SMART CONTRACT AUDIT REPORT

for

YAM FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

September 10, 2020

Document Properties

Client	Yam Finance
Title	Smart Contract Audit Report
Target	YAMv3
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	September 10, 2020	Xuxian Jiang	Final Release
1.0-rc2	September 9, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	September 8, 2020	Xuxian Jiang	Release Candidate #1
0.2	September 5, 2020	Xuxian Jiang	Add More Findings
0.1	September 3, 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 **YAMv3** design document and related smart contract source code, 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

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 started as `YAMv1`, then `YAMv2`, and currently is upgrading to `YAMv3` with new additions or updates on modifying the reserve asset to `yUSD`, and changing the voting period length and thresholds for proposal and quorum.

The basic information of YAMv3 is as follows:

Table 1.1: Basic Information of YAMv3

Item	Description
Issuer	Yam Finance
Website	https://yam.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 10, 2020

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

- https://github.com/yam-finance/yamV3 (004425f)

1.2 About PeckShield

PeckShield Inc. [24] 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 [19]:

- 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) [18], 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 YAMv3 implementation. During the first phase of our audit, we studied the smart contract source code and ran 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	4	■■■■
Low	6	■■■■■■
Informational	4	■■■■
Total	15	

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, 4 medium-severity vulnerabilities, 6 low-severity vulnerabilities and 4 informational recommendations.

Table 2.1: Key YAMv3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Inaccurate Vested Percentage Calculation in Migrator	Business Logics	Fixed
PVE-002	Medium	Explicit Wrappers for Yam Conversions	Coding Practices	Fixed
PVE-003	Informational	Improved Extra Hop Unwrapping in Delegated Calls	Coding Practices	Fixed
PVE-004	Informational	Removal of Redundant Code	Coding Practices	Fixed
PVE-005	Low	Blocked Rebasing From Possible Price Fluctuation	Coding Practices	Fixed
PVE-006	Low	Improved Precision By Multiplication-Before-Division	Numeric Errors	Fixed
PVE-007	Medium	Improved Sanity Checks When Updating Important System Parameters	Error Conditions, Return Values, Status Codes	Fixed
PVE-008	Informational	Gas Optimization in removeUniPair() And removeBalPair()	Business Logics	Fixed
PVE-009	Low	Inconsistency Between Documented and Implemented Incentivized Pool Duration	Business Logics	Fixed
PVE-010	Medium	Potential Blocking of Initial Reward Drop	Business Logics	Fixed
PVE-011	Medium	Oversized Rewards May Lock All Pool Stakes	Numeric Errors	Fixed
PVE-012	Informational	Incompatibility With Deflationary Tokens For Staking And reserveToken	Business Logics	Confirmed
PVE-013	Low	Full Charge of Proposal Execution Cost From Accompanying msg.value	Business Logics	Confirmed
PVE-014	Low	Improved Handling of Corner Cases in Proposal Submission	Error Conditions, Return Values, Status Codes	Fixed
PVE-015	Low	Improved pendingAdmin Protection in TimeLock	Security Features	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Inaccurate Vested Percentage Calculation in Migrator

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Migrator`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [12]

Description

The `YAMv2`-to-`YAMv3` migration logic is implemented according to the recent community-based governance and consensus [2]. Specifically, *"YAMv2 to YAMv3 will be a 1:1 migration with no deadline, with 50% immediately redeemable and 50% continuously vested over 30 days."*

The actual bulk work of migration is performed by the `Migrator` contract in a function named `migrate()` (as shown in the related code snippet below). It proceeds by firstly determining the vested percentage, then querying the calling user's `YAMv2` balance and calculating the vested amount, next burning the `YAMv2` balance (by essentially transferring the balance to an address no one has access to the corresponding private key), and finally minting vested `YAMv3` amount to the user. To properly kick-off the migration process, the smart contract ensures that the migration process takes place only after the `started` conditions are met.

```
    function migrate()
        external
        started
    {
        // completion percentage of vesting
        uint256 vestedPerc = now.sub(startTime).mul(BASE).div(vestingDuration);

        // completion percentage of delegator vesting
        uint256 delegatorVestedPerc = now.sub(startTime).mul(BASE).div(
            delegatorVestingDuration);

        // gets the yamValue for a user.
```

```
120         uint256 yamValue = YAMv2(yamV2).balanceOf(_msgSender());
121 
122         // half is instant redeemable
123         uint256 halfRedeemable = yamValue / 2;
124 
125         uint256 mintAmount;
126 
127         ...
128 
129         // BURN YAMv2 - UNRECOVERABLE.
130         SafeERC20.safeTransferFrom(
131             IERC20(yamV2),
132             _msgSender(),
133             address(0x000000000000000000000000000000000000dEaD),
134             yamValue
135         );
136 
137         // mint, this is in raw internalDecimals. Handled by updated _mint function
138         YAMv3(yamV3).mint(_msgSender(), mintAmount);
139     }
```

Listing 3.1: Migrator.sol

Our analysis shows that there is an issue when determining the vested percentage, which cascadingly affects the calculation of vested and minted amounts. Specifically, the vested percentage, i.e., `vestedPerc`, is calculated as: `vestedPerc = now.sub(startTime).mul(BASE).div(vestingDuration)` (line 114). It fails to take into account the entire vesting period is between `startTime` and `startTime +vestingDuration`. As a result, it could lead to the undesirable situation of having `vestedPerc > 100%`, resulting in over-minting of `YAMv3` to the calling user. A correct calculation could be the following: `vestedPerc = min(now, startTime.add(vestingDuration)).sub(startTime).mul(BASE).div (vestingDuration)`.

The calculation of the vested percentage of delegator rewards, i.e., `delegatorVestedPerc`, shares the same issue. In addition, there are two other functions `vested()` and `claimVested()` with the same pattern as `migrate()`, warranting revisions as well.

Recommendation Take into account the vesting duration and ensure the vested percentage is never larger than 100% (scaled to `BASE`). In total, there are three functions, i.e., `migrate()`, `vested()`, and `claimVested()`, that need to be adjusted.

Status This issue has been fixed in this commit: 6a505e3b7d896e5ef84da69a4c287e5d15120dec.

3.2 Explicit Wrappers for Yam Conversions

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact:High
- Target: YAM
- Category: Coding Practices [14]
- CWE subcategory: CWE-1117 [4]

Description

YAM is a great experimental protocol that builds up an elastic supply cryptocurrency by effectively integrating recent progresses and innovations in programmable money and governance. The YAMv1 setback was largely due to a bug in its inherent rebasing function and we have been seriously thinking of the root cause (on how it might be introduced) as well as possible counter-measures (that could eliminate similar bugs from being introduced in the first place).

Our analysis shows that this bug is largely introduced by not fully delineating the discrepancy between the internal YAM storage amount (in _yamBalances) and the external value amount as well as the lack of their explicit inter-conversions. The follow-up medium article titled *"SAVE YAM!"* [3] also misdiagnosed the issue by suggesting a (wrong) fix, i.e., `totalSupply = initSupply.mul(yamsScalingFactor ).div(BASE)`, for the very same reason.

In order to fully delineate their discrepancy and solve this issue once and for all, we would like to take a pro-active approach by proposing two new explicit wrappers that are responsible for converting from one to another (and vice versa). Moreover, any conversion needs to go through these two new wrappers. In other words, no other conversion (including a direct one) would be allowed!

With that, we define two following wrappers: `yamToFragment()` and `fragmentToYam()`. After these two wrappers are explicitly defined, we can then strictly enforce that their inter-conversions can only be performed by these two wrappers and there is no exception! By doing so, we can effectively eliminate this issue.

```
    /** @dev An explicit wrapper to convert the internal Yam storage amount to its
        external value.
    * @param yam The yam amount to convert.
    * @return The corresponding external value.
    */
    function yamToFragment(uint256 yam)
      external
      view
      returns (uint256)
    {
      return yam.mul(yamsScalingFactor).div(internalDecimals);
    }

```

```
    /** @dev An explicit wrapper to convert the external Yam value to its internal
        storage amount.
    * @param value The external Yam value to convert.
    * @return The corresponding internal storage amount.
    */
    function fragmentToYam(uint256 value)
      external
      view
      returns (uint256)
    {
      return value.mul(internalDecimals).div(yamsScalingFactor);
    }
```

Listing 3.2: YAM.sol

After the introduction of the above two wrappers, we note that the current codebase can still be benefited by having a stronger confidence in eliminating similar risks. In the following, we use the `initialize()` routine of the `YAM` contract as an example. For elaboration, we show the related code snippet below.

```
    /**
     * @notice Initialize the new money market
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     */
    function initialize(
        string memory name_,
        string memory symbol_,
        uint8 decimals_,
        address initial_owner,
        uint256 initSupply_
    )
        public
    {
        super.initialize(name_, symbol_, decimals_);

        initSupply = initSupply_.mul(10**24/ (BASE));
        totalSupply = initSupply_;
        yamsScalingFactor = BASE;
        _yamBalances[initial_owner] = initSupply_.mul(10**24 / (BASE));

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name)),
                getChainId(),
                address(this)
            )
        );
```

```
    }
```

Listing 3.3: YAM.sol

The `initialize()` routine takes five arguments to prepare the `YAM` token and the last parameter `initSupply_` specifies its initial total supply. Note that throughout the `YAM` contract, `initSupply` represents the sum of internal `YAM` storage amount. For naming consistency, it is strongly suggested to replace the last argument with `totalSupply_`. Otherwise, the discrepancy between the two is still blurred! With the above new wrappers, we can accordingly revise this `initialize()` routine with much needed clarification! The revision is shown below:

```
    /**
     * @notice Initialize the new money market
     * @param name_ ERC-20 name of this token
     * @param symbol_ ERC-20 symbol of this token
     * @param decimals_ ERC-20 decimal precision of this token
     */
    function initialize(
        string memory name_,
        string memory symbol_,
        uint8 decimals_,
        address initial_owner,
        uint256 totalSupply_
    )
        public
    {
        super.initialize(name_, symbol_, decimals_);

        yamsScalingFactor = BASE;
        totalSupply = totalSupply_;
        initSupply = fragmentToYam(totalSupply_);
        _yamBalances[initial_owner] = initSupply;

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name)),
                getChainId(),
                address(this)
            )
        );
    }
```

Listing 3.4: YAM.sol (revised)

Regarding the severity rating of this issue, we normally would consider `Informational`. However, considering the impact of past incident and the purpose of completely eliminating this issue, we would like to escalate this issue to `Medium`.

Recommendation Add the two explicit wrappers and accordingly revise the affected functions, including `initialize()`, for improved clarity and elimination of similar issues.

Status This issue has been fixed in the commit: a1a95ff4400776fba1b3b9cba841330d6f807dde.

3.3 Improved Extra Hop Unwrapping in Delegated Calls

- ID: PVE-003
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `YAMDelegate`, `YAMRebaser`
- Category: Coding Practices [14]
- CWE subcategory: CWE-563 [8]

Description

The `YAMDelegator` contract behaves as the proxy by relaying calls to the backend logic contract `YAMDelegate`. The call-relaying is mainly implemented by two helper routines: `delegateAndReturn()` and `delegateToViewAndReturn()`. The first one mainly relay external calls that may inflict state changes while the second one is mainly for getter-related calls without causing any state change.

We notice that the `delegateToViewAndReturn()` implementation (as shown below) returns results or forwards reverts to its caller. However, as it relays the call by making a `staticcall` call to itself, hence bringing an extra hop in the call chain. Note that each extra hop will introduce additional two `uint256` integers as the prefix of the wrapper `returndata`. In order to ensure the returned results are intact, we accordingly need to remove the two-`uint256`-integers prefix before returning back to the caller. The current implementation properly adjusts the offset of return bytes, i.e., `return(add(free_mem_ptr, 0x40), returndatasize)` (line 430). However, its length also needs to reduce the two `uint256` integers as follows, i.e., `return(add(free_mem_ptr, 0x40), returndatasize-0x40)`.

```
421     function delegateToViewAndReturn() private view returns (bytes memory) {
422         (bool success, ) = address(this).staticcall(abi.encodeWithSignature("
                delegateToImplementation(bytes)", msg.data));
423
424         assembly {
425             let free_mem_ptr := mload(0x40)
426             returndatacopy(free_mem_ptr, 0, returndatasize)
427
428             switch success
429             case 0 { revert(free_mem_ptr, returndatasize) }
430             default { return(add(free_mem_ptr, 0x40), returndatasize) }
431         }
432     }
433
434     function delegateAndReturn() private returns (bytes memory) {
435         (bool success, ) = implementation.delegatecall(msg.data);
436
```

```
        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(free_mem_ptr, returndatasize) }
        }
    }
```

Listing 3.5: YAMDelegator.sol

Recommendation Unwrap the extra call by accordingly reducing the `returndatasize` as well (in addition to adjusting the offset of returned bytes in `free_mem_ptr`).

```
    function delegateToViewAndReturn() private view returns (bytes memory) {
        (bool success, ) = address(this).staticcall(abi.encodeWithSignature("
            delegateToImplementation(bytes)", msg.data));

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(add(free_mem_ptr, 0x40), returndatasize-0x40) }
        }
    }

    function delegateAndReturn() private returns (bytes memory) {
        (bool success, ) = implementation.delegatecall(msg.data);

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(free_mem_ptr, returndatasize) }
        }
    }
```

Listing 3.6: YAMDelegator.sol

Status This issue has been fixed in this commit: 970ffb763d3d653c7ccefed342928b6ae6a65672.

3.4 Removal of Redundant Code

- ID: PVE-004
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `YAMDelegate`, `YAMRebaser`
- Category: Coding Practices [14]
- CWE subcategory: CWE-563 [8]

Description

As mentioned in Section 3.3, the `YAMDelegator` contract behaves as the proxy by relaying calls to the backend logic contract `YAMDelegate`. Accordingly, the proxy contract needs to keep current logic contract, i.e., `implementation`.

In the following, we observe that both `YAMDelegatorInterface` and `YAMDelegateInterface` inherit from `YAMDelegationStorage`, which has the implementation address for this contract. And `YAMDelegator` and `YAMDelegate` inherit from `YAMDelegatorInterface` and `YAMDelegateInterface` respectively. In other words, `YAMDelegatorInterface` indeed requires the the implementation address, but `YAMDelegateInterface` does not actually use it (line 27). With that, we may consider a removal of the inheritance of `YAMDelegationStorage` in `YAMDelegateInterface`.

```
5  contract YAMDelegationStorage {
6      /**
7       * @notice Implementation address for this contract
8       */
9      address public implementation;
10 }
11
12 contract YAMDelegatorInterface is YAMDelegationStorage {
13     /**
14      * @notice Emitted when implementation is changed
15      */
16     event NewImplementation(address oldImplementation, address newImplementation);
17
18     /**
19      * @notice Called by the gov to update the implementation of the delegator
20      * @param implementation_ The address of the new implementation for delegation
21      * @param allowResign Flag to indicate whether to call _resignImplementation on the
           old implementation
22      * @param becomeImplementationData The encoded bytes data to be passed to
           _becomeImplementation
23      */
24     function _setImplementation(address implementation_, bool allowResign, bytes memory
           becomeImplementationData) public;
25 }
26
27 contract YAMDelegateInterface is YAMDelegationStorage {
```

```
    /**
     * @notice Called by the delegator on a delegate to initialize it for duty
     * @dev Should revert if any issues arise which make it unfit for delegation
     * @param data The encoded bytes data for any initialization
     */
    function _becomeImplementation(bytes memory data) public;

    /**
     * @notice Called by the delegator on a delegate to forfeit its responsibility
     */
    function _resignImplementation() public;
}
```

Listing 3.7: YAMDelegate.sol

We also observe a few redundant code. One example is that `reservesContract` has been assigned twice with the same value in the `constructor()` of the `YAMRebaser` contract. Also, a local variable named `supplyAfterRebase` in the `rebase()` routine of the same contract is initialized (line 444), but not used. The following assertion, i.e., `assert(yam.yamsScalingFactor()<= yam.maxScalingFactor())` (line 445), is redundant as the previous call to `yam.rebase(epoch, indexDelta, positive)` guarantees the requirement is always met.

```
398     function rebase()
399         public
400     {
401         // EOA only or gov
402         require(msg.sender == tx.origin  msg.sender == gov);
403         // ensure rebasing at correct time
404         _inRebaseWindow();
405
406         // This comparison also ensures there is no reentrancy.
407         require(lastRebaseTimestampSec.add(minRebaseTimeIntervalSec) < now);
408
409         // Snap the rebase time to the start of this window.
410         lastRebaseTimestampSec = now.sub(
411             now.mod(minRebaseTimeIntervalSec)).add(rebaseWindowOffsetSec);
412
413         epoch = epoch.add(1);
414
415         // get twap from uniswap v2;
416         uint256 exchangeRate = getTWAP();
417
418         // calculates % change to supply
419         (uint256 offPegPerc, bool positive) = computeOffPegPerc(exchangeRate);
420
421         uint256 indexDelta = offPegPerc;
422
423         // Apply the Dampening factor.
424         indexDelta = indexDelta.div(rebaseLag);
425
426         YAMTokenInterface yam = YAMTokenInterface(yamAddress);
```

```

        if (positive) {
            require(yam.yamsScalingFactor().mul(BASE.add(indexDelta)).div(BASE) < yam.
                maxScalingFactor(), "new scaling factor will be too big");
        }


        uint256 currSupply = yam.totalSupply();

        uint256 mintAmount;
        // reduce indexDelta to account for minting
        if (positive) {
            uint256 mintPerc = indexDelta.mul(rebaseMintPerc).div(BASE);
            indexDelta = indexDelta.sub(mintPerc);
            mintAmount = currSupply.mul(mintPerc).div(BASE);
        }

        // rebase
        uint256 supplyAfterRebase = yam.rebase(epoch, indexDelta, positive);
        assert(yam.yamsScalingFactor() <= yam.maxScalingFactor());

        // perform actions after rebase
        afterRebase(mintAmount, offPegPerc);
    }
```

Listing 3.8: Revised YAMRebaser.sol

Recommendation Consider the removal of the unused code.

Status This issue has been confirmed and accordingly fixed in the following two commits: e7300d5b8bfdfa16b097f49b489672a73050d3aa and 4a0f3624103e4c5c6f6a969104f24ad4b6c36a2d. Note that `YAMDelegateInterface`'s inheritance from `YAMDelegationStorage` remains intact as `implementation` is used to avoid certain public functions from being marked `pure` in the `YAMDelegate` contract (lines 57 – 59).

3.5 Blocked Rebasing From Possible Price Fluctuation

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `YAMRebaser`
- Category: Coding Practices [14]
- CWE subcategory: CWE-563 [8]

Description

`YAM` is an experimental protocol that builds up an elastic supply cryptocurrency and the elastic supply capability is implemented by the `YAMRebaser` contract that measures current price fluctuation and then dynamically inflates or deflates the `YAM` 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 `YAM` and `reserveToken`. Based on recent community governance and consensus, `YAMv3` chooses `yUSD` as the `reserveToken`, instead of `yCRV` (used in `YAMv1`).

```
    /**
     * @notice Calculates TWAP from uniswap
     *
     * @dev When liquidity is low, this can be manipulated by an end of block -> next block
     *      attack. We delay the activation of rebases 12 hours after liquidity incentives
     *      to reduce this attack vector. Additional there is very little supply
     *      to be able to manipulate this during that time period of highest vuln.
     */
    function getTWAP()
        internal
        returns (uint256)
    {
      (uint 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, casting never truncates
       // cumulative price is in (uq112x112 price * seconds) units so we simply wrap it after division by time elapsed
       FixedPoint.uq112x112 memory priceAverage = FixedPoint.uq112x112(uint224((priceCumulative - priceCumulativeLast) / timeElapsed));

       priceCumulativeLast = priceCumulative;
       blockTimestampLast = blockTimestamp;

```

```
719         return FixedPoint.decode144(FixedPoint.mul(priceAverage, BASE));
720     }
```

Listing 3.9: YAMRebaser.sol

For elaboration, we show above the `getTWAP()` routine that is responsible for reading the `TWAP` from the chosen `UniswapV2` trading pair. Specifically, it 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. However, an overflow could occur during the `priceAverage` calculation. Note that the resulting `priceAverage` (line 714) is represented as a `struct uq112x112 {uint224 _x}` data structure, which has the maximum 224 bits. When it is multiplied with `BASE=10**18`, the calculation in line 719, i.e., `FixedPoint.decode144(FixedPoint.mul(priceAverage, BASE))`, may overflow and cause revert!

```
    // multiply a UQ112x112 by a uint, returning a UQ144x112
    // reverts on overflow
    function mul(uq112x112 memory self, uint y) internal pure returns (uq144x112 memory)
         {
        uint z;
        require(y == 0  (z = uint(self._x) * y) / y == uint(self._x), "FixedPoint:
            MULTIPLICATION_OVERFLOW");
        return uq144x112(z);
    }
```

Listing 3.10: FixedPoint.sol

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

Recommendation Accommodate the price fluctuation without blocking rebasing operations.

Status This issue has been fixed in this commit: 6e56cf61ac45dbc4efecc2f7d09ced4ee62aa550.

3.6 Improved Precision By Multiplication-Before-Division

- ID: PVE-006
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `YAMRebaser`
- Category: Numeric Errors [17]
- CWE subcategory: CWE-190 [5]

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 `uniswapMaxSlippage()` as an example. This routine is used to calculate the maximum trade amount under the given slippage in the `UniswapV2` trading pair of `YAM` and `reserveToken`.

```
598    function uniswapMaxSlippage(
599        uint256 token0,
600        uint256 token1,
601        uint256 offPegPerc
602    )
603      internal
604      view
605      returns (uint256)
606    {
607        if (isToken0) {
608          if (offPegPerc >= 10**17) {
609              // cap slippage
610              return token0.mul(maxSlippageFactor).div(BASE);
611          } else {
612              // in the 5-10% off peg range, slippage is essentially 2*x (where x is
                      percentage of pool to buy).
613              // all we care about is not pushing below the peg, so underestimate
614              // the amount we can sell by dividing by 3. resulting price impact
615              // should be ~= offPegPerc * 2 / 3, which will keep us above the peg
616              //
617              // this is a conservative heuristic
618              return token0.mul(offPegPerc / 3).div(BASE);
619          }
620        } else {
621            if (offPegPerc >= 10**17) {
622                return token1.mul(maxSlippageFactor).div(BASE);
623            } else {
624                return token1.mul(offPegPerc / 3).div(BASE);
625            }
626        }
627    }
```

Listing 3.11: YAMRebaser.sol

We notice the calculation of the trade amount of `YAM` (line 618) involves the multiplication of the devision result of `offPegPerc / 3`. For improved precision, it is better to calculate the multiplication before the division, i.e., `token0.mul(offPegPerc).div(BASE.mul(3))`. Similarly, the calculation of line 624 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.

```
    function uniswapMaxSlippage(
        uint256 token0,
        uint256 token1,
```

```
        uint256 offPegPerc
    )
      internal
      view
      returns (uint256)
    {
        if (isToken0) {
          if (offPegPerc >= 10**17) {
              // cap slippage
              return token0.mul(maxSlippageFactor).div(BASE);
          } else {
              // in the 5-10% off peg range, slippage is essentially 2*x (where x is percentage of pool to buy).
              // all we care about is not pushing below the peg, so underestimate
              // the amount we can sell by dividing by 3. resulting price impact
              // should be ~= offPegPerc * 2 / 3, which will keep us above the peg
              //
              // this is a conservative heuristic
              return token0.mul(offPegPerc).div(BASE.mul(3));
          }
        } else {
            if (offPegPerc >= 10**17) {
                return token1.mul(maxSlippageFactor).div(BASE);
            } else {
                return token1.mul(offPegPerc).div(BASE.mul(3));
            }
        }
    }
```

Listing 3.12: YAMRebaser.sol

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the commit: b0e6cce3bfb8403ca248ece767ccc056accbc3c0.

3.7 Improved Sanity Checks When Updating Important System Parameters

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `YAMRebaser`
- Category: Status Codes [16]
- CWE subcategory: CWE-391 [7]

Description

As mentioned in Section 3.5, the `YAMRebaser` contract plays a critical role in dynamically rebasing the total supply of `YAM` based on the fluctuating market price. The rebasing capability requires a number of delicate system-reconfigurable parameters, e.g., `maxSlippageFactor`, `rebaseMintPerc`, and `minRebaseTimeIntervalSec`. Note that `maxSlippageFactor` specifies the allowable slippage of `YAM` trading price when inflating its total supply. The `rebaseMintPerc` parameter controls the percentage of minted `YAM` when the scaling factor of `YAM` is changed. The `minRebaseTimeIntervalSec` parameter governs the rebasing interval.

However, we notice that the updates of many of these important system parameters do not have necessary sanity checks in place. As an example, we show here the two helper routines that adjust `maxSlippageFactor` and `rebaseMintPerc`. Note that if `rebaseMintPerc` is not set properly, say more than 100%, every rebasing attempt would fail.

```
    /**
    @notice Updates slippage factor
    @param maxSlippageFactor_ the new slippage factor
    *
    */
    function setMaxSlippageFactor(uint256 maxSlippageFactor_)
        public
        onlyGov
    {
        uint256 oldSlippageFactor = maxSlippageFactor;
        maxSlippageFactor = maxSlippageFactor_;
        emit NewMaxSlippageFactor(oldSlippageFactor, maxSlippageFactor_);
    }

    /**
    @notice Updates rebase mint percentage
    @param rebaseMintPerc_ the new rebase mint percentage
    *
    */
    function setRebaseMintPerc(uint256 rebaseMintPerc_)
        public
        onlyGov
```

```
    {
        uint256 oldPerc = rebaseMintPerc;
        rebaseMintPerc = rebaseMintPerc_;
        emit NewRebaseMintPercent(oldPerc, rebaseMintPerc_);
    }
```

Listing 3.13: YAMRebaser.sol

As 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. Currently, there is no sanity checks in place to ensure their correctness.

Note the `setRebaseTimingParameters()` routine has basic sanity checks in place and could be benefited by further ensuring the sum of `rebaseWindowOffsetSec` and `rebaseWindowLengthSec` will not fall outside the `minRebaseTimeIntervalSec`, i.e., `require(rebaseWindowOffsetSec_ + rebaseWindowLengthSec_ < minRebaseTimeIntervalSec_)`.

```
    function setRebaseTimingParameters(
        uint256 minRebaseTimeIntervalSec_,
        uint256 rebaseWindowOffsetSec_,
        uint256 rebaseWindowLengthSec_)
        external
        onlyGov
    {
        require(minRebaseTimeIntervalSec_ > 0);
        require(rebaseWindowOffsetSec_ < minRebaseTimeIntervalSec_);

        minRebaseTimeIntervalSec = minRebaseTimeIntervalSec_;
        rebaseWindowOffsetSec = rebaseWindowOffsetSec_;
        rebaseWindowLengthSec = rebaseWindowLengthSec_;
    }
```

Listing 3.14: YAMRebaser.sol

In addition, as these parameters control various aspects of system operation, the current implementation also emit most related events (though not all of them). Note the `setRebaseTimingParameters()` could be improved by emitting a related event as well.

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

Status This issue has been confirmed and accordingly fixed in the following two commits: 6f36f49b7285279e62e7db64f3de78725b6e1582 and 202620b48ffe62bfca491932bb4a326f742da8b2.

3.8 Gas Optimization in removeUniPair() And removeBalPair()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YAMRebaser.sol`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [12]

Description

`YAMv3` 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 (uint i = index; i < uniSyncPairs.length-1; i++){
            uniSyncPairs[i] = uniSyncPairs[i+1];
        }
        delete uniSyncPairs[uniSyncPairs.length-1];
        uniSyncPairs.length--;
    }

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

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

Listing 3.15: YAMRebaser.sol

The trick 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 what the current implementation is (line 125 – 127).

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

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

        uniSyncPairs[index] = uniSyncPairs[uniSyncPairs.length -1];
        uniSyncPairs.length--;
    }

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

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

Listing 3.16: YAMRebaser.sol (revised)

Status This issue has been fixed in the commit: 1c9ef962f5ff143a12df2221f70cb3dedbae53b0.

3.9 Inconsistency Between Documented and Implemented Incentivized Pool Duration

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `YAMIncentives`
- Category: Business Logics [15]
- CWE subcategory: CWE-837 [11]

Description

According to the documentation of YAMv3 [2], *"The YAMv3/yUSD Uniswap LP pool will receive awards totaling approximately 925k, with 92.5k distributed the first week and decreasing by 10% each following."*

As part of the audit process, we examine and identify possible inconsistency between the documentation/white paper and the implementation. Based on the smart contract code, there is a constant, i.e., `DURATION`. This particular constant is hard-coded as $625,000$ when the contract is being deployed and it can never be adjusted once deployed (even via a governance process).

A further analysis about the incentivized pool logic (implemented in the `YAMIncentives` contract) shows certain inconsistency that needs to adjusted. For elaboration, we show the related code snippet below.

In particular, the constant number `DURATION` essentially reflects the number of seconds in a week. However, if we calculate the number of seconds in a week, the number should be `NUMBER_OF_SECOND_IN_A_WEEK = 7 * 24 * 60 * 60 = 604,800`.

```
contract YAMIncentivizer is LPTokenWrapper, IRewardDistributionRecipient {
    IERC20 public yam = IERC20(0x0e2298E3B3390e3b945a5456fBf59eCc3f55DA16);
    uint256 public constant DURATION = 625000;

    uint256 public initreward = 925 * 10**2 * 10**18; // 92.5k
    uint256 public starttime = 1600560000 + 48 hours; // 2020-08-12 19:00:00 (UTC UTC +00:00)
    uint256 public periodFinish = 0;
    uint256 public rewardRate = 0;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;
    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    ...

    }
```

Listing 3.17: YAMIncentives.sol

As this `DURATION` number indeed controls the length of each iteration of `YAM` rewards that are distributed to various pools. It is important to get it as precise as possible.

Recommendation Resolve the inconsistency between the documentation and the implementation. In this particular case, it is more straightforward to adjust the code as follows:

```
contract YAMIncentivizer is LPTokenWrapper, IRewardDistributionRecipient {
    IERC20 public yam = IERC20(0x0e2298E3B3390e3b945a5456fBf59eCc3f55DA16);
    uint256 public constant DURATION = 604800;

    uint256 public initreward = 925 * 10**2 * 10**18; // 92.5k
    uint256 public starttime = 1600560000 + 48 hours; // 2020-08-12 19:00:00 (UTC UTC +00:00)
    uint256 public periodFinish = 0;
    uint256 public rewardRate = 0;
    uint256 public lastUpdateTime;
    uint256 public rewardPerTokenStored;
    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    ...

    }
```

Listing 3.18: YAMIncentives.sol

Status This issue has been fixed in the commit: 13f8fb4f0b3b014970ce6a2326b88ebe92b6e926.

3.10 Potential Blocking of Initial Reward Drop

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YAMIncentivizer`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [12]

Description

YAMv3 has developed unique tokenomics with incentivized pools. As mentioned in Section 3.9, the `UniswapV2` pool of `YAMv3/yUSD` trading pair will receive awards totaling approximately $925K$, with $92.5K$ distributed in the first week and decreasing by 10% each following week.

In this section, we examine the logic of incentivized pools. To elaborate, we show the related code snippet below. We notice that the very first initial reward drop has a built-in requirement, i.e., `require(yam.balanceOf(address(this))== 0, "already initialized")` (line 765).

```
748     function notifyRewardAmount(uint256 reward)
749         external
750         onlyRewardDistribution
751         updateReward(address(0))
752     {
753         if (block.timestamp > starttime) {
754           if (block.timestamp >= periodFinish) {
755               rewardRate = reward.div(DURATION);
756           } else {
757               uint256 remaining = periodFinish.sub(block.timestamp);
758               uint256 leftover = remaining.mul(rewardRate);
759               rewardRate = reward.add(leftover).div(DURATION);
760           }
761           lastUpdateTime = block.timestamp;
762           periodFinish = block.timestamp.add(DURATION);
763           emit RewardAdded(reward);
764         } else {
765           require(yam.balanceOf(address(this)) == 0, "already initialized");
766           yam.mint(address(this), initreward);
767           rewardRate = initreward.div(DURATION);
768           lastUpdateTime = starttime;
769           periodFinish = starttime.add(DURATION);
770           emit RewardAdded(reward);
771         }
772     }
```

Listing 3.19: YAMIncentivizer.sol

If an external `YAM` owner has a non-zero balance and simply transfers `1 WEI` of `YAM` to the `YAMIncentivizer` contract before the initial reward drop occurs, then the initial reward drop would fail. It would be successful if no one has the balance before the first call to `notifyRewardAmount()` is made.

Recommendation Deploy the related contracts in a coherent fashion and be the first one to call `notifyRewardAmount()` so that the initial reward will be properly loaded to the pool.

Status This issue has been fixed in the commit: 88611a5ceacea76a599a0977bf1e14d79d08ba59.

3.11 Oversized Rewards May Lock All Pool Stakes

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YAMIncentivizer`
- Category: Numeric Errors [17]
- CWE subcategory: CWE-190 [5]

Description

In this section, we continue to examine the logic of incentivized pools and focus on the `rewardPerToken()` routine. This routine 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 `YAMIncentivizer` to update and use the latest reward rate.

Our analysis leads to the discovery of a potential 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 683−686), especially when the `rewardRate` is largely controlled by an external entity, i.e., `rewardDistribution` (through the `notifyRewardAmount()` function).

```
663     modifier updateReward(address account) {
664         rewardPerTokenStored = rewardPerToken();
665         lastUpdateTime = lastTimeRewardApplicable();
666         if (account != address(0)) {
667             rewards[account] = earned(account);
668             userRewardPerTokenPaid[account] = rewardPerTokenStored;
669         }
670         _;
671     }
672 
673     function lastTimeRewardApplicable() public view returns (uint256) {
674         return Math.min(block.timestamp, periodFinish);
675     }
676 
677     function rewardPerToken() public view returns (uint256) {
678         if (totalSupply() == 0) {
```

```
                return rewardPerTokenStored;
            }
            return
                rewardPerTokenStored.add(
                    lastTimeRewardApplicable()
                        .sub(lastUpdateTime)
                        .mul(rewardRate)
                        .mul(1e18)
                        .div(totalSupply())
                );
        }
```

Listing 3.20: YAMIncentivizer.sol

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 important to transfer the ownership to the governance and ensure the given reward amount will not be oversized to overflow and lock users' funds.

Recommendation Deploy the related contracts in a coherent fashion and promptly transfer the ownership to the governance by ensuring the reward amount is proper, without resulting in overflowing and locking users' funds.

Status This issue has been fixed in the commit: 88611a5ceacea76a599a0977bf1e14d79d08ba59.

3.12 Incompatibility With Deflationary Tokens For Staking And reserveToken

- ID: PVE-012
- Severity: Informational
- Likelihood: N/A
- Impact: Medium
- Target: `YAMIncentivizer`, `YAMRebaser`
- Category: Business Logics [15]
- CWE subcategory: CWE-708 [9]

Description

In YAMv3, the `YAMIncentivizer` contract operates as the main entry for interaction with staking users. The staking users `deposit` `UniswapV2`'s LP tokens into the incentivized pool and in return get proportionate share of the pool's rewards. Later on, the staking users can `withdraw` their own assets from the pool. With assets in the pool, users can earn whatever incentive mechanisms proposed or adopted via governance.

Naturally, the above two functions, i.e., `deposit()` and `withdraw()`, are involved in transferring users' assets into (or out of) the pool. Using the `deposit()` function as an example, it needs to transfer deposited assets from the user account to the pool (line 210). When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts (lines 211 – 212).

```
    function stake(uint256 amount) public {
        _totalSupply = _totalSupply.add(amount);
        _balances[msg.sender] = _balances[msg.sender].add(amount);
        uni_lp.safeTransferFrom(msg.sender, address(this), amount);
    }

    function withdraw(uint256 amount) public {
        _totalSupply = _totalSupply.sub(amount);
        _balances[msg.sender] = _balances[msg.sender].sub(amount);
        uni_lp.safeTransfer(msg.sender, amount);
    }
```

Listing 3.21: YAMIncentivizer.sol

However, in the cases of deflationary tokens, as shown in the above code snippet, the input amount may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens. Apparently, these balance inconsistencies are damaging to accurate pool management and affects protocol-wide operation and maintenance. For the very same reason, we emphasize that the reserve token cannot be deflationary. (And keep in mind that `USDT` may become deflationary if the control switch in its token contract is turned on.)

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into `YAMv3`. With the nature of choosing possible pool assets for staking and incentives, it is possible to effectively regulate the set of assets allowed into the protocol.

Fortunately, the `UniswapV2`'s LP tokens are not deflationary tokens and there is no need to take any action in `YAMv3`. However, it is a potential risk if the current code base is used elsewhere or the need to add other arbitrary tokens arises (e.g., in listing new DEX pairs).

Recommendation Regulate the set of LP tokens supported in `YAMv3` and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been confirmed. As no deflationary tokens are being used for staking or reserves, the team decides no change necessary for the time being.

3.13 Full Charge of Proposal Execution Cost From Accompanying msg.value

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YAMGovernorAlpha`
- Category: Business Logics [15]
- CWE subcategory: CWE-770 [10]

Description

`YAM` adopts the governance implementation from `Compound` by adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. The original governance has been successfully audited by `OpenZeppelin`.

In the following, we would like to comment on a particular issue regarding the proposal execution cost. Note that the actual proposal execution is kicked off by invoking the governance's `execute()` function. This function is marked as `payable`, indicating the transaction sender is responsible for supplying required amount of `ETHs` as each inherent action (line 215) in the proposal may require accompanying certain `ETHs`, specified in `proposal.values[i]`, where i is the i^{th} action inside the proposal.

```
264     function execute(uint256 proposalId)
265         public
266         payable
267     {
268         require(state(proposalId) == ProposalState.Queued, "GovernorAlpha::execute:
            proposal can only be executed if it is queued");
269         Proposal storage proposal = proposals[proposalId];
270         proposal.executed = true;
271         for (uint256 i = 0; i < proposal.targets.length; i++) {
272             timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i],
                proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
                proposal.eta);
273         }
```

```
        emit ProposalExecuted(proposalId);
    }
```

Listing 3.22: YAMGovernorAlpha.sol

Though it is likely the case that a majority of these actions do not require any `ETHs`, i.e., `proposal.values[i] = 0`, we may be less concerned on the payment of required `ETHs` for the proposal execution. However, in the unlikely case of certain particular actions that do need `ETHs`, the issue of properly attributing the associated cost arises. With that, we need to better keep track of `ETHs` charged for each action and ensure that the transaction sender (who initiates the proposal execution) actually pays the cost. In other words, we do not rely on the governance's balance of `ETH` for the payment.

Recommendation Properly charge the proposal execution cost by ensuring the amount of accompanying ETH deposit is sufficient. If necessary, we can also return possible leftover in `msgValue` back to the sender.

```
    function execute(uint256 proposalId)
        public
        payable
    {
        require(state(proposalId) == ProposalState.Queued, "GovernorAlpha::execute:
            proposal can only be executed if it is queued");
        Proposal storage proposal = proposals[proposalId];
        proposal.executed = true;
        uint msgValue = msg.value;
        for (uint256 i = 0; i < proposal.targets.length; i++) {
            msgValue = sub256(msgValue, proposal.values[i])
            timelock.executeTransaction.value(proposal.values[i])(proposal.targets[i],
                proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
                proposal.eta);
        }
        emit ProposalExecuted(proposalId);
    }
```

Listing 3.23: YAMGovernorAlpha.sol (revised)

Status This issue has been confirmed. Considering the possibility that the `TimeLock` contract, if necessary, should be able to retrieve funds in a subsequent transaction, the team decides no change necessary for the time being.

3.14 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-014
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YAMGovernorAlpha`
- Category: Business Logics [15]
- CWE subcategory: CWE-837 [11]

Description

As discussed in Section 3.13, `YAM` adopts the governance implementation from `Compound` by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes()` and `proposalThreshold()`. Previously, we have examined the payment of proposal execution cost. In this section, we elaborate one corner case during a proposal submission, especially regarding the proposer qualification.

To be qualified as a proposer, the governance subsystem requires the proposer to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified by `proposalThreshold()`. In `YAMv3`, this number requires the votes of $50000 * 10 ** 24$ (about 1% of `YAM` token's total supply).

```
    function propose(
        address[] memory targets,
        uint[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        string memory description
    )
        public
        returns (uint256)
    {
        require(yam.getPriorVotes(msg.sender, sub256(block.number, 1)) >
            proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal
            threshold");
        require(targets.length == values.length && targets.length == signatures.length
            && targets.length == calldatas.length, "GovernorAlpha::propose: proposal
            function information arity mismatch");
        require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
        require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too
            many actions");

        uint256 latestProposalId = latestProposalIds[msg.sender];
        if (latestProposalId != 0) {
          ProposalState proposersLatestProposalState = state(latestProposalId);
          require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::
              propose: one live proposal per proposer, found an already active proposal"
              );
          require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha
              ::propose: one live proposal per proposer, found an already pending
```

```
                proposal");
180         }
181         ...
182     }
```

Listing 3.24: YAMGovernorAlpha.sol

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies up-front the qualification of the proposer (line 170): `require(yam.getPriorVotes(msg.sender, sub256(block.number, 1))> proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal threshold")`. Note that the number of prior votes is strictly higher than `proposalThreshold()`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 284) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold()`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```
277     function cancel(uint256 proposalId)
278         public
279     {
280         ProposalState state = state(proposalId);
281         require(state != ProposalState.Executed, "GovernorAlpha::cancel: cannot cancel
            executed proposal");

283         Proposal storage proposal = proposals[proposalId];
284         require(msg.sender == guardian || yam.getPriorVotes(proposal.proposer, sub256(
            block.number, 1)) < proposalThreshold(), "GovernorAlpha::cancel: proposer
            above threshold");

286         proposal.canceled = true;
287         for (uint256 i = 0; i < proposal.targets.length; i++) {
288             timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal
                .signatures[i], proposal.calldatas[i], proposal.eta);
289         }

291         emit ProposalCanceled(proposalId);
292     }
```

Listing 3.25: YAMGovernorAlpha.sol

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold()`.

```
    function propose(
        address[] memory targets,
        uint[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        string memory description
```

```
    )
        public
        returns (uint256)
    {
        require(yam.getPriorVotes(msg.sender, sub256(block.number, 1)) >=
            proposalThreshold(), "GovernorAlpha::propose: proposer votes below proposal
            threshold");
        require(targets.length == values.length && targets.length == signatures.length
            && targets.length == calldatas.length, "GovernorAlpha::propose: proposal
            function information arity mismatch");
        require(targets.length != 0, "GovernorAlpha::propose: must provide actions");
        require(targets.length <= proposalMaxOperations(), "GovernorAlpha::propose: too
            many actions");

        uint256 latestProposalId = latestProposalIds[msg.sender];
        if (latestProposalId != 0) {
          ProposalState proposersLatestProposalState = state(latestProposalId);
          require(proposersLatestProposalState != ProposalState.Active, "GovernorAlpha::
              propose: one live proposal per proposer, found an already active proposal"
              );
          require(proposersLatestProposalState != ProposalState.Pending, "GovernorAlpha
              ::propose: one live proposal per proposer, found an already pending
              proposal");
        }
        ...
    }
```

Listing 3.26: YAMGovernorAlpha.sol (revised)

Status This issue has been fixed in the commit: aa41103924d82e28f00c2a7b2fddd56e5da18f89.

3.15 Improved pendingAdmin Protection in TimeLock

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TimeLock`
- Category: Security Features [13]
- CWE subcategory: CWE-287 [6]

Description

In `YAMv3`, the governance contract, i.e., `YAMGovernorAlpha`, plays a critical role in governing and regulating the system-wide operations (e.g., rebasing configuration, reward adjustment, and ownership management). It also has the privilege to control or govern the life-cycle of proposals and enact on them regarding their submissions, executions, and revocations.

With great privilege comes great responsibility. Our analysis shows that the governance contract is indeed privileged and its execution agent, i.e., the `TimeLock` contract, needs to be properly authorized for various privileged tasks.

During our analysis, we identify a vulnerable time window during which `TimeLock` could be hijacked. Specifically, if we examine the `setPendingAdmin()` routine, it is a public function and any one can call it. More importantly, the first call to it (right after its deployment) is unprotected as `admin_initialized == false`. Because of that, `pendingAdmin` may be overwritten with whatever argument given in the first call. Due to the sensitiveness and importance of `TimeLock`, it is imperative to protect the first call to `setPendingAdmin()` as well. A natural requirement will be the first call needs to be initiated from the current `admin`.

```
    /**
    @notice queues a new pendingAdmin
    @param pendingAdmin_ the new pendingAdmin address
     */
    function setPendingAdmin(address pendingAdmin_)
        public
    {
        // allows one time setting of admin for deployment purposes
        if (admin_initialized) {
          require(msg.sender == address(this), "Timelock::setPendingAdmin: Call must
              come from Timelock.");
        } else {
          admin_initialized = true;
        }
        pendingAdmin = pendingAdmin_;

        emit NewPendingAdmin(pendingAdmin);
    }
```

Listing 3.27: TimeLock.sol

Recommendation Apply necessary authentication to ensure the first call to `setPendingAdmin()` only comes from the current `admin`.

```
    /**
    @notice queues a new pendingAdmin
    @param pendingAdmin_ the new pendingAdmin address
     */
    function setPendingAdmin(address pendingAdmin_)
        public
    {
        // allows one time setting of admin for deployment purposes
        if (admin_initialized) {
          require(msg.sender == address(this), "Timelock::setPendingAdmin: Call must
              come from Timelock.");
        } else {
          require(msg.sender == admin, "Timelock::setPendingAdmin: First call must come
              from admin.");
```

```
            admin_initialized = true;
        }
        pendingAdmin = pendingAdmin_;

        emit NewPendingAdmin(pendingAdmin);
    }
```

Listing 3.28: TimeLock.sol (revised)

Status This issue has been fixed in the commit: 0946a5bb4f869d4602cabda2cd1a28f46013e40a.

3.16 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.5.15;` instead of `pragma solidity ^0.5.15;`.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Moreover, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the YAMv3 design and implementation. We truly believe that YAM 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. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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

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 [20, 21, 22, 23, 25].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [26] 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] Yam Finance. The road to v3: Yamv2 interim governance summary. https://medium.com/@yamfinance/the-road-to-v3-yamv2-interim-governance-summary-f17ba4a9d1aa.

[3] Yam Finance. SAVE YAM! https://medium.com/@yamfinance/save-yam-245598d81cec.

[4] MITRE. CWE-1117: Callable with Insufficient Behavioral Summary. https://cwe.mitre.org/data/definitions/1117.html.

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

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

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

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

[9] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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