SMART CONTRACT AUDIT REPORT

for

Valas Finance

Prepared By: Xiaomi Huang

PeckShield

Jun 10, 2022

Document Properties

Client	Valas Finance
Title	Smart Contract Audit Report
Target	Valas Finance
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jun 10, 2022	Shulin Bie	Final Release
1.0-rc	May 25, 2022	Shulin Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Valas Finance` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Valas Finance

`Valas Finance` is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., `AAVE`. The protocol extends the original version with new features for staking-based incentivization and fee distribution. With that, the liquidity providers earn `50%` of the protocol revenue, and the `VALAS` holders that stake or lock their tokens earn the other `50%`.

Table 1.1: Basic Information of Valas Finance

Item	Description
Target	Valas Finance
Website	https://valasfinance.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun 10, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/valas-finance/valas-fold.git (f6e0085)
- https://github.com/valas-finance/valas-protocol.git (b9085f0)

And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/valas-finance/valas-fold.git (54c4db8)
- https://github.com/valas-finance/valas-protocol.git (4ff1094)

1.2 About PeckShield

PeckShield Inc. [13] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Valas Finance` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	2	■ ■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Valas Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-002	Medium	Revisited Reentrancy Protection In Current Implementation	Time and State	Confirmed
PVE-003	High	Possible Price Manipulation In ProtocolOwnedDEXLiquidity	Time and State	Fixed
PVE-004	Informational	Immutable States If Only Set At Constructor()	Coding Practices	Fixed
PVE-005	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-006	Informational	Suggested Event Generation For Key Operations	Coding Practices	Fixed
PVE-007	Low	Fork-Compliant Domain Separator In AToken	Business Logic	Confirmed
PVE-008	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Valas Finance` protocol, the `LendingPool` contract is designed to be the main entry for interaction with borrowing/lending users. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and mints the corresponding `valToken` to represent the depositor's share in the lending pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    /**
    * @dev Deposits an `amount` of underlying asset into the reserve, receiving in return overlying aTokens.
    * - E.g. User deposits 100 USDC and gets in return 100 aUSDC
    * @param asset The address of the underlying asset to deposit
    * @param amount The amount to be deposited
    * @param onBehalfOf The address that will receive the aTokens, same as msg.sender if the user
    *   wants to receive them on his own wallet, or a different address if the beneficiary of aTokens
    *   is a different wallet
    * @param referralCode Code used to register the integrator originating the operation, for potential rewards.
    *   0 if the action is executed directly by the user, without any middle-man
    **/
  function deposit(
```

```
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external override whenNotPaused {
    DataTypes.ReserveData storage reserve = _reserves[asset];

    ValidationLogic.validateDeposit(reserve, amount);

    address aToken = reserve.aTokenAddress;

    reserve.updateState();
    reserve.updateInterestRates(asset, aToken, amount, 0);

    IERC20(asset).safeTransferFrom(msg.sender, aToken, amount);

    bool isFirstDeposit = IAToken(aToken).mint(onBehalfOf, amount, reserve.
        liquidityIndex);

    if (isFirstDeposit) {
      _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
      emit ReserveUsedAsCollateralEnabled(asset, onBehalfOf);
    }

    emit Deposit(asset, msg.sender, onBehalfOf, amount, referralCode);
  }
```

Listing 3.1: `LendingPool::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Valas Finance`. In `Valas Finance` protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Note that other routines, i.e., `MasterChef::deposit()` and `MultiFeeDistribution::stake()`, share the similar issue.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been confirmed by the team. There is no need to support deflationary/re-basing tokens.

3.2 Revisited Reentrancy Protection In Current Implementation

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact:High
- Target: Multiple Contracts
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [15] exploit, and the recent Uniswap/Lendf.Me hack [14].

In the MasterChef contract, we notice the deposit() routine has potential reentrancy risk. To elaborate, we show below the related code snippet of the MasterChef::deposit() routine. In the deposit() routine, we notice IERC20(_token).safeTransferFrom(address(msg.sender), address(this), _amount) (lines 247 - 251) will be called to transfer the underlying assets into the MasterChef contract. If the _token faithfully implements the ERC777-like standard, then the deposit() routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when transfer() or transferFrom() actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering tokensToSend() and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining `GasTokens`).

In our case, the above hook can be planted in `IERC20(_token).safeTransferFrom(address(msg.sender), address(this), _amount)` (lines 247 - 251) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `user.rewardDebt` intact (used for the calculation of pending rewards at line 242). With a lower `user.rewardDebt`, the re-entered `deposit()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains.

```
233     function deposit(address _token, uint256 _amount) external {
234         PoolInfo storage pool = poolInfo[_token];
235         require(pool.lastRewardTime > 0);
236         _updateEmissions();
237         _updatePool(_token, totalAllocPoint);
238         UserInfo storage user = userInfo[_token][msg.sender];
239         uint256 userAmount = user.amount;
240         uint256 accRewardPerShare = pool.accRewardPerShare;
241         if (userAmount > 0) {
242             uint256 pending = userAmount.mul(accRewardPerShare).div(1e12).sub(user.
                    rewardDebt);
243             if (pending > 0) {
244                 userBaseClaimable[msg.sender] = userBaseClaimable[msg.sender].add(
                        pending);
245             }
246         }
247         IERC20(_token).safeTransferFrom(
248             address(msg.sender),
249             address(this),
250             _amount
251         );
252         userAmount = userAmount.add(_amount);
253         user.amount = userAmount;
254         user.rewardDebt = userAmount.mul(accRewardPerShare).div(1e12);
255         ...
256     }
```

Listing 3.2: `MasterChef::deposit()`

We observe the current implementation of the `MasterChef` and `MultiFeeDistribution` contracts haven't considered reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been confirmed by the team. The protocol will not support `ERC777`-like token.

3.3 Possible Price Manipulation In ProtocolOwnedDEXLiquidity

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `ProtocolOwnedDEXLiquidity`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

While examining the `ProtocolOwnedDEXLiquidity` contract, we notice there is a price manipulation vulnerability that can be exploited by the malicious actor to steal the assets in the contract. To elaborate, we show below the related code snippet of the `ProtocolOwnedDEXLiquidity` contract. By design, the `_buy()` routine is used to buy `vWBNB` (i.e., the lp token of the `Valas Finance`'s `WBNB` lending pool) with `lpToken` (i.e., the lp token of the `PancakeSwap`'s `VALAS-WBNB` pool). In the `_buy()` routine, the `lpTokensPerOneBNB()` routine is executed (line 125) to calculate the `WBNB`'s price relative to `lpToken`. However, in the `lpTokensPerOneBNB()` routine, we notice the `WBNB`'s price is calculated according to the current state of the `PancakeSwap`'s `VALAS-WBNB` pool, which may have been price-manipulated. The malicious actor can buy a huge amount of `vWBNB` with tiny `lpToken` via making the `WBNB` in the pool exceptionally cheap.

```
113     function lpTokensPerOneBNB() public view returns (uint256) {
114         uint totalSupply = lpToken.totalSupply();
115         (,uint reserve1,) = lpToken.getReserves();
116         return totalSupply.mul(1e18).mul(45).div(reserve1).div(100);
117     }
118
119     function _buy(uint _amount, uint _cooldownTime) internal {
120         UserRecord storage u = userData[msg.sender];
121
122         require(_amount >= minBuyAmount, "Below min buy amount");
123         require(block.timestamp >= u.nextClaimTime, "Claimed too recently");
124
125         uint lpAmount = _amount.mul(lpTokensPerOneBNB()).div(1e18);
126         lpToken.transferFrom(msg.sender, address(this), lpAmount);
127         vWBNB.transfer(msg.sender, _amount);
128         vWBNB.transfer(address(treasury), _amount);
129
130         u.nextClaimTime = block.timestamp.add(_cooldownTime);
131         u.claimCount = u.claimCount.add(1);
132         u.totalBoughtBNB = u.totalBoughtBNB.add(_amount);
133         totalSoldBNB = totalSoldBNB.add(_amount);
134
135         emit SoldBNB(msg.sender, _amount);
136     }
```

Listing 3.3: `ProtocolOwnedDEXLiquidity::lpTokensPerOneBNB()`

Recommendation Ensure the safety of the price oracle used in above-mentioned routine.

Status The issue has been addressed by the following commits: `dbe2fac` and `4ff1094`.

3.4 Immutable States If Only Set At Constructor()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `Valas Finance` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
contract ChefIncentivesController is Ownable {
    ...

    address public poolConfigurator;

    IMultiFeeDistribution public rewardMinter;
}
```

Listing 3.4: `ChefIncentivesController`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been addressed by the following commit: `0ddf3d2`.

3.5 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ProtocolOwnedDEXLiquidity/Fold`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }
    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.5: `ZRX.sol`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` and `approve()` as well, i.e., `safeTransferFrom()` and `safeApprove()`.

In the following, we show the `ProtocolOwnedDEXLiquidity::_buy()` routine. If the `USDT`-like token is supported as `vWBNB`, the unsafe version of `vWBNB.transfer(msg.sender, _amount)` (line 127) may revert as there is no return value in the `USDT`-like token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace `vWBNB.transfer(msg.sender, _amount)` (line 127) with `safeTransfer()`.

```
119     function _buy(uint _amount, uint _cooldownTime) internal {
120         UserRecord storage u = userData[msg.sender];
121 
122         require(_amount >= minBuyAmount, "Below min buy amount");
123         require(block.timestamp >= u.nextClaimTime, "Claimed too recently");
124 
125         uint lpAmount = _amount.mul(lpTokensPerOneBNB()).div(1e18);
126         lpToken.transferFrom(msg.sender, address(this), lpAmount);
127         vWBNB.transfer(msg.sender, _amount);
128         vWBNB.transfer(address(treasury), _amount);
129 
130         u.nextClaimTime = block.timestamp.add(_cooldownTime);
131         u.claimCount = u.claimCount.add(1);
132         u.totalBoughtBNB = u.totalBoughtBNB.add(_amount);
133         totalSoldBNB = totalSoldBNB.add(_amount);
134 
135         emit SoldBNB(msg.sender, _amount);
136     }
137 }
```

Listing 3.6: `ProtocolOwnedDEXLiquidity::_buy()`

Note another routine, i.e., `Fold::_approve()`, can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `approve()`, `transfer()`, and `transferFrom()`.

Status The issue has been addressed by the following commits: `93c9ab3` and `54c4db8`.

3.6 Suggested Event Generation For Key Operations

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [4]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setMinters(address[] memory _minters) external onlyOwner {
        require(!mintersAreSet);
        for (uint i; i < _minters.length; i++) {
            minters[_minters[i]] = true;
        }
        mintersAreSet = true;
    }

    function setIncentivesController(IChefIncentivesController _controller) external
        onlyOwner {
        incentivesController = _controller;
    }

    // Add a new reward token to be distributed to stakers
    function addReward(address _rewardsToken) external override onlyOwner {
        require(rewardData[_rewardsToken].lastUpdateTime == 0);
        rewardTokens.push(_rewardsToken);
        rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
        rewardData[_rewardsToken].periodFinish = block.timestamp;
    }
```

Listing 3.7: MultiFeeDistribution

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Note that other routines, i.e., `ProtocolOwnedDEXLiquidity::setParams()`, `TokenVesting::start()`, and `ValasToken::setMinter()/setTreasury()`, can be similarly improved.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: `986cf60`.

3.7 Fork-Compliant Domain Separator In AToken

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `AToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `AToken` token contract strictly follows the widely-accepted ERC20 specification. In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `initialize()` function (lines 81-89).

```
64    function initialize(
65        ILendingPool pool,
66        address treasury,
67        address underlyingAsset,
68        IAaveIncentivesController incentivesController,
69        uint8 aTokenDecimals,
70        string calldata aTokenName,
71        string calldata aTokenSymbol,
72        bytes calldata params
73    ) external override initializer {
74        uint256 chainId;
75
76        //solium-disable-next-line
77        assembly {
78        chainId := chainid()
79        }
80
81        DOMAIN_SEPARATOR = keccak256(
82        abi.encode(
83            EIP712_DOMAIN,
84            keccak256(bytes(aTokenName)),
```

```
            keccak256(EIP712_REVISION),
            chainId,
            address(this)
        )
        );

        ...
    }
```

Listing 3.8: `AToken::initialize()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        require(owner != address(0), 'INVALID_OWNER');
        //solium-disable-next-line
        require(block.timestamp <= deadline, 'INVALID_EXPIRATION');
        uint256 currentValidNonce = _nonces[owner];
        bytes32 digest =
        keccak256(
            abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
                currentValidNonce, deadline))
            )
        );
        require(owner == ecrecover(digest, v, r, s), 'INVALID_SIGNATURE');
        _nonces[owner] = currentValidNonce.add(1);
        _approve(owner, spender, value);
    }
```

Listing 3.9: `AToken::permit()`

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` inside the `permit()` function.

Status The issue has been confirmed by the team.

3.8 Trust Issue Of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `Valas Finance` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring the price oracle). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    /// @notice External function called by the Aave governance to set or replace
        sources of assets
    /// @param assets The addresses of the assets
    /// @param sources The address of the source of each asset
    function setAssetSources(address[] calldata assets, address[] calldata sources)
        external
        onlyOwner
    {
        _setAssetsSources(assets, sources);
    }
```

Listing 3.10: `AaveOracle::setAssetSources()`

```
    function mint(address _to, uint256 _value) external returns (bool) {
        if (msg.sender != minter) {
            require(msg.sender == treasury);
            treasuryMintedTokens = treasuryMintedTokens.add(_value);
            require(treasuryMintedTokens <= maxTreasuryMintable);
        }
        balanceOf[_to] = balanceOf[_to].add(_value);
        totalSupply = totalSupply.add(_value);
        require(maxTotalSupply >= totalSupply);
        emit Transfer(address(0), _to, _value);
        return true;
    }

    function setTreasury(address _treasury) external {
        require(msg.sender == treasury);
        treasury = _treasury;
    }
```

Listing 3.11: `ValasToken::mint()&&setTreasury()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure.

Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team. The team introduces `multi-sig` mechanism to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the Valas Finance design and implementation. Valas Finance is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., AAVE. The protocol extends the original version with new features for staking-based incentivization and fee distribution. With that, the liquidity providers earn 50% of the protocol revenue, and the VALAS holders that stake or lock their tokens earn the other 50%. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[9] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[11] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[12] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[13] PeckShield. PeckShield Inc. https://www.peckshield.com.

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.