SMART CONTRACT AUDIT REPORT

for

YUAN FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China
November 2, 2020

Document Properties

Client	Yuan Finance
Title	Smart Contract Audit Report
Target	Yuan
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 2, 2020	Xuxian Jiang	Final Release
0.2	October 28, 2020	Xuxian Jiang	Additional Findings
0.1	October 25, 2020	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

Given the opportunity to review the design document and related smart contract source code of the **Yuan** protocol, 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 Yuan

Yuan is an elastic token tracking the exchange rate of `Chinese Yuan` against `USD`. Unlike other elastic tokens, which purely rely on money expansion/contraction driven by deterministic algorithm, Yuan plans to develop or co-develop other protocols (i.e., AMM, lending etc) by additionally injecting more demand side variables into the equation. By design, Yuan is a 100% community-driven project and there is no pre-mining and centralized control on its operation and governance. All proposals are voted fully on-chain and managed through the governance.

The basic information of Yuan is as follows:

Table 1.1: Basic Information of Yuan

Item	Description
Issuer	Yuan Finance
Website	https://1yuan.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 2, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Yuan assumes a trusted oracle with timely price feeds on `USDx - Chinese Yuan` exchange rate and the oracle itself is not part of this audit.

- https://github.com/yuan-finance/yuan.git (1a80498f)

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

- https://github.com/yuan-finance/yuan.git (42ac632)

1.2 About PeckShield

PeckShield Inc. [20] 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 [15]:

- 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.

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) [14], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 1.3: 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

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 Yuan. 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	5	■■■■■
Informational	4	■■■■
Total	11	

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 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, 5 low-severity vulnerabilities and 4 informational recommendations.

Table 2.1: Key Yuan Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Overflow Mitigation in notifyRewardAmount()	Numeric Errors	Fixed
PVE-002	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-003	Informational	Improved Accuracy of getFixedRewardRate()	Business Logics	Confirmed
PVE-004	Low	Inconsistent RewardAdded And RewardPaid Events	Business Logics	Fixed
PVE-005	Informational	Simplified Logic in getReward()	Business Logics	Fixed
PVE-006	Low	Unexpected Kick-Off of YUANIncentivizer	Security Features	Confirmed
PVE-007	Low	Inconsistent ScalingFactor of initreward-Related Mints	Business Logics	Fixed
PVE-008	Informational	Removal of Unneeded Migration Functionality	Coding Practices	Confirmed
PVE-009	Medium	Blocked Rebasing With Paused PriceOracle	Business Logics	Confirmed
PVE-010	Informational	Gas Optimization in removeUniPair() And removeBalPair()	Coding Practices	Confirmed
PVE-011	Low	Improved Sanity Checks When Updating Important System Parameters	Error Conditions, Return Values, Status Codes	Fixed

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 Potential Overflow Mitigation in notifyRewardAmount()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Incentivize Pools`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [2]

Description

The Yuan protocol shares the same architectural design of `YAM` with similar incentivizer mechanisms. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `YUANIncentivizer` to update and use the latest reward rate.

The reason is due to the known potential overflow pitfall when a new oversized reward amount is added into the pool. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integer, it is possible for their multiplication to have an undesirable overflow (lines 780−786), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution` (through the `notifyRewardAmount()` function).

```
761     modifier updateReward(address account) {
762         rewardPerTokenStored = rewardPerToken();
763         lastUpdateTime = lastTimeRewardApplicable();
764         if (account != address(0)) {
765             rewards[account] = earned(account);
766             userRewardPerTokenPaid[account] = rewardPerTokenStored;
767         }
768         _;
769     }
770 
771     function lastTimeRewardApplicable() public view returns (uint256) {
772         return Math.min(block.timestamp, periodFinish);
```

```
    }

    function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
            rewardPerTokenStored.add(
                lastTimeRewardApplicable()
                    .sub(lastUpdateTime)
                    .mul(rewardRate)
                    .mul(1e18)
                    .div(totalSupply())
            );
    }
```

Listing 3.1: YUANIncentivizer.sol

Apparently, this issue is made possible if the reward amount is given as the argument to `notifyRewardAmount()` such that the calculation of `rewardRate.mul(1e18)` always overflows, hence locking all deposited funds! Note that an authentication check on the caller of `notifyRewardAmount()` greatly alleviates such concern. Currently, only the `rewardDistribution` address is able to call `notifyRewardAmount()` and this address is set by the owner. Apparently, if the owner is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the given reward amount will not be oversized to overflow and lock users' funds.

We point out that the `YUANIncentivizer` contract has necessary anti-overflow mechanism in place, i.e., `require(reward < uint256(-1)/ 10**18, "rewards too large, would lock")`. However, it may consider additional leeway to accommodate the inherent scaling factor of the rebasing `YUAN` with the following revision: `require(reward < uint256(-1)/ 10**22, "rewards too large, would lock")`.

With the presence of several similar incentive contracts, we observe inconsistency in mitigating the above overflow risk. Specifically, there is no anti-overflow mitigation place yet in `YUANUSDCETHPool`, while `YUANUSDxUSDCPool` has limited, insufficient mitigation.

Recommendation Be consistent and sufficient in mitigating the potential overflow risk in all incentivize pools. An example revision to the `notifyRewardAmount()` routine in `YUANUSDxUSDCPool` is shown below. Note the following revision also includes an arithmetic optimization (Section 3.2) and an improved sanity check (Section 3.6).

```
    function notifyRewardAmount(uint256 reward)
        external
        onlyRewardDistribution
        updateReward(address(0))
    {
        require(reward < uint256(-1) / 10**22, "rewards too large, would lock");

```

```
        uint256 _firstReward = reward.mul(1e18).div(2e18-(2e18>>DURATION.div(
            halveInterval));
        if (block.timestamp > starttime) {
            require(block.timestamp >= periodFinish && periodFinish>0, "not over yet");
            initialRewardRate = _firstReward.div(halveInterval);
            lastUpdateTime = block.timestamp;
            distributionTime = block.timestamp;
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(reward);
        } else {
            initialRewardRate = _firstReward.div(halveInterval);
            lastUpdateTime = starttime;
            distributionTime = starttime;
            periodFinish = starttime.add(DURATION);
            emit RewardAdded(reward);
        }
    }
```

Listing 3.2: YUANUSDxUSDCPool.sol (revised)

Status This issue has been fixed in the commit: 30b15a18ac3a09c30de7f45abff36ae15c090c38.

3.2 Improved Precision By Multiplication-Before-Division

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `YUANUSDxUSDCPool`, `YUANRebaser`
- Category: Numeric Errors [13]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `notifyRewardAmount()` as an example. This routine is used to inject additional rewards into the pool and requires the calculation of new reward rate for staking users.

```
    function notifyRewardAmount(uint256 reward)
        external
        onlyRewardDistribution
        updateReward(address(0))
```

```
890     {
891         uint256 _firstReward = (reward >> 1).mul(1e18).div(
892             1e18 - (5e17 >> (DURATION.div(halveInterval).sub(1)))
893         );
894         if (block.timestamp > starttime) {
895             require(block.timestamp >= periodFinish, "not over yet");
896             initialRewardRate = _firstReward.div(halveInterval);
897             lastUpdateTime = block.timestamp;
898             distributionTime = block.timestamp;
899             periodFinish = block.timestamp.add(DURATION);
900             emit RewardAdded(reward);
901         } else {
902             initialRewardRate = _firstReward.div(halveInterval);
903             lastUpdateTime = starttime;
904             distributionTime = starttime;
905             periodFinish = starttime.add(DURATION);
906             emit RewardAdded(reward);
907         }
908     }
```

Listing 3.3: YUANUSDxUSDCPool.sol (revised)

We notice the calculation of `_firstReward` (line 891) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `reward.mul(1e18).div(2e18-(2e18>>DURATION.div(halveInterval))`.

Similarly, other calculations at lines 476 – 484 of `YUANRebaser` can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

```
470         if (positive) {
471             uint256 rebaseMintPerc = rebaseMintPercs[0] +
472                 rebaseMintPercs[1] +
473                 rebaseMintPercs[2];
474             uint256 mintPerc = indexDelta.mul(rebaseMintPerc).div(BASE);

476             mintAmounts[0] = currSupply
477                 .mul(indexDelta.mul(rebaseMintPercs[0]).div(BASE))
478                 .div(BASE);
479             mintAmounts[1] = currSupply
480                 .mul(indexDelta.mul(rebaseMintPercs[1]).div(BASE))
481                 .div(BASE);
482             mintAmounts[2] = currSupply
483                 .mul(indexDelta.mul(rebaseMintPercs[2]).div(BASE))
484                 .div(BASE);

486             indexDelta = indexDelta.sub(mintPerc);
487         }
```

Listing 3.4: YUANRebaser.sol

Recommendation Revise the above calculations to better mitigate possible precision loss. The example revision of `notifyRewardAmount()` has been shown in Section 3.1.

Status This issue has been fixed in the commit: 393256c9e60eb69f3b65af624674d1e383472459.

3.3 Improved Accuracy of getFixedRewardRate()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YUANUSDxUSDCPool`
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [8]

Description

The reward distribution process, by design, requires real-time updates on a number of interwoven parameters, e.g., `rewardRate`, `lastUpdateTime`, and `periodFinish`. The incentive mechanism in Yuan has implemented its own `checkhalve()` that reduces `rewardRate` by half according to the specified `halveInterval`.

We have examined the execution logics of possible occasions that may inject new rewards into the pool and our analysis shows that the helper routine (i.e., `getFixedRewardRate()`) to obtain the reward rate at the given timestamp can be better improved. Specifically, we show below the related code snippet of `getFixedRewardRate()`.

The routine's execution logic is rather straightforward in firstly calculating the specific period in which the given timestamp falls and then determining the reward rate in that period. However, our analysis shows that the reward rate is only valid in the time range of `[distributionTime, periodFinish)`. The current implementation returns a non-zero reward rate that should be zero after the time passes the `periodFinish`.

```
    function getFixedRewardRate(uint256 _timestamp)
        public
        view
        returns (uint256)
    {
        if (_timestamp < distributionTime) return 0;
        return
            initialRewardRate >>
            (Math.min(_timestamp, periodFinish).sub(distributionTime) /
                halveInterval);
    }
```

Listing 3.5: YUANUSDxUSDCPool.sol

Recommendation Revise the logic in `getFixedRewardRate()` to better validate the applicable time range for the requested reward rate. An example revision is shown below.

```
    function getFixedRewardRate(uint256 _timestamp)
        public
        view
        returns (uint256)
    {
        if (_timestamp < distributionTime && _timestamp >= periodFinish) return 0;
        return
            initialRewardRate >>
            (Math.min(_timestamp, periodFinish).sub(distributionTime) /
                halveInterval);
    }
```

Listing 3.6: YUANUSDxUSDCPool.sol

Status This issue has been confirmed. Since this issue only affects the UI (and the UI issue can be readily applied with remedy measures), the team decides to leave it as is.

3.4 Inconsistent RewardAdded And RewardPaid Events

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YUANUSDxUSDCPool`
- Category: Business Logics [11]
- CWE subcategory: CWE-837 [7]

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 monitoring tools.

`Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. For example, Yuan has a number of risk parameters that are dynamically adjustable via governance. However, the current implementation can be greatly benefited by emitting related events in a consistent manner.

If the following, we use the reward supply and distribution in `YUANUSDxUSDCPool` as an example. The reward can be retrieved via `getReward()` and is supplied via `notifyRewardAmount()`.

```
    function getReward() public updateReward(msg.sender) checkStart {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
            rewards[msg.sender] = 0;
            uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
```

```
810             uint256 trueReward = reward.mul(scalingFactor).div(10**18);
811             yuan.safeTransfer(msg.sender, trueReward);
812             emit RewardPaid(msg.sender, trueReward);
813         }
814     }
```

Listing 3.7: YUANUSDxUSDCPool.sol

```
886     function notifyRewardAmount(uint256 reward)
887         external
888         onlyRewardDistribution
889         updateReward(address(0))
890     {
891         uint256 _firstReward = (reward >> 1).mul(1e18).div(
892             1e18 - (5e17 >> (DURATION.div(halveInterval).sub(1)))
893         );
894         if (block.timestamp > starttime) {
895             require(block.timestamp >= periodFinish, "not over yet");
896             initialRewardRate = _firstReward.div(halveInterval);
897             lastUpdateTime = block.timestamp;
898             distributionTime = block.timestamp;
899             periodFinish = block.timestamp.add(DURATION);
900             emit RewardAdded(reward);
901         } else {
902             initialRewardRate = _firstReward.div(halveInterval);
903             lastUpdateTime = starttime;
904             distributionTime = starttime;
905             periodFinish = starttime.add(DURATION);
906             emit RewardAdded(reward);
907         }
908     }
```

Listing 3.8: YUANUSDxUSDCPool.sol

We notice that the supplied reward amount is recorded in the emitted event, i.e., `RewardAdded(reward)` (lines 900 and 906), and the claimed reward is recorded as well via `RewardPaid(msg.sender, trueReward)` (line 812). Interestingly, the supplied amount is recorded without taking into account the `scalingFactor` while the claimed amount takes it into account.

Recommendation Resolve the inconsistency in emitted events based on the scale factor.

Status This issue has been fixed in the commit: bd07d7339da4bf4184dd19b4ef9ed9474b9fa34a.

3.5 Simplified Logic in getReward()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Incentivize Pools`
- Category: Business Logics [11]
- CWE subcategory: CWE-770 [6]

Description

In the `YUANIncentives` contract, the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 755).

```
805     function getReward() public updateReward(msg.sender) checkStart {
806         uint256 reward = earned(msg.sender);
807         if (reward > 0) {
808             rewards[msg.sender] = 0;
809             uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
810             uint256 trueReward = reward.mul(scalingFactor).div(10**18);
811             yuan.safeTransfer(msg.sender, trueReward);
812             emit RewardPaid(msg.sender, trueReward);
813         }
814     }
```

Listing 3.9: YUANUSDxUSDCPool.sol

```
751     modifier updateReward(address account) {
752         rewardPerTokenStored = rewardPerToken();
753         lastUpdateTime = lastTimeRewardApplicable();
754         if (account != address(0)) {
755             rewards[account] = earned(account);
756             userRewardPerTokenPaid[account] = rewardPerTokenStored;
757         }
758         _;
759     }
```

Listing 3.10: YUANUSDxUSDCPool.sol

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 806).

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
    function getReward() public updateReward(msg.sender) checkStart {
        uint256 reward = rewards[msg.sender];
        if (reward > 0) {
            rewards[msg.sender] = 0;
            uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
            uint256 trueReward = reward.mul(scalingFactor).div(10**18);
            yuan.safeTransfer(msg.sender, trueReward);
            emit RewardPaid(msg.sender, trueReward);
        }
    }
```

Listing 3.11: YUANUSDxUSDCPool.sol

Status This issue has been fixed in the commit: e571545e11ae8c397d30b41a1a09ece2d5eee4b2.

3.6 Unexpected Kick-Off of YUANIncentivizer

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `YUANIncentivizer`
- Category: Security Features [9]
- CWE subcategory: CWE-284 [3]

Description

The built-in `YUANIncentivizer` contract includes an incentivizer logic that rewards early users with newly minted `YUAN` tokens. In essence, by following the initial base from the `Synthetix` protocol, The incentivizer logic supports three occasions that may bring additional reward amounts into the pool.

The first occasion is the configured `initreward` that initializes and bootstraps the rewarding process (lines 873 – 883); The second occasion is the explicit injection of rewards via `notifyRewardAmount (reward)` (lines 862 – 871) after the initialization; The third occasion happens with the recurring, but decayed `initreward` (triggered via `checkhalve`). In each occasion, the protocol emits related events that record the new reward amount into the pool. These events are located at line 883 (event I), 871 (event II), and 844 (event III) respectively. For illustration, we show the related routines below.

```
854    function notifyRewardAmount(uint256 reward)
855        external
856        onlyRewardDistribution
857        updateReward(address(0))
858    {
859        // https://sips.synthetix.io/sips/sip-77
```

```
        require(reward < uint256(-1) / 10**18, "rewards too large, would lock");
        if (block.timestamp > starttime) {
            if (block.timestamp >= periodFinish) {
                rewardRate = reward.div(DURATION);
            } else {
                uint256 remaining = periodFinish.sub(block.timestamp);
                uint256 leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(DURATION);
            }
            lastUpdateTime = block.timestamp;
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(reward);
        } else {
            require(
                initreward < uint256(-1) / 10**18,
                "rewards too large, would lock"
            );
            require(!initialized, "already initialized");
            initialized = true;
            yuan.mint(address(this), initreward);
            rewardRate = initreward.div(DURATION);
            lastUpdateTime = starttime;
            periodFinish = starttime.add(DURATION);
            emit RewardAdded(reward);
        }
    }
```

Listing 3.12: YUANIncentivizer.sol

To further elaborate, we show below the code snippet of the `stake()` routine. We notice that `stake()` essentially relays the call to the inherited contract, which properly transfers the staked assets into the pool and makes necessary bookkeeping in its internal records.

```
    function stake(uint256 amount)
        public
        updateReward(msg.sender)
        checkhalve
        checkStart
    {
        require(amount > 0, "Cannot stake 0");
        super.stake(amount);
        emit Staked(msg.sender, amount);
    }
```

Listing 3.13: YUANIncentivizer.sol

We point out that the `stake()` routine has an associated modifier, i.e., `checkhalve()`. This modifier, as the name indicates, reduces the `initreward` amount based on the pre-determined decaying schedule and adjusts the `rewardRate` accordingly.

```
    modifier checkhalve() {
        if (block.timestamp >= periodFinish) {
```

```
            initreward = initreward.mul(90).div(100);
            uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
            uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
            yuan.mint(address(this), newRewards);

            rewardRate = initreward.div(DURATION);
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(initreward);
        }
        _;
    }
```

Listing 3.14: YUANIncentivizer.sol

However, the logic in the modifier has a flaw that fails to validate whether the rewarding process has been started or not. As a result, even the rewarding has not been kicked off yet, this modifier can still mint new `YUAN` tokens and activate the token distribution process. This is apparently unintended and violates the proposal design. Note that the occasion II shares a similar issue.

Recommendation Ensure that the reward process will not been activated until it has been properly initialized. An example revision to the modifier can be found as follows:

```
    modifier checkhalve() {
        if (breaker) {
          // do nothing
        } else if (block.timestamp >= periodFinish && periodFinish>0 ) {
            initreward = initreward.mul(90).div(100);
            uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
            uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
            yam.mint(address(this), newRewards);

            lastUpdateTime = block.timestamp;
            rewardRate = initreward.div(DURATION);
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(initreward);
        }
        _;
    }
```

Listing 3.15: YAMIncentivizerWithVoting.sol (revised)

Status This issue has been confirmed.

3.7 Inconsistent ScalingFactor of initreward-Related Mints

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YUANIncentivizer`
- Category: Business Logics [11]
- CWE subcategory: CWE-770 [6]

Description

As mentioned in Section 3.6, the `YUANIncentivizer` contract rewards early users with newly minted `YUAN` tokens. And there are three occasions that may bring additional reward amounts into the pool.

In this section, we further examine the first and third occasions as they are directly related to the pre-configured `initreward`. Note that the first occasion initializes and bootstraps the rewarding process while the third occasion happens with the recurring, but decayed `initreward` (triggered via `checkhalve`).

Within each occasion, the protocol mints corresponding `YUAN` tokens into the reward pool. Due to the rebasing nature of `YUAN`, the minted amount needs to be accordingly adjusted. However, we notice that the adjustment is only made in the third occasion, but not the first occasion.

```
835    modifier checkhalve() {
836        if (block.timestamp >= periodFinish) {
837            initreward = initreward.mul(90).div(100);
838            uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
839            uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
840            yuan.mint(address(this), newRewards);
841
842            rewardRate = initreward.div(DURATION);
843            periodFinish = block.timestamp.add(DURATION);
844            emit RewardAdded(initreward);
845        }
846        _;
847    }
```

Listing 3.16: YUANIncentivizer.sol

Specifically, the third occasion (shown in the above code snippet) queries current `scalingFactor` and multiplies the reward amount with the `scalingFactor` as the minted amount (line 840). However, the first occasion simply mints the reward amount without taking into account the `scalingFactor`. As a result, current implementation unnecessarily introduces inconsistency in the minted amounts.

Recommendation Be consistent when minting the token amount into the reward pool. An example revision to the `notifyRewardAmount` routine can be found as follows and this revision takes `scalingFactor` into account.

```
    function notifyRewardAmount(uint256 reward)
        external
        onlyRewardDistribution
        updateReward(address(0))
    {
        // https://sips.synthetix.io/sips/sip-77
        require(reward < uint256(-1) / 10**18, "rewards too large, would lock");
        if (block.timestamp > starttime) {
            if (block.timestamp >= periodFinish) {
                rewardRate = reward.div(DURATION);
            } else {
                uint256 remaining = periodFinish.sub(block.timestamp);
                uint256 leftover = remaining.mul(rewardRate);
                rewardRate = reward.add(leftover).div(DURATION);
            }
            lastUpdateTime = block.timestamp;
            periodFinish = block.timestamp.add(DURATION);
            emit RewardAdded(reward);
        } else {
            require(
                initreward < uint256(-1) / 10**18,
                "rewards too large, would lock"
            );
            require(!initialized, "already initialized");
            initialized = true;
            uint256 scalingFactor = YUAN(address(yuan)).yuansScalingFactor();
            uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
            yam.mint(address(this), newRewards);
            rewardRate = initreward.div(DURATION);
            lastUpdateTime = starttime;
            periodFinish = starttime.add(DURATION);
            emit RewardAdded(reward);
        }
    }
```

Listing 3.17: YAMIncentivizerWithVoting.sol (revised)

Status This issue has been fixed in the commit: bd07d7339da4bf4184dd19b4ef9ed9474b9fa34a.

3.8 Removal of Unneeded Migration Functionality

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YUANToken`
- Category: Coding Practices [10]
- CWE subcategory: CWE-563 [5]

Description

Yuan makes use of a number of reference libraries and contracts, such as `SafeMath`, `ERC20`, and `Uniswap`, to facilitate the protocol implementation and organization. For instance, the `YUANRebaser` smart contract interacts with at least four different external contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `YUANToken` contract, the current functionality of supporting migration is not needed as it is specific to the forked `YAM` protocol that migrates for an earlier version to current one.

```
    /** @notice sets the migrator
     * @param migrator_ The address of the migrator contract to use for authentication.
     */
    function _setMigrator(address migrator_) external onlyGov {
        address oldMigrator = migrator_;
        migrator = migrator_;
        emit NewMigrator(oldMigrator, migrator_);
    }

\begin{lstlisting}[language=Solidity, caption=\lstinline{YUANToken.sol}, firstnumber=75,
    basicstyle=\scriptsize, numberblanklines=true]
    function _mint(address to, uint256 amount) internal {
        if (msg.sender == migrator) {
            // migrator directly uses v2 balance for the amount

            // increase initSupply
            initSupply = initSupply.add(amount);

            // get external value
            uint256 scaledAmount = _yuanToFragment(amount);

            // increase totalSupply
            totalSupply = totalSupply.add(scaledAmount);

            // make sure the mint didnt push maxScalingFactor too low
            require(
                yuansScalingFactor <= _maxScalingFactor(),
                "max scaling factor too low"
```

```
            );

            // add balance
            _yuanBalances[to] = _yuanBalances[to].add(amount);

            // add delegates to the minter
            _moveDelegates(address(0), _delegates[to], amount);
            emit Mint(to, scaledAmount);
            emit Transfer(address(0), to, scaledAmount);
        } else {
            // increase totalSupply
            totalSupply = totalSupply.add(amount);

            // get underlying value
            uint256 yuanValue = _fragmentToYuan(amount);

            // increase initSupply
            initSupply = initSupply.add(yuanValue);

            // make sure the mint didnt push maxScalingFactor too low
            require(
                yuansScalingFactor <= _maxScalingFactor(),
                "max scaling factor too low"
            );

            // add balance
            _yuanBalances[to] = _yuanBalances[to].add(yuanValue);

            // add delegates to the minter
            _moveDelegates(address(0), _delegates[to], yuanValue);
            emit Mint(to, amount);
            emit Transfer(address(0), to, amount);
        }
    }
```

Listing 3.18: YUANToken.sol

We also observe a few redundant code. One example is that the built-in support of interacting with `Balancer` in `YUANRebaser` is not needed for the time being and may be considered for removal.

Recommendation Consider the removal of the unused code/functionality.

Status The examined migration functionality is based on the original `YAMv3` protocol without any modification. Considering that it does not affect normal functionalities in Yuan, the team decides to keep it intact.

3.9 Blocked Rebasing With Paused PriceOracle

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: YUANRebaser
- Category: Business Logics [11]
- CWE subcategory: CWE-770 [6]

Description

Yuan is an experimental protocol that builds up an elastic supply cryptocurrency and the elastic supply capability is implemented by the YUANRebaser contract that measures current price fluctuation and then dynamically inflates or deflates the YUAN total supply based on the pre-configured adjustment schedule. The price fluctuation is measured by reading current exchangeRate, i.e., the time-weighted average price (or TWAP), from the UniswapV2 trading pair of YUAN and USDx. It further gauges the price by taking into account the exchange rate between USDx and Chinese Yuan, which is provided from a trusted price oracle.

```
    function getTWAP() internal returns (uint256) {
        (
            uint256 priceCumulative,
            uint32 blockTimestamp
        ) = UniswapV2OracleLibrary.currentCumulativePrices(
            uniswap_pair,
            isToken0
        );
        uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired

        // no period check as is done in isRebaseWindow

        // overflow is desired
        uint256 priceAverage = uint256(
            uint224((priceCumulative - priceCumulativeLast) / timeElapsed)
        );

        priceCumulativeLast = priceCumulative;
        blockTimestampLast = blockTimestamp;

        // BASE is on order of 1e18, which takes 2^60 bits
        // multiplication will revert if priceAverage > 2^196
        // (which it can because it overflows intentially)
        if (priceAverage > uint192(-1)) {
            // eat loss of precision
            // effectively: (x / 2**112) * 1e18
            return (priceAverage >> 112) * BASE;
        }
        // cant overflow
```

```
847         // effectively: (x * 1e18 / 2**112)
848         return (priceAverage * BASE) >> 112;
849     }
850
851     /**
852      * @notice Calculates exchange rate
853      *
854      */
855     function getExchangeRate() internal returns (uint256) {
856         // TODO: Check the timestamp of the price
857         uint256 price = IPriceOracle(priceOracle).getPrice(reserveToken);
858         require(price > 0, "Reserve token price can not be 0");
859
860         return getTWAP().mul(price).div(BASE);
861     }
```

Listing 3.19: YUANRebaser.sol

For elaboration, we show above the `getTWAP()` routine that is responsible for reading the `TWAP` from the chosen `UniswapV2` trading pair as well as the the `getExchangeRate()` routine that integrates `getTWAP()` with the off-chain price oracle. Note that `getTWAP()` basically measures the cumulative trading price in so-called `uq112x112 price * seconds` units so that we simply divide it by the time elapsed to obtain the average price. The `getExchangeRate()` routine reads the price oracle which however has a built-in method to pause. A paused price oracle simply returns 0, which effectively reverts the current rebase operation (line 858).

```
    function _setPaused(bool requestedState) public returns (uint256) {
        // Check caller = anchorAdmin
        if (msg.sender != anchorAdmin) {
            return
                failOracle(
                    address(0),
                    OracleError.UNAUTHORIZED,
                    OracleFailureInfo.SET_PAUSED_OWNER_CHECK
                );
        }

        paused = requestedState;
        emit SetPaused(requestedState);

        return uint256(Error.NO_ERROR);
    }
```

Listing 3.20: PriceOracle.sol

A reverted reading of `getExchangeRate()` immediately fails this rebasing attempt and undermines the elastic supply capability of `YAM`.

Recommendation The dependence on an off-chain component for the critical rebasing operations invites community attentions and challenges. It also necessitates additional thoughts for improved and robust design.

Status This issue has been confirmed. However, due to the lack of a reliable feed of the on-chain exchange rate between `USDx` and `Chinese Yuan`, the team decides to leave it as is and plans to upgrade it later after a reliable solution for the needed on-chain exchange rate surfaces.

3.10 Gas Optimization in removeUniPair() And removeBalPair()

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YUANRebaser`
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [8]

Description

The Yuan protocol supports the real-time `sync()` of `UniswapV2` pairs (and `gulp()` of `Balancer` pairs) right after applying the rebasing operation. The purpose is to take advantage of latest asset increase, if any, of the pair pool so that the inflating-related swaps use the latest reserves. As there may have a number of pairs for inclusion, the implementation maintains two arrays: one for `UniswapV2` pairs and another for `Balancer` pairs.

While reviewing the support of new `UniswapV2` pairs and `Balancer` pairs, we notice the removal of certain element indexed by `index` from the respective array could benefit from known best practice in reducing the gas consumption. Especially, when we have a large array of related pairs, the improvement could save a lot of gas!

```
    function removeUniPair(uint256 index) public onlyGov {
        if (index >= uniSyncPairs.length) return;

        for (uint256 i = index; i < uniSyncPairs.length - 1; i++) {
            uniSyncPairs[i] = uniSyncPairs[i + 1];
        }
        uniSyncPairs.length--;
    }

    function removeBalPair(uint256 index) public onlyGov {
        if (index >= balGulpPairs.length) return;

        for (uint256 i = index; i < balGulpPairs.length - 1; i++) {
            balGulpPairs[i] = balGulpPairs[i + 1];
        }
        balGulpPairs.length--;
```

```
281     }
```

Listing 3.21: YUANRebaser.sol

The idea is that we could simply replace the element to be removed with the last element in the array and `pop()` the last element out. This reduces a lot of gas usage if you need to walk through a huge array and replace each element with the next element as in current implementation (line 268 – 270).

Recommendation Replace the element to be removed with the last element and `pop()` the last element out.

```
265     function removeUniPair(uint256 index) public onlyGov {
266         if (index >= uniSyncPairs.length) return;

268         uniSyncPairs[index] = uniSyncPairs[uniSyncPairs.length-1];
269         uniSyncPairs.length--;
270     }

272     function removeBalPair(uint256 index) public onlyGov {
273         if (index >= balGulpPairs.length) return;

275         balGulpPairs[index] = balGulpPairs[balGulpPairs.length-1];
276         balGulpPairs.length--;
277     }
```

Listing 3.22: YUANRebaser.sol (revised)

Status This issue has been confirmed. Since the related functions are forked from the original `YAMv3` codebase, the team decides to leave it as is for the time being.

3.11 Improved Sanity Checks When Updating Important System Parameters

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `YUANRebaser`
- Category: Status Codes [12]
- CWE subcategory: CWE-391 [4]

Description

The rebasing operation in Yuan is a complex one and requires a number of delicate system-reconfigurable parameters, e.g., `maxSlippageFactor`, `rebaseMintPerc`, and `minRebaseTimeIntervalSec`. Note that `maxSlippageFactor` specifies the allowable slippage of `YUAN` trading price when inflating its total supply. The `rebaseMintPerc` parameters control the three percentages of minted `YUAN` into `treasury`, `liquidity`, and `lending` respectively. The `minRebaseTimeIntervalSec` parameter governs the rebasing interval.

However, we notice that the updates of `rebaseMintPerc` deserve additional sanity checks. As an example, we show here the helper routine that adjusts `rebaseMintPerc`. Note that if `rebaseMintPerc` is not set properly, say more than 100%, every rebasing attempt would fail.

```
    function setRebaseMintPerc(uint256 reserveIndex_, uint256 rebaseMintPerc_)
        public
        onlyGov
    {
        require(reserveIndex_ < 3);
        require(rebaseMintPerc_ < MAX_MINT_PERC_PARAM);

        uint256 oldRebaseMintPercs = rebaseMintPercs[reserveIndex_];
        rebaseMintPercs[reserveIndex_] = rebaseMintPerc_;

        emit NewRebaseMintPercent(
            reserveIndex_,
            oldRebaseMintPercs,
            rebaseMintPerc_
        );
    }
```

Listing 3.23: YUANRebaser.sol

The current implementation has a threshold-validation in place. Note that the current `MAX_MINT_PERC_PARAM` is initialized as 25%, which allows each minted percentage to be no more than 25%. However, since there are three such percentages, the current sanity checks may not be sufficient.

Note that these routines update these important parameters that may impact the overall operation and health, great care needs to be taken to ensure these parameters fall in an appropriate range.

Recommendation Apply necessary sanity checks to ensure the updated parameters always fall in a proper range. Also emit corresponding events when these risk parameters are being updated.

Status This issue has been fixed in the commit: 4672ac59b4e758c874f2929c38cfd82bfe9ee91c.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Yuan protocol. Following `YAM`, the proposed `YUAN` protocol presents a valuable add-on for current novel experiments of on-chain community-based governance and elastic supply cryptocurrencies. We are keen in its exploration and any insights from the experiment will be very helpful for the DeFi community. This protocol follows the clean solid design of `YAM` with a coherent organization 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.

5 | Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [16, 17, 18, 19, 21].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [22] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

[4] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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

[6] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[7] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

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

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

[16] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[17] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[18] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[19] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[21] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[22] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.