CD SECURITY

AUDIT REPORT

VOX Finance
September 2023

Introduction

A time-boxed security review of the **Vox Finance** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

About **Vox Finance**

The **Vox Finance** protocol allows holders of the `VOX` token to `lock` their tokens into the `VoxStakingPool` or the `VoxLiquidityFarm` in exchange for rewards. The `VoxStakingPool` has a `minimumLock` of 2 weeks and a `maximumLock` of 52 weeks, while the `VoxLiquidityFarm` does not have locking periods. Both of the contracts have a different `withdrawalFee` that is taken from the user. However, it is important to note that the fee can be changed by the `owner` and set to `withdrawalFeeMax`.

The `VOX` token has a 4.0 % fee on each buy and sell transaction which is distributed between `marketingWallet`, `liquidityPool` and a part of it is burned. More documentation and information about the Tokenomics can be found here.

Threat Model

Roles & Actors

- Users - able to stake their `VOX` tokens or `deposit` them to `VoxLiquidityFarm`.
- Owner - able to set critical parameters like `withdrawalFee`, `rewardsDuration`, `setTreasury`, `recoverERC20`, `setLockingPeriods`. It can also add and remove addresses that are `ExcludedFromFees` and `ExcludedMaxTransactionAmount`. The owner has extensive access to functions that are `restricted` or use the `onlyOwner` modifier.
- SwapManager - able to `addLiquidity`, `buyAndBurn` `VOX` tokens, and it is approved to `swap` tokens for `ETH`. Also, it is `ExcludedFromFees` and `ExcludedMaxTransactionAmount`.
- Marketing Wallet - receives 50% of each fee charged on buy/sell transactions.

Security Interview

Q: What in the protocol has value in the market?

A: The `VOX` tokens that are locked in the contract and `rewardsToken`.

Q: What is the worst thing that can happen to the protocol?

A: If the protocol is put into DoS state or locked tokens are stolen.

Q: In what case can the protocol/users lose money?

A: If an attacker is able to drain the `VoxStakingPool` / `VoxLiquidityFarm` or is able to claim the rewards of other users because of miscalculations.

Severity classification

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

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 9c94722e32965b6298d885f6d323fc55bfa8a0e4**

Scope

The following smart contracts were in scope of the audit:

- `VoxLiquidityFarm.sol`
- `VoxStakingPool.sol`
- `VoxSwapManager.sol`
- `VoxToken.sol`
- `VoxTokenAirdrop.sol`
- `VoxVestingWallet.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 3 issues
- Low: 8 issues
- Informational: 12 issues

Findings Summary

ID	Title	Severity

ID	Title	Severity
[H-01]	There is no slippage control in `addLiquidity` and `swapToWeth` methods	High
[M-01]	Owner can steal all of the `stakingToken`	Medium
[M-02]	`notifyRewardAmount` can lead to loss of yields for the users	Medium
[M-03]	`setRewardsDuration` allows setting near zero or enormous rewardsDuration	Medium
[L-01]	Check array arguments have the same length	Low
[L-02]	Use two-step ownership transfer approach	Low
[L-03]	Avoid using `tx.origin` for validation	Low
[L-04]	Missing 0 address check	Low
[L-05]	Handle 0 `reward` case	Low
[L-06]	Set bounds for `multiplier`	Low
[L-07]	Transactions may revert because of a deadline	Low
[L-08]	Add a timelock to restricted to owner functions that set critical values	Low
[I-01]	Using `SafeMath` when compiler is ^0.8.0	Informational
[I-02]	NatSpecs are incomplete	Informational
[I-03]	Make use of Solidity time units	Informational
[I-04]	Use custom errors instead of require statements with string error	Informational
[I-05]	Not used events can be removed	Informational
[I-06]	Unclear error message	Informational
[I-07]	CEI pattern is not followed	Informational
[I-08]	Variables can be turned into an immutable	Informational
[I-09]	Most setter functions do not emit events	Informational
[I-10]	Improper naming	Informational
[I-11]	Contracts are not inheriting their interfaces	Informational
[I-12]	Solidity safe pragma best practices are not used	Informational

Detailed Findings

[H-01] There is no slippage control in `addLiquidity` and `swapToWeth` methods

Severity

Impact: High, as `VoxToken` contract will lose money due to sandwich attacks

Likelihood: Medium, since MEV is very prominent, the chance of that happening is pretty high

Description

File: `VoxSwapManager.sol`

We can see the following code in these functions:

Function: `addLiquidity`

```
router.addLiquidity(
            address(vox),
            vox.weth(),
            voxAmount,
            wethAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            owner(),
            block.timestamp
        );
```

Function: `swapToWeth`

```
 router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            voxAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
```

Function: `buyAndBurn`

```
        router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            wethAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
```

The “0”s here are the value of the `amountOutMin` argument which is used for slippage tolerance. 0 value here essentially means 100% slippage tolerance. This is a very easy target for MEV and bots to do a flash loan sandwich attack on each of the strategy's swaps, resulting in a very big slippage on each trade. 100% slippage tolerance can be exploited in a way that the strategy (so the vault and the users) receive much less value than it should have. This can be done on every trade if the trade transaction goes through a public mempool.

Recommendations

Add a protection parameter to the above-mentioned functions, so that the `VoxToken` contract can specify the minimum out amount.

CLIENT

Acknowledged - corrected.

[M-01] Owner can steal all of the `stakingToken`

Severity

Impact: High, as all of the staked tokens can be withdrawn

Likelihood: Low, as it requires a malicious/compromised owner

Description

The `recoverERC20` function inside `VoxStakingPool` rightfully checks if the passed `tokenAddress` is different from the `rewardsToken` address. However, it does not check if it is not the same as the `stakingToken` address which should be the case as can be seen from the comment:

```
    function recoverERC20(address tokenAddress, uint tokenAmount)
        external
        onlyOwner
    {
        // Cannot recover the staking token or the rewards token
        require(
            tokenAddress != address(rewardsToken),
            "Cannot withdraw the staking or rewards tokens"
        );
        ..
    }
```

This could be exploited by a malicious or compromised owner. This admin privilege allows the owner to sweep the staking tokens, potentially harming depositors by rug-pulling.

Recommendations

Add an additional check inside the require statement:

`tokenAddress != address(stakingToken)`

CLIENT

Acknowledged - corrected.

[M-02] `notifyRewardAmount` can lead to loss of yields for the users

Severity

Impact: High, because users` yield can be manipulated

Likelihood: Low, this is restricted function and only the `owner` can call it

Description

The `notifyRewardAmount` function takes a `reward` amount and extends the `periodFinish` to now + `rewardsDuration`:

`periodFinish = block.timestamp.add(rewardsDuration);`

It rebases the `leftover` rewards and the new `reward` over the `rewardsDuration` period.

```
        if (block.timestamp >= periodFinish) {
            rewardRate = reward.div(rewardsDuration);
        } else {
            uint remaining = periodFinish.sub(block.timestamp);
            uint leftover = remaining.mul(rewardRate);
            rewardRate = reward.add(leftover).div(rewardsDuration);
        }
```

This can lead to a dilution of the reward rate and rewards being dragged out forever by malicious new reward deposits.

Let's take a look at the following example:

1. For the sake of the example, imagine the current `rewardRate` is `1000 rewards / rewardsDuration`.
2. When 10% of `rewardsDuration` has passed, a malicious owner calls `notifyRewards` with `reward = 0`.
3. The new `rewardRate = 0 + 900 / rewardsDuration`, which means the `rewardRate` just dropped by 10%.
4. This can be repeated infinitely. After another 10% of reward time passed, they trigger notifyRewardAmount(0) to reduce it by another 10% again: `rewardRate = 0 + 720 / rewardsDuration`.

The rewardRate should never decrease by a `notifyRewardAmount` call.

Recommendations

There are two potential fixes to this issue:

1. If the `periodFinish` is not changed at all and not extended on every `notifyRewardAmount` call. The `rewardRate` should just increase by `rewardRate += reward / (periodFinish - block.timestamp)`.

2. Keep the `rewardRate` constant but extend `periodFinish` time by `+= reward / rewardRate`.

CLIENT

Acknowledged - corrected.

[M-03] `setRewardsDuration` allows setting near zero or enormous rewardsDuration, which breaks reward logic

Severity

Impact: High, as it breaks reward logic

Likelihood: Low, as it requires an error from the owner's side or a compromised/malicious owner

Description

File: `VoxStakingPool.sol`

`notifyRewardAmount` method will be inoperable if `rewardsDuration` is set to zero. It will cease to produce meaningful results if `rewardsDuration` be too small or too big.

The setter does not control the value, allowing zero/near zero/enormous duration:

```
    function setRewardsDuration(uint _rewardsDuration) external restricted
{
        require(
            block.timestamp > periodFinish,
            "Previous rewards period must be complete before changing the
duration for the new period"
        );
        rewardsDuration = _rewardsDuration;
        emit RewardsDurationUpdated(rewardsDuration);
    }
```

Division by the duration is used in `notifyRewardAmount`:

```
if (block.timestamp >= periodFinish) {
    rewardRate = reward.div(rewardsDuration);
```

Recommendations

Check for min and max range in the rewardsDuration setter, as too small or too big rewardsDuration breaks the logic.

CLIENT

Acknowledged - corrected.

[L-01] Check array arguments have the same length

When the `sendBatch` function is called inside `VoxTokenAirdrop`, two array-type arguments are passed. Validate that the arguments have the same length so you do not get unexpected errors if they don't.

CLIENT

Acknowledged - corrected.

[L-02] Use two-step ownership transfer approach

The `owner` role is crucial for the protocol as there are a lot of functions with the `onlyOwner` and the `restricted` modifiers. Make sure to use a two-step ownership transfer approach by using `Ownable2Step` from OpenZeppelin as opposed to `Ownable` as it gives you the security of not unintentionally sending the `owner` role to an address you do not control. Also, consider using only `onlyOwner` modifier instead of using both `onlyOwner` and `restricted` modifiers because they are basically the same and using both only creates confusion.

CLIENT

Acknowledged - corrected.

[L-03] Avoid using `tx.origin` for validation

Inside `VoxToken.sol`, the following `require` statement is used:

```
   require(
       _holderLastTransferTimestamp[tx.origin] <
       block.number,
       "_transfer:: Transfer Delay enabled.  Only one purchase per block
allowed."
    );
```

This can be easily bypassed if the function is called by a contract. Use `msg.sender` instead of `tx.origin`.

CLIENT

Acknowledged - corrected.

[L-04] Missing 0 address check

In `VoxStakingPool`'s constructor we can see that there is a 0 address check for `stakingToken` but such check is missing for `rewardsToken`.

```
constructor(
        address _rewardsToken,
        address _stakingToken
    ) {
        rewardsToken = IERC20(_rewardsToken);
        if (_stakingToken != address(0)) {
            stakingToken = IERC20(_stakingToken);
        }
    }
```

Consider adding a 0 address check for `rewardsToken` as well.

CLIENT

Acknowledged - corrected.

[L-05] Handle 0 `reward` case

In `getReward` a check is missing if the rewards are equal to 0.

Consider adding the following check with a custom errror:

```
function getReward() public nonReentrant updateReward(msg.sender) {
        uint reward = rewards[msg.sender];
+       if(reward == 0) revert ZeroRewards();
        if (reward > 0) {
            rewards[msg.sender] = 0;
            rewardsToken.safeTransfer(msg.sender, reward);
            emit RewardPaid(msg.sender, reward);
        }
```

CLIENT

Acknowledged - corrected.

[L-06] Set bounds for `multiplier`

In `setMultiplier` the owner of the contract can set a new value for the `multiplier`. However, there might be a problem if there is a compromised or malicious owner. Set a max bound in `setMultplier`.

CLIENT

Acknowledged - corrected.

[L-07] Transactions may revert because of a deadline

In the `VoxSwapManager`, the `router.addLiquidity` is called and the `block.timestamp` is passed as `deadline`. This means that if the execution takes longer than the current timestamp, the transaction will revert as it can be seen from the Uniswap documentation. It is the same for `router.swapExactTokensForTokensSupportingFeeOnTransferTokens` and `router.swapExactTokensForTokensSupportingFeeOnTransferTokens`. Consider changing it to `block.timestamp + 2 minutes`, for example, to give it a bit of tolerance.

CLIENT

Acknowledged - corrected.

[L-08] Add a `timelock` to `restricted` functions that set critical values

It is a good practice to give time for users to react and adjust to critical changes. A `timelock` provides more guarantees and reduces the level of trust required, thus decreasing the risk for users. It also indicates that the project is legitimate. Here, no `timelock` capabilities seem to be used. We believe this impacts multiple users enough to make them want to react/be notified ahead of time.

Consider adding a `timelock` to functions like: `setWithdrawalFee`, `setLockingPeriod`, etc.

CLIENT

Acknowledged - corrected.

[I-01] Using `SafeMath` when compiler is ^0.8.0

There is no need to use `SafeMath` when the compiler is ^0.8.0 because it has built-in under/overflow checks.

[I-02] NatSpecs are incomplete

`@notice`, `@param` and `@return` fields are missing throughout all of the contracts. NatSpec documentation is essential for a better understanding of the code by developers and auditors and is strongly recommended. Please refer to the NatSpec format and follow the guidelines outlined there.

[I-03] Make use of Solidity time units

Instead of setting `rewardsDuration = 126144000` you can set it to `rewardsDuration = 4 years` as it is less confusing and more readable.

[I-04] Use custom errors instead of require statements with string error

Custom errors reduce the contract size and can provide easier integration with a protocol. Note that Custom Errors are available from compiler version `0.8.4`.

[I-05] Not used events can be removed

The following events inside `VoxToken.sol` are not used and can be removed:

`UpdateUniswapV2Router()`, `OperationsWalletUpdated()`, `TeamWalletUpdated()`

[I-06] Unclear error message

There are error messages that are not clear. Consider changing them as they might be confusing for the user.

```
    require(amount > 0, "!stake-0");
    require(_pools[msg.sender], '!pool');

    require(shares > 0, '!shares');
    require(_privatePools[msg.sender], '!private');
    require(_locked[account] >= shares, '!locked');

    require(shares > 0, '!shares');
    require(_privatePools[msg.sender], '!private');
    require(_balances[account].sub(_locked[account]) >= shares,
'!locked');
```

[I-07] CEI pattern is not followed

The `deposit` function inside `VoxStakingContract` is updating the balances after the tokens are transferred. `nonReentrant` modifier is used which makes reentrancy attacks impossible but as a best security practice it is always preferable to follow the CEI pattern.

[I-08] Variables can be turned into an immutable

`rewardsToken` and `stakingToken` variables in `VoxStakingPool.sol` can be made immutable since they are only set in the constructor and never changed after that as well as `maxWallet`, `maxTransactionAmount`, `buyTotalFees`, `buyMarketingFee`, `buyLiquidityFee`, `buyBurnFee`, `sellTotalFees`, `sellMarketingFee`, `sellLiquidityFee`, `sellBurnFee` in `VoxToken.sol`

[I-09] Most setter functions do not emit events

Examples of this are `setTreasury`, `setStakingPool` or `setWithdrawalFee` - state-changing methods should emit events so that off-chain monitoring can be implemented. Make sure to emit a proper event in each state-changing method to follow best practices. Other functions that are missing event emissions are: `enableTrading`, `removeLimits`, `disableTransferDelay`, `updateSwapTokensAtAmount`, `excludeFromMaxTransaction`, `updateSwapManager`, `updateSwapEnabled` in `VoxToken.sol`.

[I-10] Improper naming

In `VoxStakingPool.sol` we can see the function `getReward` which transfers the rewards to msg.sender. It is more appropriate to name the function something like `claimReward` because 'get' is used for getter functions and that can be confusing.

[I-11] Contracts are not inheriting their interfaces

It's a best practice for contract implementations to inherit their interface definition. Doing so would improve the contract's clarity, and force the implementation to comply with the defined interface.

[I-12] Solidity safe pragma best practices are not used

All of the contracts use floatable `^0.8.0` version. Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. Furthermore, consider using a newer version of the compiler as the latest available version is `0.8.19` which has a lot of new features and optimizations.