SMART CONTRACT AUDIT REPORT

for

TESTA FINANCE

Prepared By: Shuxiao Wang

PeckShield

January 29, 2021

Document Properties

Client	Testa Finance
Title	Smart Contract Audit Report
Target	Testa Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 29, 2021	Xuxian Jiang	Final Release
1.0-rc	January 28, 2021	Xuxian Jiang	Release Candidate
0.3	January 19, 2021	Xuxian Jiang	Additional Findings #2
0.2	January 16, 2021	Xuxian Jiang	Additional Findings #1
0.1	January 15, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction 4
1.1 About Testa Finance 4
1.2 About PeckShield 5
1.3 Methodology 5
1.4 Disclaimer 7

2 Findings 9
2.1 Summary 9
2.2 Key Findings 10

3 Detailed Results 11
3.1 Timely Conversion Of Staking Rewards To lpTokens 11
3.2 Improved PlanetFarm::payToEntry() Logic 12
3.3 Business Logic Design In withdraw() of PlanetFarm And SpaceFarm 13
3.4 Reward Loss With Zero-Withdrawal In harvestAndWithdraw() 15
3.5 Potential Sandwich Attacks To Maximize getTestaReward()/Minimize getTestaFee() 16
3.6 Improved Sanity Checks For System/Function Parameters 18
3.7 Duplicate Pool Detection and Prevention 19
3.8 Recommended Explicit Pool Validity Checks 21
3.9 Suggested Adherence of Checks-Effects-Interactions 23
3.10 Timely massUpdatePools During Pool Weight Changes 24
3.11 Trust Issue of Admin Keys 26
3.12 Improved Precision By Multiplication And Division Reordering 27

4 Conclusion 29

References 30

1 Introduction

Given the opportunity to review the design document and related smart contract source code of Testa Finance, we in the report outline 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 Testa Finance

`Testa Finance` is a platform originally built on the Ethereum blockchain utilizing Ethereum smart contract for staking, unstaking or yield farming the digital assets. Users can stake `Testa` (ERC-20 token) to receive `jTesta` (ERC-20 token) and earn the profitable airdrops from the `DAO` consensus network. Both `Testa` and `jTesta` holders can earn great benefits and perks from the ecosystem including `Testamex`, the hybrid Cryptocurrency exchange. The initial implementation is heavily influenced by the `Alpha Homora V1` protocol with its own customization and extensions to meet unique protocol needs.

The basic information of the Testa Finance protocol is as follows:

Table 1.1: Basic Information of Testa Finance

Item	Description
Issuer	Testa Finance
Website	https://testa.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 29, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/starcard-org/yield-delegation/tree/peckshield-audit (e2b9200)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/starcard-org/yield-delegation/tree/peckshield-audit (6d501e4)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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.

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

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.

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) [13], 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 design and implementation of the Testa Finance protocol. 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	7	■■■■■■■
Informational	3	■■■
Total	12	

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 2 medium-severity vulnerabilities, 7 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings of Testa Finance

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Conversion Of Staking Rewards To lpTokens	Business Logic	Confirmed
PVE-002	Low	Improved PlanetFarm::payToEntry() Logic	Business Logic	Fixed
PVE-003	Informational	Business Logic Design In withdraw() of PlanetFarm And SpaceFarm	Business Logic	Confirmed
PVE-004	Medium	Reward Loss With Zero-Withdrawal In harvestAndWithdraw()	Business Logic	Fixed
PVE-005	Low	Potential Sandwich Attacks To Maximize getTestaReward()/Minimize getTestaFee()	Time And State	Confirmed
PVE-006	Informational	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Confirmed
PVE-007	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-008	Informational	Recommended Explicit Pool Validity Checks	Security Features	Fixed
PVE-009	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-010	Low	Timely massUpdatePools During Pool Weight Changes	Business Logics	Fixed
PVE-011	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-012	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Timely Conversion Of Staking Rewards To lpTokens

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

Description

The Testa Finance protocol is heavily influenced by `Alpha Homora` and shares the same architecture design to allow users to take leveraged yield farming. At the core, the protocol permits users to take a loan position to maximize yielding potentials. In the meantime, supplying users can get rewards from accumulated interest and fees.

To elaborate, we show below the code snippet of the `destroy()` routine in the `UniswapOrbit` contract. The execution logic is rather straightforward: it firstly transfers the LP tokens from the position to the liquidation strategy, and then returns all available `ETHs` back to the operator, i.e., `Station`.

```
    /// @dev Liquidate the given position by converting it to ETH and return back to
        caller.
    /// @param id The position ID to perform liquidation
    function destroy(uint256 id, address user) external onlyOperator nonReentrant {
        require(IERC20(tokenPermission.getTerminateTokenPermission()).balanceOf(user) >=
             tokenPermission.getTerminateTokenAmount(), "insufficient token hoding");
        // 1. Convert the position back to LP tokens and use liquidate strategy.
        _removeShare(id);
        lpToken.transfer(address(liqStrat), lpToken.balanceOf(address(this)));
        liqStrat.operate(address(0), 0, abi.encode(fToken, 0));
        // 2. Return all available ETH back to the operator.
        uint256 wad = address(this).balance;
        SafeToken.safeTransferETH(msg.sender, wad);
        emit Destroy(id, wad);
    }
```

```
        //translate vault shares into delegating vault shares
        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _new_shares;
        } else {
            shares = (_new_shares.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
        rewardDebt[msg.sender] = balanceOf(msg.sender).mul(accRallyPerShare).div(1e12);
    }
```

Listing 3.1: UniswapOrbit::destroy()

However, our analysis shows that when the given position is destroyed, it may still accumulate staking rewards entitled to the destroyed position. Note that current implementation does not return the accumulated staking rewards back to the user. In a similar vein, when a position is being launched (via the `launch()` routine, the handling `orbit` does not timely convert staking rewards to `lpTokens` for existing users. As a result, the staking rewards may be shared with the new user.

Recommendation Timely collect and convert staking rewards to respective `lpTokens` to ensure fair reward distribution.

Status The issue has been confirmed. However, following the same design choice from `Alpha Homora V1`, the team decides to leave as is.

3.2 Improved PlanetFarm::payToEntry() Logic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PlanetFarm`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In Testa Finance, the `PlanetFarm` contract is a farming pool that utilizes collateral token generated from using the protocol. By design, if any user wants to participate in the farming pool, the user needs to have at least 777 `TESTA` available to activate access.

To elaborate, we show below the related `payToEntry()` logic that allows users to pay the entrance fee of at least 777 `TESTA`. It comes to our attention the enforcement of `config.getPayAmount()== amount` (line 1010) stipulates the exact amount needs to be paid.

```
1007    function payToEntry(uint256 amount) external {
```

```
        (bool success, ) = testa.call(abi.encodeWithSignature("transferFrom(address,address,uint256)", msg.sender, config.getCompany(), amount));
        require((amount > 0) && success);
        if(config.getPayAmount() == amount){
            users[msg.sender] = true;
        }
    }
```

Listing 3.2: PlanetFarm::payToEntry()

The payment requirement with the exact amount causes unnecessary friction. And we suggest to make it consistent with the design document in allowing anyone to transact with the protocol if the user pays at least 777 `TESTA`

Recommendation Revise the current logic to allow any one who paid at least the specified amount to participate in the farming pool. An example revision is shown as follows:

```
    function payToEntry(uint256 amount) external {
        (bool success, ) = testa.call(abi.encodeWithSignature("transferFrom(address,address,uint256)", msg.sender, config.getCompany(), amount));
        require((amount > 0) && success);
        if(config.getPayAmount() <= amount){
            users[msg.sender] = true;
        }
    }
```

Listing 3.3: PlanetFarm::payToEntry()

Status The issue has been fixed by the following commit: `6d501e4`.

3.3 Business Logic Design In withdraw() of PlanetFarm And SpaceFarm

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PlanetFarm`, `SpaceFarm`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In Section 3.2, the `PlanetFarm` contract is a farming pool that utilizes collateral token generated from using the protocol. Users can stake in required tokens to collect rewards.

There are two main functions participating users can interact, i.e., `deposit()` and `withdraw()`. The first function directly moves the depositing assets from the staking user to the protocol while the second one withdraws the staked funds back. In the following, we show the implementation of these two routines from the `PlanetFarm` contract.

```
    // Deposit LP tokens to TestaFarm for Testa allocation.
    function deposit(uint256 amount) public checkPayToEntry {
        require(amount > 0 , "invalid amount");
        UserInfo storage user = userInfo[msg.sender];
        updateUserReward(msg.sender);
        lpToken.safeTransferFrom(address(msg.sender), address(this), amount);
        jETHToken.safeTransferFrom(address(msg.sender), address(this), amount);
        user.amount = user.amount.add(amount);

        totalStake = totalStake.add(amount);
        emit Deposit(msg.sender, amount);
    }

    // Withdraw LP tokens from TestaFarm.
    function withdraw(uint256 _amount) public checkPayToEntry {
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= _amount, "Not enought token to withdraw");
        if(_amount > 0) {
            uint256 rewardAmount = getUserReward(msg.sender);
            removeReward(msg.sender, rewardAmount);
            SafeToken.safeTransferETH(config.getCompany(), rewardAmount);

            user.amount = user.amount.sub(_amount);
            lpToken.safeTransfer(address(msg.sender), _amount);
            jETHToken.safeTransfer(address(msg.sender), _amount);
            totalStake = totalStake.sub(_amount);
        }
        emit Withdraw(msg.sender, _amount);
    }
```

Listing 3.4: PlanetFarm::withdraw()

Our analysis shows that the `withdraw()` logic indeed properly returns the staked funds back to participating users. However, the rewards possibly accumulated from the staked funds are transferred to `config.getCompany()`, not the user. It is suggested to return the rewards back to the user as well. A similar issue is also present in the `SpaceFarm` contract.

Recommendation Properly return the collected rewards to the user, instead of the protocol-specified `config.getCompany()`.

Status This is a design choice. The team has confirmed that the users need to explicitly collect rewards via `harvest()` and/or `harvestAndWithdraw()`, not via `withdraw()`.

3.4 Reward Loss With Zero-Withdrawal In harvestAndWithdraw()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: PlanetFarm
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.3, participating users in the PlanetFarm pool are supposed to claim their rewards via harvest()/harvestAndWithdraw(), not via withdraw(). In this section, we further examine the harvestAndWithdraw() routine.

To elaborate, we show below the implementation of harvestAndWithdraw(). This routine implements a rather straightforward logic in firstly validating the given _amount for withdrawal, next computing the accumulated rewardAmount, and finally transferring the proper amount back to users.

```
    function harvestAndWithdraw(uint256 _amount) public nonReentrant checkPayToEntry {
        UserInfo storage user = userInfo[msg.sender];
        uint256 lpSupply = lpToken.balanceOf(address(this));

        ( , int maxProgressive) = config.getProgressive();
        require(getBlockPass() <= config.getActivateAtBlock());
        require((progressive == maxProgressive) && (lpSupply != 0), "Must have lpSupply
            and reach maxProgressive to harvest");
        require(user.amount >= _amount, "No lpToken cannot withdraw");

        uint256 rewardAmount = getUserReward(msg.sender);
        uint256 _harvestFee = config.getTestaFee(rewardAmount);

        require(IERC20(testa).balanceOf(address(msg.sender)) > _harvestFee, "Must have
            enought testa before harvest");
        (bool success, ) = testa.call(abi.encodeWithSignature("transferFrom(address,
            address,uint256)", msg.sender, config.getCompany(), _harvestFee));
        require(success);
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            removeReward(msg.sender, rewardAmount);

            lpToken.safeTransfer(address(msg.sender), _amount);
            jETHToken.safeTransfer(address(msg.sender), _amount);
            totalStake = totalStake.sub(_amount);
            SafeToken.safeTransferETH(msg.sender, rewardAmount);
        }
        emit HarvestAndWithdraw(msg.sender, _amount);
```

```
1040     }
```

Listing 3.5: PlanetFarm::harvestAndWithdraw()

Note that for a user to collect the rewards, the user is supposed to certain `_harvestFee` to the configured destination, i.e., `config.getCompany()` (line 1028). However, if the given `amount=0`, the user pays the `_harvestFee`, but does not get the `rewardAmount`. Next time, the user needs to pay the `_harvestFee` again! The logic is incorrect and needs to be revised to always transfer the `rewardAmount` no matter whatever the given `amount` is.

Recommendation Correct the logic in always rewarding the user in the `harvestAndWithdraw()` routine..

Status The issue has been fixed by the following commit: `6d501e4`.

3.5 Potential Sandwich Attacks To Maximize getTestaReward()/Minimize getTestaFee()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TestaFarm`, `SpaceFarm`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

In order to encourage participation, the protocol is designed to reward users who periodically check in to `activate()` or progress the `TestaFarm` pool. The reasoning is the protocol needs to measure the latest liquidity every 5000 Ethereum blocks. In particular, if the latest liquidity is higher than the last checkpoint, the internal indicator will move forward 1 step; if the liquidity is lower than previous check point, indicator will move backward 1 step. On each activation, available `ETH` rewards are proportionally allocated for the users based on their deposits in the pool. And the user who successfully calls the `activate()` function first in each cycle will receive 10 `USDT` worth of `TESTA` in return as a reward.

To elaborate, we show below the `firstActivate()` routine from the `TestaFarm` contract. By calling this routine, the pool can compute current liquidity (line 1067) and reward the caller with `TESTA` reward.

```
1062     function firstActivate() public onlyEOA nonReentrant {
1063         require(IERC20(jTesta).balanceOf(msg.sender) >= config.getJTestaAmount(), "
             Insufficient jTesta amount");
1064         require(initStartBlock == startBlock);
```

```
        require(block.number >= initStartBlock, "Cannot activate until the specific block time arrive");

        currentLiquidity = config.getLiquidity();
        startBlock = block.number;
        startLiquidity = currentLiquidity;
        // send Testa to user who press activate button
        safeTestaTransfer(msg.sender, config.getTestaReward());
    }
```

Listing 3.6: TestaFarm:: firstActivate ()

```
    /// @dev Return the amount of Testa wei rewarded if we are activate the progress function.
    function getTestaReward() public view override returns (uint256) {
        ( uint112 _reserve0, uint112 _reserve1, ) = pair.getReserves();
        uint256 reserve = uint256(_reserve0).mul(1e18).div(uint256(_reserve1));
        uint256 ethPerDollar = uint256(getLatestPrice()).mul(1e10); // 1e8
        uint256 testaPerDollar = ethPerDollar.mul(1e18).div(reserve);

        uint256 _activateReward = activateReward.mul(1e18);
        uint256 testaAmount = _activateReward.mul(1e18).div(testaPerDollar);
        return testaAmount;
    }
```

Listing 3.7: PlanetFarmConfig::getTestaReward()

We notice the reward amount is computed in a way that depends on current pool reserves on `Uniswap`. As a result, the current swap rate can be manipulated by powerful miners attacks. (Note that a flashloan attack may not be possible as the `activate()` is restricted to EOA accounts only.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user (or a larger reward to the user who calls `activate()` in our case). A similar issue is also present in another `getTestaFee()` routine.

As a mitigation, we may consider specifying the restriction on possible surge on `totalSupply` or imposing certain lock time for the users to claim the rewards. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of liquidity providers.

Status This issue has been confirmed. However, as mentioned earlier, the front-running attack is inherent in current DEXes and there is still a need to search for more effective countermeasures.

3.6 Improved Sanity Checks For System/Function Parameters

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Testa Finance protocol is no exception. Specifically, if we examine the `SpaceFarm` contract, it has defined a number of system-wide risk parameters: `testaPerBlock`, `maxProgressive`, `minProgressive`, and `numberOfBlock`. In the following, we show corresponding routines that allow for their changes.

```
    function setTestaPerBlock(uint256 _testaPerBlock) public onlyOwner{
        testaPerBlock = _testaPerBlock;
    }

    function setProgressive(int _maxProgressive, int _minProgressive) public onlyOwner{
        maxProgressive = _maxProgressive;
        minProgressive = _minProgressive;
    }

    function setNumberOfBlock(uint256 _numberOfBlock) public onlyOwner{
        numberOfBlock = _numberOfBlock;
    }

    function setActiveReward(uint256 _activeReward) public onlyOwner{
        activeReward = _activeReward;
    }
```

Listing 3.8: Multiple Setters in SpaceFarm

This parameter defines an important aspect of the protocol operation and needs to exercise extra care when configuring or updating it. Our analysis shows the update logic on it can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `testaPerBlock` may mint unreasonably large number of `TESTA`, hence greatly affecting valuation and user participation.

Recommendation Validate any changes regarding the system-wide parameter to ensure the changes fall in an appropriate range. If necessary, also consider emitting relevant events for its changes.

Status The issue has been confirmed.

3.7 Duplicate Pool Detection and Prevention

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The Testa Finance protocol provides incentive mechanisms that reward the staking of supported assets with TESTA tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded TESTA tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
        do.
    function add(uint256 startBlock, uint256 _allocPoint, address _lpToken, address
        _jETHToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        IUniswapV2Pair uniswap = IUniswapV2Pair(_lpToken);
        ( , uint112 _reserve1, ) = uniswap.getReserves();

        poolInfo.push(PoolInfo({
            jETHToken: IERC20(_jETHToken),
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accTestaPerShare: 0,
            debtIndexKey: 0,
```

```
            uniswap: uniswap,
            startLiquidity: _reserve1,
            startBlock: startBlock,
            initStartBlock: startBlock
        }));


    }
```

Listing 3.9: SpaceFarm::add()

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers. And the `TestaFarm` contract shares the same issue.

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
        do.
    function add(uint256 startBlock, uint256 _allocPoint, address _lpToken, address
        _jETHToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        IUniswapV2Pair uniswap = IUniswapV2Pair(_lpToken);
        ( , uint112 _reserve1, ) = uniswap.getReserves();

        poolInfo.push(PoolInfo({
            jETHToken: IERC20(_jETHToken),
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accTestaPerShare: 0,
            debtIndexKey: 0,
            uniswap: uniswap,
            startLiquidity: _reserve1,
            startBlock: startBlock,
            initStartBlock: startBlock
        }));


```

```
    }
```

Listing 3.10: Revised SpaceFarm::add()

Status The issue has been fixed by the following commit: `6d501e4`.

3.8 Recommended Explicit Pool Validity Checks

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

The reward mechanism in Testa Finance relies on the pool contract for a number of tasks, including the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers. In the following, we use the `SpaceFarm` contract as an example and show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 jETHToken;           // Address of LP token contract.
        IUniswapV2Pair uniswap;
        uint112 startLiquidity;
        uint256 allocPoint;       // How many allocation points assigned to this pool.
            Testa to distribute per block.
        uint256 lastRewardBlock;  // Last block number that Testa distribution occurs.
        uint256 accTestaPerShare; // Accumulated Testa per share, times 1e18. See below.
        uint256 debtIndexKey;
        uint256 startBlock;
        uint256 initStartBlock;
    }
```

Listing 3.11: SpaceFarm::PoolInfo

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending `TESTA` rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given_pool_id or _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
    // Deposit LP tokens to TestaFarm for Testa allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);

        if (user.amount > 0) {
          user.pendingTesta[pool.debtIndexKey] = pendingTesta(_pid, msg.sender);
        }

        if(_amount > 0) {
            pool.jETHToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }

        user.rewardDebt[pool.debtIndexKey] = user.amount.mul(pool.accTestaPerShare).div(1e18);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.12: SpaceFarm::deposit()

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including set(), deposit(), withdraw(), emergencyWithdraw(), and updatePool(). Another farming pool, i.e., TestaFarm, shares the same issue.

Recommendation Apply necessary sanity checks to ensure the given _pid is legitimate. Accordingly, a new modifier validatePool can be developed and appended to each function in the above list.

```
     modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }

    // Deposit LP tokens to TestaFarm for Testa allocation.
    function deposit(uint256 _pid, uint256 _amount) public validatePool(_pid)  {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);

        if (user.amount > 0) {
          user.pendingTesta[pool.debtIndexKey] = pendingTesta(_pid, msg.sender);
        }

        if(_amount > 0) {
            pool.jETHToken.safeTransferFrom(address(msg.sender), address(this), _amount);
```

```
1304            user.amount = user.amount.add(_amount);
1305        }
1306
1307        user.rewardDebt[pool.debtIndexKey] = user.amount.mul(pool.accTestaPerShare).div
            (1e18);
1308        emit Deposit(msg.sender, _pid, _amount);
1309    }
```

Listing 3.13: SpaceFarm::deposit()

Status The issue has been fixed by the following commit: `6d501e4`.

3.9 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `TestaFarm` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 1305) starts before effecting the update on internal states (lines 1307 – 130), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `emergencyWithdraw()` function.

```
1031    // Withdraw without caring about rewards. EMERGENCY ONLY.
1032    function emergencyWithdraw(uint256 _pid) public {
1033        PoolInfo storage pool = poolInfo[_pid];
1034        UserInfo storage user = userInfo[_pid][msg.sender];
```

```
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt[pool.debtIndexKey] = 0;
    }
```

Listing 3.14: TestaFarm::emergencyWithdraw()

Another similar violation can be found in the `deposit()` and `withdraw()` routines within the same contract.

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. An example revision on the `emergencyWithdraw` routine is shown below:

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount=user.amount
        user.amount = 0;
        user.rewardDebt[pool.debtIndexKey] = 0;
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.15: Revised TestaFarm::emergencyWithdraw()

Status The issue has been fixed by the following commit: `6d501e4`.

3.10 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.7, the Testa Finance protocol provides incentive mechanisms that reward the staking of supported assets with `TESTA` tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's Testa allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.16: SpaceFarm::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's Testa allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.17: SpaceFarm::set()

Status The issue has been fixed by the following commit: `6d501e4`.

3.11 Trust Issue of Admin Keys

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In Testa Finance, the privileged account plays a critical role in governing and regulating the system-wide operations (e.g., pool management, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for staking and rewards. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function emergencyWithdraw(uint256 amount) public onlyOwner{
        uint256 totalETHBal = address(this).balance;
        if(amount > totalETHBal){
            SafeToken.safeTransferETH(msg.sender, totalETHBal);
        }else{
            SafeToken.safeTransferETH(msg.sender, amount);
        }
    }
```

Listing 3.18: PlanetFarm::set()

Specifically, we examine the privileged function `emergencyWithdraw()` in `PlanetFarm`. Notice that the privileged account is able to withdraw the full `ETH` balance.

Moreover, if a supplying user in the `bank` or `station` wants to withdraw the supplied assets, the user needs to call `withdraw()`. In the following, we show the code snippet of this `withdraw()` routine.

```
    /// @dev Withdraw ETH from the bank by burning the share tokens.
    function withdraw(uint256 share) external accrue(0) nonReentrant {
        uint256 amount = share.mul(totalETH()).div(totalSupply());
        _burn(msg.sender, share);
        uint256 profit = amount.sub(share);
        uint256 referralAmount = profit.mul(universe.getUniverseShare()).div(10000);
        address payable refferal = universe.getRefferral();
        (bool sent, bytes memory data) = refferal.call.value(referralAmount)("");
        require(sent, "Failed to transfer");
        SafeToken.safeTransferETH(msg.sender, amount.sub(referralAmount));
    }
```

Listing 3.19: Station::withdraw()

Apparently, the withdraw path is designed to compute and push the referral reward. This means that the `refferal` is always invoked in the critical path. If the referral call fails, all supplying users may not be able to withdraw their funds back. This could be worrisome and these privileged accounts need to properly managed and be transparent to the community.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. And 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. The team plans to initially hold the admin key in case there are some fixes, and later will add governance or burn the key if necessary.

3.12 Improved Precision By Multiplication And Division Reordering

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: UniswapOrbitConfig
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [2]

Description

In Testa Finance, there is a UniswapOrbitConfig contract that allows for customization of various configuration paramters for each supported orbit. Each orbit configuration is required to export the following interfaces: acceptDebt(), launcher(), and terminator(). The first interface indicates whether the given orbit accepts more debt; and the last two interface returns the work factor and the kill factor, respectively.

These three interfaces all rely on one helper function, i.e., isStable(). This helper is designed to detect whether the related Uniswap pair is stable or not. To elaborate, we show below this helper's implementation.

```
    /// @dev Return whether the given orbit is stable, presumably not under manipulation
        .
    function isStable(address orbit) public view returns (bool) {
        IUniswapV2Pair lp = IUniswapOrbit(orbit).lpToken();
        address token0 = lp.token0();
        address token1 = lp.token1();
        // 1. Check that reserves and balances are consistent (within 1%)
        (uint256 r0, uint256 r1,) = lp.getReserves();
        uint256 t0bal = token0.balanceOf(address(lp));
        uint256 t1bal = token1.balanceOf(address(lp));
        require(t0bal.mul(100) <= r0.mul(101), "bad t0 balance");
        require(t1bal.mul(100) <= r1.mul(101), "bad t1 balance");
        // 2. Check that price is in the acceptable range
```

```
182         (uint256 price, uint256 lastUpdate) = oracle.getPrice(token0, token1);
183         require(lastUpdate >= now - 7 days, "price too stale");
184         uint256 lpPrice = r1.mul(1e18).div(r0);
185         uint256 maxPriceDiff = orbits[orbit].maxPriceDiff;
186         require(lpPrice <= price.mul(maxPriceDiff).div(10000), "price too high");
187         require(lpPrice >= price.mul(10000).div(maxPriceDiff), "price too low");
188         // 3. Done
189         return true;
190     }
```

Listing 3.20: UniswapOrbitConfig:: isStable ()

It comes to our attention that this routine computes the requirements `require(lpPrice <= price.mul(maxPriceDiff).div(10000)` and `require(lpPrice >= price.mul(10000).div(maxPriceDiff)` (lines 67–68).

It is important to note that the lack of `float` support in `Solidity` may introduce subtle, but troublesome issue: precision loss. One possible precision loss stems from the computation when both multiplication (`mul`) and division (`div`) are involved. Specifically, the requirements at lines 67 – 68 is better performed as follows: `require(lpPrice.mul(10000)<= price.mul(maxPriceDiff)` and `require(lpPrice.mul(maxPriceDiff)>= price.mul(10000)`

A better approach is to avoid any unnecessary division operation that might lead to precision loss. In other words, the requirement of the form `A > B / C` can be converted into `A * C > B` under the condition that `A * C` does not introduce any overflow.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in `Solidity`. An example revision to the above requirements is shown as: `require(lpPrice.mul(10000)<= price.mul(maxPriceDiff)` and `require(lpPrice.mul(maxPriceDiff)>= price.mul(10000)`

Status The issue has been confirmed. However, considering the consistency with the initial version from `Alpha Homora V1`, the team decides to leave as is.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Testa Finance protocol. The audited system presents a unique addition to current DeFi offerings by providing users opportunities for staking, unstaking or yield farming the digital assets. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.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: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

[12] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

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

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