SMART CONTRACT AUDIT REPORT

for

YAM FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

October 23, 2020

Document Properties

Client	Yam Finance
Title	Smart Contract Audit Report
Target	YAMv3 Proposal
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jeff Liu, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 23, 2020	Xuxian Jiang	Final Release
0.2	October 20, 2020	Xuxian Jiang	Additional Findings
0.1	October 18, 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 **YAMv3 Proposal**, 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 YAMv3 Proposal

YAM started in August 2020 as an experimental protocol of elastic supply cryptocurrency and community-based governance. Some of the design goals of YAM protocol include elastic token supply to achieve token price stability, a community-controlled governable treasury, and fair distribution mechanism to incentivize community participation of mining and governance. The protocol has been successfully migrated from earlier versions to current `YAMv3` and this audited proposal further amends the logic in `YAMv3` with the new voting support with staked assets and the stablecoin-referenced pricing of `YAMv3` tokens.

The basic information of YAMv3 Proposal is as follows:

Table 1.1: Basic Information of YAMv3 Proposal

Item	Description
Issuer	Yam Finance
Website	https://yam.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 23, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this repository contains a number of sub-directories (e.g., `reserves`, `token`, and `rebaser`) and this audit covers only the `tests/proposal_round_2` sub-directory.

- https://github.com/yam-finance/yamV3.git (d9bf1db)

1.2 About PeckShield

PeckShield Inc. [18] 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 [13]:

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

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

Table 1.3: The Full List of Check Items

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [12], 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 1.4: 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 YAMv3 Proposal. 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	1	■
Medium	1	■
Low	2	■ ■
Informational	2	■ ■
Total	6	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, 2 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key YAMv3 Proposal Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Unexpected Kick-Off of YAMIncentivizerWith-Voting	Security Features	Fixed
PVE-002	Low	Inconsistent RewardAdded Amounts in YAM-IncentivizerWithVoting	Error Conditions, Return Values, Status Codes	Fixed
PVE-003	Medium	Stale lastUpdateTime With Inaccurate Calculation of rewardPerToken()	Business Logics	Fixed
PVE-004	Low	Consistent Handling of minBlockBeforeVoting	Business Logics	Fixed
PVE-005	Informational	Gas Optimization in removeIncentivizer()	Coding Practices	Fixed
PVE-006	Informational	Simplified Logic in getReward()	Business Logics	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 Unexpected Kick-Off of YAMIncentivizerWithVoting

- ID: PVE-001
- Severity: High
- Likelihood: Low
- Impact: High
- Target: `YAMIncentivizerWithVoting`
- Category: Security Features [8]
- CWE subcategory: CWE-284 [2]

Description

The implemented `YAMv3` proposal includes a revised incentivizer logic that rewards early users with newly minted `YAM` tokens. In essence, by inheriting the basic functionality from the original `Synthetix` reward pool, this proposal further adds the support of allowing staking users to make use of the voting power associated with their staked assets.

To elaborate, we show below the code snippet of the `stake()` routine in the `YAMIncentivizerWithVoting` contract that implements the revised incentivizer logic. 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 and checkpoints in its internal records.

```
    // stake visibility is public as overriding LPTokenWrapper's stake() function
    function stake(uint256 amount) public updateReward(msg.sender) checkhalve {
        require(amount > 0, "Cannot stake 0");
        super.stake(amount);
        emit Staked(msg.sender, amount);
    }
```

Listing 3.1: YAMIncentivizerWithVoting.sol

```
    function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        uint256 new_bal = _balances[msg.sender].add(amount);
        _balances[msg.sender] = new_bal;
        address delegate = delegates[msg.sender];
        if (delegate == address(0)) {
```

```
            delegates[msg.sender] = msg.sender;
            delegate = msg.sender;
        }
        _moveDelegates(address(0), delegate, amount);
        _writeSupplyCheckpoint();
        uni_lp.safeTransferFrom(msg.sender, address(this), amount);
    }
```

Listing 3.2: YAMIncentivizerWithVoting.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 (breaker) {
          // do nothing
        } else if (block.timestamp >= periodFinish) {
            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);

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

Listing 3.3: YAMIncentivizerWithVoting.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 `Yam` tokens and activate the token distribution process. This is apparently unintended and violates the proposal design.

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 && initialized) {
            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);
```

```
989             periodFinish = block.timestamp.add(DURATION);
990             emit RewardAdded(initreward);
991         }
992         _;
993     }
```

Listing 3.4: YAMIncentivizerWithVoting.sol (revised)

Status This issue has been fixed in the commit: d2b647510da78aa7f3ab4f5262d6a13b4d1970c4.

3.2 Inconsistent RewardAdded Amounts in YAMIncentivizerWithVoting

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: N/A
- Target: `YAMIncentivizerWithVoting`
- Category: Status Codes [11]
- CWE subcategory: CWE-682 [4]

Description

As mentioned in Section 3.1, the `YAMv3` proposal includes a revised incentivizer logic that rewards early users with newly minted `YAM` tokens. Our analysis shows that there are 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 1014–1024); The second occasion is the explicit injection of rewards via `notifyRewardAmount(reward)` (lines 1003 – 1012) 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 1024 (event I), 1012 (event II), and 989 (event III) respectively. For illustration, we show the related routines below.

```
994     function notifyRewardAmount(uint256 reward)
995         external
996         onlyRewardDistribution
997         updateReward(address(0))
998     {
999         // https://sips.synthetix.io/sips/sip-77
1000        // increased buffer for scaling factor ( supports up to 10**4 * 10**18 scaling
            factor)
1001        require(reward < uint256(-1) / 10**22, "rewards too large, would lock");
1002        if (block.timestamp > starttime && initialized) {
1003          if (block.timestamp >= periodFinish) {
1004              rewardRate = reward.div(DURATION);
```

```
1005             } else {
1006                 uint256 remaining = periodFinish.sub(block.timestamp);
1007                 uint256 leftover = remaining.mul(rewardRate);
1008                 rewardRate = reward.add(leftover).div(DURATION);
1009             }
1010             lastUpdateTime = block.timestamp;
1011             periodFinish = block.timestamp.add(DURATION);
1012             emit RewardAdded(reward);
1013         } else {
1014             // increased buffer for scaling factor
1015             require(initreward < uint256(-1) / 10**22, "rewards too large, would lock");
1016             require(!initialized, "already initialized");
1017             initialized = true;
1018             uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
1019             uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
1020             yam.mint(address(this), newRewards);
1021             rewardRate = initreward.div(DURATION);
1022             lastUpdateTime = starttime;
1023             periodFinish = starttime.add(DURATION);
1024             emit RewardAdded(newRewards);
1025         }
1026     }
```

Listing 3.5: YAMIncentivizerWithVoting.sol

```
978     modifier checkhalve() {
979         if (breaker) {
980             // do nothing
981         } else if (block.timestamp >= periodFinish) {
982             initreward = initreward.mul(90).div(100);
983             uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
984             uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
985             yam.mint(address(this), newRewards);
986 
987             rewardRate = initreward.div(DURATION);
988             periodFinish = block.timestamp.add(DURATION);
989             emit RewardAdded(initreward);
990         }
991         _;
992     }
```

Listing 3.6: YAMIncentivizerWithVoting.sol

Our analysis shows that the above three events are emitted without reward amounts denominated at the same scale. Specifically, the event I (line 1024) records the reward amount, i.e., `newRewards = initreward.mul(scalingFactor).div(10**18)`, with the `scalingFactor` included; The event II (line 1012) emits the given reward amount simply from the given parameter to the `notifyRewardAmount()` routine without the `scalingFactor` included; The event III (line 989) records the decayed reward amount, i.e., `initreward = initreward.mul(90).div(100)`, again without taking into account the `scalingFactor`. As a result, current implementation unnecessarily introduces inconsistency in the emitted events.

Recommendation Be consistent in the emitted `RewardAdded` events on whether taking the `scalingFactor` into account in order to better facilitate off-chain analytics and reporting tools. An example revision to the `notifyRewardAmount` routine can be found as follows and this revision does not take `scalingFactor` into account.

```
    function notifyRewardAmount(uint256 reward)
        external
        onlyRewardDistribution
        updateReward(address(0))
    {
        // https://sips.synthetix.io/sips/sip-77
        // increased buffer for scaling factor ( supports up to 10**4 * 10**18 scaling
            factor)
        require(reward < uint256(-1) / 10**22, "rewards too large, would lock");
        if (block.timestamp > starttime && initialized) {
          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 {
          // increased buffer for scaling factor
          require(initreward < uint256(-1) / 10**22, "rewards too large, would lock");
          require(!initialized, "already initialized");
          initialized = true;
          uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
          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(initreward);
        }
    }
```

Listing 3.7: YAMIncentivizerWithVoting.sol (revised)

Status This issue has been fixed in the commit: 5159bf7df78137a34d4d2a33baa41c046114f8b6.

3.3 Stale lastUpdateTime With Inaccurate Calculation of rewardPerToken()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YAMIncentivizerWithVoting`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

The reward distribution process, by design, requires real-time updates on a number of interwoven parameters, e.g., `rewardRate`, `lastUpdateTime`, and `periodFinish`. In each occasion that may bring additional reward amounts into the pool (Section 3.2), these parameters need to be timely and accurately updated.

We have examined the execution logics of the three occasions that inject new rewards into the pool and our analysis shows that there is one particular occasion where a specific parameter has not been timely updated. In the following, we examine this particular occasion, i.e., the third occasion.

To elaborate, we show below the related code snippet of the third occasion.. This occasion happens after the previous reward period expires. We note that both `rewardRate` and `periodFinish` are properly updated (lines 987 – 988), but not `lastUpdateTime`.

```
978     modifier checkhalve() {
979         if (breaker) {
980           // do nothing
981         } else if (block.timestamp >= periodFinish) {
982             initreward = initreward.mul(90).div(100);
983             uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
984             uint256 newRewards = initreward.mul(scalingFactor).div(10**18);
985             yam.mint(address(this), newRewards);
986
987             rewardRate = initreward.div(DURATION);
988             periodFinish = block.timestamp.add(DURATION);
989             emit RewardAdded(initreward);
990         }
991         _;
992     }
```

Listing 3.8: YAMIncentivizerWithVoting.sol

A stale `lastUpdateTime` adversely affects the calculation of `rewardPerToken()` that is in charge of determining current reward amount per staked token. Therefore, in this case, the `rewardPerToken()` routine may calculate inaccurate gains for each staking user. In fact, without timely updating lastUpdateTime, the users who withdraw earlier from the pool are favorably treated with more rewards at the cost of later users who withdraw last.

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

Listing 3.9: YAMIncentivizerWithVoting.sol

Recommendation Timely update the lastUpdateTime parameter in the third occasion, i.e., checkhalve. An example revision can be found as follows:

```
    modifier checkhalve() {
        if (breaker) {
          // do nothing
        } else if (block.timestamp >= periodFinish && initialized) {
            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 3.10: YAMIncentivizerWithVoting.sol (revised)

Status This issue has been fixed in the commit: dc4f286d52011fb628fcfe4346090b80dc2be1bb.

3.4 Consistent Handling of minBlockBeforeVoting

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LPTokenWrapper`
- Category: Business Logics [10]
- CWE subcategory: CWE-837 [6]

Description

To facilitate the reward distribution and support voting with staked assets, each incentivizer has been configured with two new parameters, i.e., `minBlockSet` and `minBlockBeforeVoting`. The first parameter signals the proper initialization of the second parameter, which specifies the minimal block number for voting. A dedicate routine, i.e., `setMinBlockBeforeVoting()`, has been designed to initialize these two parameters.

```
    function setMinBlockBeforeVoting(uint256 blockNum)
        external
    {
        // only gov
        require(msg.sender == owner(), "!governance");
        require(!minBlockSet, "minBlockSet");
        minBlockBeforeVoting = blockNum;
        minBlockSet = true;
    }
```

Listing 3.11: LPTokenWrapper.sol

Our analysis shows these two parameters are not properly enforced. In order to support voting with staked assets, there are two associated helper routines, i.e., `getPriorVotes()` and `getCurrentVotes()`. In the following, we show the code snippet of `getPriorVotes()`.

```
    function getPriorVotes(address account, uint256 blockNumber)
        public
        view
        returns (uint256)
    {
        require(blockNumber < block.number, "Incentivizer::_getPriorLPStake: not yet
            determined");
        if (blockNumber < minBlockBeforeVoting) {
            return 0;
        }
        // get incentivizer's uniswap pool yam votes
        uint256 poolVotes = YAM(address(yam)).getPriorVotes(address(uni_lp), blockNumber
            );

        // get prior stake
        uint256 priorStake = _getPriorLPStake(account, blockNumber);
```

```
741
742         // get prior LP stake
743         uint256 lpTotalSupply = getPriorSupply(blockNumber);
744
745         // get percent ownership of staked LPs
746         uint256 percentOfVote = priorStake.mul(BASE).div(lpTotalSupply);
747
748         // votes * percentage / percentage max
749         // note: this will overestimate the number of votes based on
750         //       % of LP pool tokens staked here
751         return poolVotes.mul(percentOfVote).div(BASE);
752     }
```

Listing 3.12: LPTokenWrapper.sol

The `getPriorVotes()` routine has indeed verified the case on `if(blockNumber < minBlockBeforeVoting)return 0` (lines 733 – 735). However, it needs to be used together with `minBlockSet` as follows: `if(!minBlockSet || block.number < minBlockBeforeVoting)return 0;`.

The current `getCurrentVotes()` routine does not validate with these two parameters and need to be revised similarly in `getPriorVotes()`.

Recommendation Resolve the inconsistency in enforcing the above incentivizer-related configuration parameters. In the following, we show one example revision:

```
727     function getPriorVotes(address account, uint256 blockNumber)
728         public
729         view
730         returns (uint256)
731     {
732         require(blockNumber < block.number, "Incentivizer::_getPriorLPStake: not yet
                determined");
733         if (!minBlockSet  blockNumber < minBlockBeforeVoting) {
734             return 0;
735         }
736         // get incentivizer's uniswap pool yam votes
737         uint256 poolVotes = YAM(address(yam)).getPriorVotes(address(uni_lp), blockNumber
                );
738
739         // get prior stake
740         uint256 priorStake = _getPriorLPStake(account, blockNumber);
741
742         // get prior LP stake
743         uint256 lpTotalSupply = getPriorSupply(blockNumber);
744
745         // get percent ownership of staked LPs
746         uint256 percentOfVote = priorStake.mul(BASE).div(lpTotalSupply);
747
748         // votes * percentage / percentage max
749         // note: this will overestimate the number of votes based on
750         //
```

Listing 3.13: LPTokenWrapper.sol

Status This issue has been fixed in two related commits: f113397e and 20e3e55.

3.5 Gas Optimization in removeIncentivizer()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DualGovernorAlpha
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [3]

Description

The revised governance contract, i.e., DualGovernorAlpha, supports the dynamic addition and removal of incentivizers that allow for the voting support for users with staked assets. As there may have a number of incentivizers for inclusion, the implementation maintains an array for all supported incentivizers.

While reviewing the support of incentivizers, we notice the removal of certain element indexed by index from the incentivizers array could benefit from known best practice in saving gas cost. Especially, when we have a large array, the improvement could save a lot of gas!

```
169     function removeIncentivizer(uint256 index)
170         public
171     {
172         require(msg.sender == address(timelock), "GovernorAlpha::!timelock");
173         if (index >= incentivizers.length) return;
174         for (uint i = index; i < incentivizers.length-1; i++) {
175             incentivizers[i] = incentivizers[i+1];
176         }
177         incentivizers.length--;
178     }
```

Listing 3.14: DualGovernorAlpha.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 there is a need to walk through a huge array and replace each element with the next element as in current implementation (lines 174 – 176).

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

```
    function removeIncentivizer(uint256 index)
        public
    {
        require(msg.sender == address(timelock), "GovernorAlpha::!timelock");
```

```
        if (index >= incentivizers.length) return;

        if (index != incentivizers.length-1) {
            incentivizers[index] = incentivizers[incentivizers.length -1];
        }
        incentivizers.length--;
    }
```

Listing 3.15: DualGovernorAlpha.sol

Status This issue has been fixed in the commit: be04d4fe93581b0817d68a882d8ae35e4660e773.

3.6 Simplified Logic in getReward()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YAMIncentivizerWithVoting`
- Category: Business Logics [10]
- CWE subcategory: CWE-770 [5]

Description

In the `YAMIncentivizerWithVoting` 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 917).

```
    function getReward() public updateReward(msg.sender) checkhalve {
        uint256 reward = earned(msg.sender);
        if (reward > 0) {
            rewards[msg.sender] = 0;
            uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
            uint256 trueReward = reward.mul(scalingFactor).div(10**18);
            yam.safeTransfer(msg.sender, trueReward);
            emit RewardPaid(msg.sender, trueReward);
        }
    }
```

Listing 3.16: YAMIncentivizerWithVoting.sol

```
913    modifier updateReward(address account) {
914        rewardPerTokenStored = rewardPerToken();
915        lastUpdateTime = lastTimeRewardApplicable();
```

```
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }
```

Listing 3.17: YAMIncentivizerWithVoting.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 968).

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

```
967     function getReward() public updateReward(msg.sender) checkhalve {
968         uint256 reward = rewards[msg.sender];
969         if (reward > 0) {
970             rewards[msg.sender] = 0;
971             uint256 scalingFactor = YAM(address(yam)).yamsScalingFactor();
972             uint256 trueReward = reward.mul(scalingFactor).div(10**18);
973             yam.safeTransfer(msg.sender, trueReward);
974             emit RewardPaid(msg.sender, trueReward);
975         }
976     }
```

Listing 3.18: YAMIncentivizerWithVoting.sol

Status This issue has been fixed in the commit: ed1a49d5b4e62f3ec60eff4127bc4fc53d8c7f55.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the proposed amendments to `YAMv3`. We believe that the `YAM` protocol presents an interesting and novel experiment of on-chain community-based governance and elastic supply cryptocurrency, and we are very impressed by the overall design and implementation. This proposal follows the previous clean design 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 [14, 15, 16, 17, 19].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] 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-287: Improper Access Control. https://cwe.mitre.org/data/definitions/284.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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