is very complicated with a lot of gas inefficiency, many minor trades are happening, so the gas cost can be higher than the value traded. Therefore, it is recommended to avoid making a lot of single exchanges and transfers. Instead, it is better to batch them. In addition to that, the storage is not used efficiently. Some variables are redundant, like `user.underlyingRatio` or `user.amountReceiptToken` calculations and accounting, which is always the same as `user.amountfToken` .