THE ARCADIA GROUP

Audit of The ValueGovernanceVault 2 Contract

a report of findings by

Van Cam Pham, PhD

innovative fortuna iuvat

October 20th, 2020

Table of Contents

Document Info

Client	Value DeFi
Title	Smart Contract Audit
Auditors	Van Cam Pham, PhD
Reviewed By	Joel Farris
Approved By	Rasikh Morani

Contact

For more information on this report, contact The Arcadia Media Group Inc.

Rasikh Morani
(972) 543-3886
rasikh@arcadiamgroup.com
https://t.me/thearcadiagroup

Executive Summary

A Representative Party of the Value DeFi ("ValueDeFi") engaged The Arcadia Group ("Arcadia"), a software development, research and security company, to conduct a review of the following Value Governance Vault smart contract on the Value DeFi repo at Commit #a6697d7baf7c604076d9b6eb13b2f6069b475b80.

ValueGovernanceVault.sol

Arcadia completed the review using various methods primarily consisting of dynamic and static analysis. This process included a line by line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

Conclusion

While most of the findings do not require any immediate action, some may require additional disclosure and communication to the end users for clarity, additionally code review and audits should be completed prior to launch in order to be maximally effective and to lower end-user risk.

Findings

1. Constructor input parameters

- VAULT-1
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: ValueGovernanceVault.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 118

In the ValueGovernanceVault contract, the constructor takes `_startBlock` as its input. This input parameter should be checked for its value greater than the current block number in order to avoid the situation that the contract rewarding block would be before the block at which the contract is deployed.

```
constructor (ITokenInterface _yfvToken,
       ITokenInterface _valueToken,
       ITokenInterface _vUSD,
       ITokenInterface _vETH,
       uint _valuePerBlock,
       uint _vusdPerBlock,
       uint _startBlock) public ERC20("GovVault:ValueLiquidity", "gvVALUE") {
       yfvToken = _yfvToken;
       valueToken = _valueToken;
       vUSD = _vUSD;
       vETH = _vETH;
       valuePerBlock = _valuePerBlock;
       vusdPerBlock = _vusdPerBlock;
       lastRewardBlock = _startBlock;
       governance = msg.sender;
   }
```

Action Recommended: Add a `require` statement to verify that _startBlock must be after the current block.

2. Code Readability

- VAULT-2
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: ValueGovernanceVault.sol
- Category: Informational
- Finding Type: Dynamic
- Lines: 129, 134, 139, 144, 149, 154, 159, 164, 171, 178, 183, 188, 194, 199, 206

Repeated require statement checking code should have a modifier in the contract. Specifically, `require(msg.sender == governance, "!governance");` is repeated in 15 functions.

```
function setFundCap(uint _fundCap) external {
    require(msg.sender == governance, "!governance");
    fundCap = _fundCap;
}

function setTotalDepositCap(uint _totalDepositCap) external {
    require(msg.sender == governance, "!governance");
    totalDepositCap = _totalDepositCap;
}

function setGovernance(address _governance) public {
    require(msg.sender == governance, "!governance");
    governance = _governance;
}...
```

Action Recommended: The code should have a modifier isGovernance to check whether function callers in the functions are the governance contract. This improves readability.

3. Hardcoding Token Addresses

- VAULT-3
- Severity: Medium
- Likelihood: Low
- Impact: Low

- Target: ValueGovernanceVault.sol
- Category: Low
- Finding Type: Dynamic
- Lines: 105

In the ValueGovernanceVault contract, as the token contracts for Value, vUSD, vETH are already deployed, ValueGovernanceVault the contract should hardcode these addresses of the token contracts.

Using constructor parameters to pass the token contract addresses is good for testing but downgrading code readability and explicitness.

```
constructor (ITokenInterface _yfvToken,
       ITokenInterface _valueToken,
       ITokenInterface _vUSD,
       ITokenInterface _vETH,
       uint _valuePerBlock,
       uint _vusdPerBlock,
       uint _startBlock) public ERC20("GovVault:ValueLiquidity", "gvVALUE") {
       yfvToken = _yfvToken;
       valueToken = _valueToken;
       vUSD = _vUSD;
       vETH = _vETH;
       valuePerBlock = _valuePerBlock;
       vusdPerBlock = _vusdPerBlock;
       lastRewardBlock = _startBlock;
       governance = msg.sender;
   }
```

Action Recommended: Hardcoding the token contract addresses in the contract while still allowing to change the token contract addresses in the constructor, by using a simple check for input token addresses.

```
     constructor (ITokenInterface _yfvToken,
       ITokenInterface _valueToken,
       ITokenInterface _vUSD,
       ITokenInterface _vETH,
       uint _valuePerBlock,
       uint _vusdPerBlock,
       uint _startBlock) public ERC20("GovVault:ValueLiquidity", "gvVALUE") {
       yfvToken = _yfvToken != address(0x0)?_yfvToken:0x..;
       valueToken = _valueToken != address(0x0)?_valueToken:0x..;
       vUSD = _vUSD != address(0x0)?_vUSD:0x..;
       vETH = _vETH != address(0x0)?_vETH:0x..;
       valuePerBlock = _valuePerBlock;
```

```
        vusdPerBlock = _vusdPerBlock;
        lastRewardBlock = _startBlock;
        governance = msg.sender;
    }
```

4. Users would not receive bonus Value reward

- VAULT-4
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: ValueGovernanceVault.sol
- Category: Medium
- Finding Type: Dynamic
- Lines: 393-402

In the ValueLiquidityToken contract, any user can deposit Value or YFV to receive a corresponding amount of gvValue token (called shares). The user can stake the shares to receive rewards paid in Value, vUSD, and vETH (newly minted). The user can decide to lock the shares for a certain amount of time (from 7 days to 150 days) in order to receive bonus rewards paid in Value for the locked shares. The user should receive bonus rewards regardless of the time the user makes the transaction to withdraw the rewards. The function `_getReward`, however, does not take bonus rewards into account if the user withdraws rewards after the expected unlock time. Specifically, line 395, the lockedAmount share of the user is reset to 0 without calculating the bonus rewards the user should receive. This is not the behavior that the user expects as the bonus rewards should be sent to the user regardless of when the user makes transactions to withdraw rewards.

```
    function _getReward() internal {
        UserInfo storage user = userInfo[msg.sender];
        uint _pendingValue = user.amount.mul(accValuePerShare).div(1e12).sub(user.valueRewardDebt);
        if (_pendingValue > 0) {
            if (user.lockedAmount > 0) {
                if (user.unlockedTime < block.timestamp) {
                    user.lockedAmount = 0;
                } else {
                    uint _bonus =
_pendingValue.mul(user.lockedAmount.mul(user.boostedExtra).div(1e12)).div(user.amount);
                    uint _ceilingBonus = _pendingValue.mul(33).div(100); // 33%
                    if (_bonus > _ceilingBonus) _bonus = _ceilingBonus; // Additional check to avoid
insanely high bonus!
                    _pendingValue = _pendingValue.add(_bonus);
                }
            }
            user.accumulatedStakingPower = user.accumulatedStakingPower.add(_pendingValue);
            uint actualPaid = _pendingValue.mul(99).div(100); // 99%
            uint commission = _pendingValue - actualPaid; // 1%
            safeValueMint(msg.sender, actualPaid);
            address _referrer = address(0);
            if (rewardReferral != address(0)) {
                _referrer = IYFVReferral(rewardReferral).getReferrer(msg.sender);
            }
            if (_referrer != address(0)) { // send commission to referrer
                safeValueMint(_referrer, commission);
                CommissionPaid(_referrer, commission);
            } else { // send commission to valueInsuranceFund
                safeValueMint(valueInsuranceFund, commission);
                CommissionPaid(valueInsuranceFund, commission);
            }
        }
        uint _pendingVusd = user.amount.mul(accVusdPerShare).div(1e12).sub(user.vusdRewardDebt);
        if (_pendingVusd > 0) {
            safeVusdMint(msg.sender, _pendingVusd);
        }
    }
```

Action Recommended: Function `_getReward` should take bonus rewards into account even if users withdraw after unlock time. Here's a possible solution:

```
if (user.unlockedTime < block.timestamp) {
   uint lockedAmount = user.lockedAmount;
   user.lockedAmount = 0;
   if (user.previousRewardTimestamp < user.unlockedTime) {
      uint timeFromLastRewardTimeTilNow = block.timestamp.sub(user.previousRewardTimestamp);
      uint previousRewardTimestamp = user.previousRewardTimestamp;
      if (previousRewardTimestamp == 0) previousRewardTimestamp = user.unlockedTime.sub(user.lockedDays
* 86400);
      uint timeFromLastRewardTimeTilUnlockTime = user.unlockedTime.sub(previousRewardTimestamp);
      uint pendingValueFromLastRewardTimeTilUnlockTime =
_pendingValue.mul(timeFromLastRewardTimeTilUnlockTime).div(timeFromLastRewardTimeTilNow);
      uint _bonus =
pendingValueFromLastRewardTimeTilUnlockTime.mul(lockedAmount.mul(user.boostedExtra).div(1e12)).div(user
.amount);
      uint _ceilingBonus = pendingValueFromLastRewardTimeTilUnlockTime.mul(33).div(100); // 33%
      if (_bonus > _ceilingBonus) _bonus = _ceilingBonus;
         _pendingValue = _pendingValue.add(_bonus);
        }
} else {
   uint _bonus =
_pendingValue.mul(user.lockedAmount.mul(user.boostedExtra).div(1e12)).div(user.amount);
   uint _ceilingBonus = _pendingValue.mul(33).div(100); // 33%
   if (_bonus > _ceilingBonus) _bonus = _ceilingBonus; // Additional check to avoid insanely high
bonus!
         _pendingValue = _pendingValue.add(_bonus);
    }



///update user.previousRewardTimestamp at the end of _getReward function
```

As a summary of the solution, the code computes the pending rewards the user can receive since the last time _getReward function is called by the user til the expected unlocked time of the locked share. Based on the computed pending value, bonus will be calculated accordingly. `previousRewardTimestamp` is a newly introduced field in User struct in order to record the last time `_getReward` is called. `previousRewardTimestamp` should be updated every time `_getReward` is called.

5. Function `pendingValue` returns false results if called after unlockedTime

- VAULT-5
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: ValueGovernanceVault.sol
- Category: Bonus rewards
- Finding Type: Dynamic
- Lines: 405-430

In the ValueLiquidityToken contract, any user can depositdeposit Value or YFV to receive a corresponding amount of gvValue token (called shares). The user can stake the shares to receive rewards paid in Value, vUSD, and vETH (newly minted). The user can decide to lock the shares for a certain amount of time (from 7 days to 150 days) in order to receive bonus rewards paid in Value for the locked shares. The user should be able to read pending rewards plus bonus rewards regardless of the time the user makes the call to compute the pending Value rewards. The function `pendingValue`, however, does not take bonus rewards into account if the function is called after the expected unlock time. Specifically, line 422, the function does not calculate the bonus rewards the user should receive.

```
function pendingValue(address _account)
       public
       view
       returns (uint256 _pending)
    {
       UserInfo storage user = userInfo[_account];
```

```
        uint256 _accValuePerShare = accValuePerShare;
        uint256 lpSupply = balanceOf(address(this));
        if (block.number > lastRewardBlock && lpSupply != 0) {
            uint256 numBlocks = block.number.sub(lastRewardBlock);
            _accValuePerShare = accValuePerShare.add(
                numBlocks.mul(valuePerBlock).mul(1e12).div(lpSupply)
            );
        }
        _pending = user.amount.mul(_accValuePerShare).div(1e12).sub(
            user.valueRewardDebt
        );
        if (user.lockedAmount > 0 && user.unlockedTime >= block.timestamp) {
            uint256 _bonus = _pending
                .mul(user.lockedAmount.mul(user.boostedExtra).div(1e12))
                .div(user.amount);
            uint256 _ceilingBonus = _pending.mul(33).div(100); // 33%
            if (_bonus > _ceilingBonus) _bonus = _ceilingBonus; // Additional check to avoid insanely
high bonus!
            _pending = _pending.add(_bonus);
        }
    }
```

Action Recommended: `pendingValue` should take bonus rewards into account even if users call the function after unlock time. See Issue VAULT-4 for a possible solution.

6. Function `withdrawAll` will not send bonus rewards if called after unlocked time.

- VAULT-6
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: ValueGovernanceVault.sol
- Category: Bonus rewards
- Finding Type: Dynamic
- Lines: 405-430

In the ValueLiquidityToken contract, the user should be able to withdraw pending rewards plus bonus rewards regardless of the time the user makes transactions to execute function `withdrawAll`. The function `withdrawAll`, however, resets lockedAmount to 0 if the function is called after the expected unlock time.

```
function withdrawAll(uint8 _flag) public discountCHI(_flag) {
       UserInfo storage user = userInfo[msg.sender];
       uint _amount = user.amount;
       if (user.lockedAmount > 0) {
           if (user.unlockedTime < block.timestamp) {
               user.lockedAmount = 0;
           } else {
               _amount = user.amount.sub(user.lockedAmount);
           }
       }
       unstake(_amount, 0x0);
       withdraw(balanceOf(msg.sender), 0x0);
   }
```

Action Recommended: See Issue VAULT-4 for a possible solution. Once Vault-4's solution is implemented, `withdrawAll` should let unstake reset `lockedAmount` as the function `unstake` already implements the code to reset lockedAmount.

```
function withdrawAll(uint8 _flag) public discountCHI(_flag) {
        UserInfo storage user = userInfo[msg.sender];
        uint _amount = user.amount;
        if (user.lockedAmount > 0) {
            if (user.unlockedTime >= block.timestamp) {
                _amount = user.amount.sub(user.lockedAmount);
            }
        }
        unstake(_amount, 0x0);
        withdraw(balanceOf(msg.sender), 0x0);
    }
```

Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or for damages resulting from the use of the provided information. Additionally Arcadia would like to emphasize that use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period of time.