UMAMI DAO

SMART CONTRACT AUDIT

zokyo

September 7th, 2022 | v. 1.0

Security Audit Score

PASS

Zokyo's Security Team has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

TECHNICAL SUMMARY

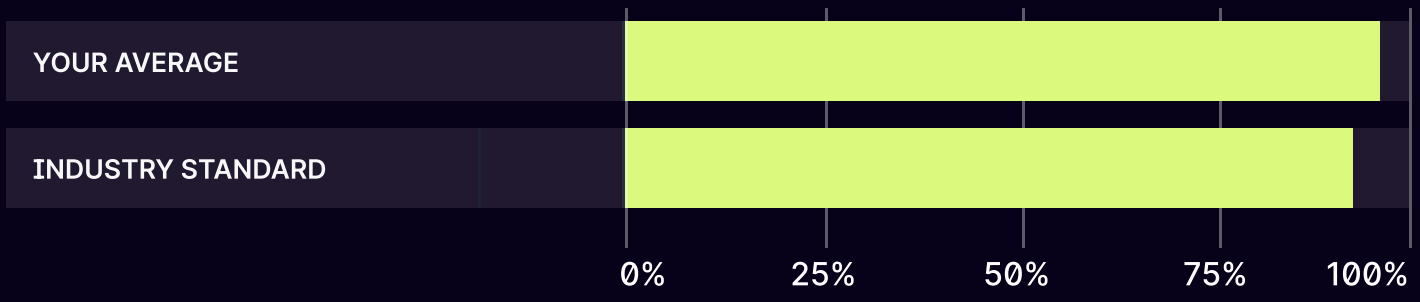
This document outlines the overall security of the Umami DAO smart contracts, evaluated by Zokyo's Blockchain Security team.

The scope of this audit was to analyze and document the Umami DAO smart contract codebase for quality, security, and correctness.

Contract Status

There was 1 critical issue found during the audit. (See Complete Analysis)

Testable Code



The testable code is 99.05%, which is above the industry standard of 95%. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the Ethereum network's fast-paced and rapidly changing environment, we at Zokyo recommend that the Umami DAO team put in place a bug bounty program to encourage further and active analysis of the smart contract.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The Smart contract's source code was taken from the Umami DAO repository.

Repository: https://github.com/Arbi-s/marinateV2

Last commit: 2593b617b3aa956ff0ab39a15aa66e43d76a4960

Contracts under the scope:

- MarinateReceiver.sol
- MarinateV2.sol
- ContractWhitelist.sol

Throughout the review process, Zokyo Security ensures that the contract:

- Implements and adheres to existing Token standards appropriately and effectively;
- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices inefficient use of gas, without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the latest vulnerabilities;
- Whether the code meets best practices in code readability, etc.

Zokyo's Security Team has followed best practices and industry-standard techniques to verify the implementation of Umami DAO smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are discovered. Part of this work includes writing a unit test suite using the Tru e testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	02	Cross-comparison with other, similar smart contracts by industry leaders.
03	Testing contract logic against common and uncommon attack vectors.	04	Thorough manual review of the codebase, line by line.

Executive Summary

Contracts are well written and structured. There was one critical issue found during the audit, alongside four with high severity, seven of medium severity and some issues with low severity and information issues . They are described in detail in the “Complete Analysis” section.

All of them were successfully resolved by the Umami DAO team. The mentioned findings may have an effect only in the case of specific conditions performed by the contract owner. Every of the mentioned findings may have an effect only in case of specific conditions performed by the contract owner and the investors interacting with it.

STRUCTURE AND ORGANIZATION OF DOCUMENT

For ease of navigation, sections are arranged from most critical to least critical. Issues are tagged “Resolved” or “Unresolved” depending on whether they have been fixed or addressed. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

CRITICAL | RESOLVED

MarinateV2.sol - withdrawMultiplier & stakeMultipler , marinator stakes NFT with tokenId = x while another marinator stakes tokenId = yof same NFT. The former marinator is able to withdraw the tokenId=y. NFT tokenIds are not holding same value, therefore that leads to loss of funds for some stakers.

Recommendation:
Keep track of token Ids of NFTs
fix#1 - Partner removed parts related to NFTs.

HIGH | RESOLVED

MarinateV2.sol - function setScale causes a flaw in the tokenomical model. SCALE value affects quantities like totalTokenRewardsPerStake. Changing SCALE using setScale does not update the quantities that were precalculated, hence this leads to a miscalculation when you execute that line after changing SCALE to a new value
uint256 owedPerUnitStake = totalTokenRewardsPerStake[token] -
paidTokenRewardsPerStake[token][user];
uint256 totalRewards = (info.multipliedAmount * owedPerUnitStake) / SCALE;
This is limited to be called by admin though so as long as admin does not call it, issue is not having an effect on the project.

Recommendation:
refer to the tokenomics of the project and recall what does SCALE value affect to update these values when you're setting SCALE to a new value in the body of this method.
fix#1 - SCALEnow is immutable, setScaleis removed.

HIGH | RESOLVED

MarinateV2.sol - the implementation of hooks _beforeTokenTransfer & _afterTokenTransfer leads to issues in specific scenarios in which a marinator (i.e. contract holding mUMAMIs) is being dropped out of the whitelist. The issue is described in the following scenario, shown also in Proof of Concept (PoC) tests.

Contract A & B are both whitelisted, so they stake their UMAMI Contract A is no longer whitelisted
Contract A sends the mUMAMI to B and in the process B collects rewards while A does not
Contract B unstakes all mUMAMI and collects rewards
MarinateV2 is possessing a quantity of uncollectible rewards
The rewards are never claimed for Contract A even if it comes back to the white list

Also in this context the totalMultipliedStaked does not hold the expected value proportional to the totalSupply of MarinateV2 since the value it holds at the end of this scenario will actually lead to unallocated funds when new rewards are added since totalTokenRewardsPerStakeis not fairly distributed.

Recommendation:
declare a storage map for unallocatedRewards which going to receive rewards in this case and allocate them to totalTokenRewardsPerStake.

fix#1 - Transfers now are only allowed when both parties are whitelisted. Also, rewards are sent to the marinator before it is removed from whitelist.

HIGH | RESOLVED

MarinateV2.sol - migrateToken transfers reward tokens to another address and does not update the value of totalTokenRewardsPerStake accordingly, which shall in turn cause loss of funds to some stakers trying to collect rewards.

fix#1 - Partner removed migrateToken

HIGH | RESOLVED

ContractWhitelist.sol - unfortunately, the implementation of isContract can be surpassed by an attacker as contract does not show up the containment of code during constructor call. In the PoC tests, ImpersonatorFactory does just that: a factory of contracts (Imperso

Recommendation:
f the attacker could not be discovered during stake, s/he can be discovered in other entrypoints like addToContractWhitelist or withdraw. In that sense, if a contract is discovered to have staked tokens without being added to the whitelist (and not removed priorly by admin) then the suitable measure might be taken to deincentivize that.

fix#1 - Resolve attempt done by partner but in a way that introduced a limitation on operation. **Investors cannot use Multisig.** The Impersonator issue though is solved and this attack is now mitigated.

MEDIUM | RESOLVED

ContractWhitelist.sol - isEligibleSender which is a modifier applied to limit access on stake is now blocking multisig wallets due to tx.origin == msg.sender.

fix#2 - Partner made it clear that by design their intention is to be aware and whitelist contracts including multisigs.

MEDIUM | RESOLVED

MarinateV2.sol - Regarding removeApprovedRewardToken, require(IERC20(token).balanceOf(address(this)) == 0, "Reward token not completely claimed by everyone yet"). **New issue is introduced:** having balance required to be strictly equal to zero make it exposed to a scenario in which reward tokens with small uncollectible residues are not possible to be removed. A scenario is reproduced in a Proof of Concept test.

Recommendation:
will be to adjust numbers, in terms of tokenomics of the project, in a way to deal with this or make sure the last staker collecting reward is actually collecting everything by rounding up the division in certain occasions rather than rounding down each time.

fix#2 - removeApprovedRewardToken(), the method has been removed

MEDIUM | RESOLVED

MarinateV2.sol - removeApprovedRewardToken & removeApprovedMultiplierToken, MarinateV2 still holds balances of NFTs and reward tokens after removal and can not be dealt with using normal circumstances.

Recommendation:
better off having a require statement that checks MarinateV2 does not have balance of those assets before removing them from their respective lists.

fix#1 - NFT logic in the contract is already removed, hence removeApprovedMultiplierToken issue is no longer relevant. Regarding removeApprovedRewardToken, partner added a check on the reward token balance of the contract to ensure there are no residuals left: require(IERC20(token).balanceOf(address(this)) == 0, "Reward token not completely claimed by everyone yet"). Hence token will not be removed from list of reward tokens while there exist a balance of it.

MEDIUM | RESOLVED

MarinateV2.sol - addApprovedMultiplierToken, setScale & setDepositLevel: new values are not validated before setting. For instance, there's no upper nor lower limit for mulptiplierthat is set in addApprovedMultiplierToken.

fix#1 - NFT logic and setScale are removed, hence first two issues are no longer relevant. body of setDepositLimit is unchanged hence issue persists partially only for that case.

fix#2 - partner added a check on deposit limit
require(limit < IERC20(UMAMI).totalSupply(), "Deposit limit cannot be greater than totalSupply");

MEDIUM | RESOLVED

MarinateReceiver.sol - onlyAdminguard applied on private function _addRewards. This causes an issue for sendBalancesAsRewardsthat is supposed to be callable by admin and bots (automation) as it will only make it callable by admin and block automation calls.

Recommendation:
no point of applying that guard in this context since the function is private already.

fix#1 - onlyAdminguard is removed.

MEDIUM | RESOLVED

MarinateV2.sol - migrateToken(), Too much for admin, might be lured into it if s/he is a single wallet having this kind of control. Also, the community might be concerned over this act of withdrawing funds.

Recommendation:
add multisig

fix#1 - function is removed

LOW | RESOLVED

MarinateV2.sol & MarinateReciever.sol - unchecked return values:

- MarinateV2.constructor : rewardTokens.add(_UMAMI)
- MarinateV2.addApprovedRewardToken: rewardTokens.add(token)
- MarinateV2.removeApprovedRewardToken: rewardTokens.remove(token)
- MarinateReceiver.addDistributedToken: distributedTokens.add(token)
- MarinateReceiver.removeDistributedToken: distributedTokens.remove(token)

Recommendation:
wrap in require statements

LOW | RESOLVED

MarinateV2.sol & MarinateReciever.sol - removeApprovedRewardToken, removeApprovedMultiplierToken & removeDistributedToken - for loop might incur too much computation that goes above limit if list is too long. It can be avoided by utilizing a better data structure for this purpose.

Recommendation:
Checkout enumerables by openzeppelin
https://docs.openzeppelin.com/contracts/3.x/api/utils#EnumerableSet

fix#1 - NFT logic is omitted, hence removeApprovedMultiplierTokenissue is no longer relevant. Regarding the other two functions, partner utilized the enumerables in the contracts.

INFORMATIONAL | RESOLVED

ALL contracts, Lock the pragma to a specific version, since not all the EVM compiler versions support all the features, especially the latest one's which are kind of beta versions, So the intended behavior written in code might not be executed as expected. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler, which may have higher risks of undiscovered bugs.

Recommendation:
fix version to 0.8.4

fix#1 - solidity version is locked to 0.8.4 for the 3 contracts in scope

INFORMATIONAL | RESOLVED

MarinateV2.sol - follow checks-effects-interactions pattern in implementation of _stakeMultiplier.

Recommendation:
Watchout for the transfer nft interaction.

fix#1 - since NFT logic is omitted, this issue is no longer relevant.

	MarinateV2.sol	MarinateReceiver.sol	ContractWhitelist.sol
Re-entrancy	Pass	Pass	Pass
Access Management Hierarchy	Pass	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass	Pass
Unexpected Ether	Pass	Pass	Pass
Delegatecall	Pass	Pass	Pass
Default Public Visibility	Pass	Pass	Pass
Hidden Malicious Code	Pass	Pass	Pass
Entropy Illusion (Lack of Randomness)	Pass	Pass	Pass
External Contract Referencing	Pass	Pass	Pass
Short Address/Parameter Attack	Pass	Pass	Pass
Unchecked CALL Return Values	Pass	Pass	Pass
Race Conditions/Front Running	Pass	Pass	Pass
General Denial Of Service (DOS)	Pass	Pass	Pass
Uninitialized Storage Pointers	Pass	Pass	Pass
Floating Points and Precision	Pass	Pass	Pass
Tx.Origin Authentication	Pass	Pass	Pass
Signatures Replay	Pass	Pass	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass	Pass	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security team

As part of our work assisting Umami DAO in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Truffle testing framework.

Tests were based on the functionality of the code, as well as a review of the Umami DAO contract requirements for details about issuance amounts and how the system handles these.

MarinateReceiver
- addDistributedToken()
  - ✓ adds token to distributed tokens (636ms)
- removeDistributedToken()
  - ✓ removes token from distributed tokens (282ms)
- setMarinateAddress
  - ✓ setMarinateAddress (172ms)
- recoverToken
  - ✓ recoverToken (170ms)
- recoverEth
  - ✓ recoverEth (196ms)
- sendBalancesAsRewards
  - ✓ sendBalancesAsRewards (1403ms)

MarinateV2
- Initialization
  - ✓ Tests initialization variables (274ms)
- stake
  - ✓ stakes umami tokens (485ms)
- addReward
  - ✓ addReward (480ms)
- addApprovedRewardToken
  - ✓ addApprovedRewardToken (166ms)
- removeApprovedRewardToken
  - ✓ removeApprovedRewardToken (197ms)
- addApprovedMultiplierToken
  - ✓ addApprovedMultiplierToken (345ms)
- removeApprovedMultiplierToken
  - ✓ removeApprovedMultiplierToken (221ms)

```
setScale, setStakeEnabled, setMultiplierStakeEnabled, setStakingWithdrawEnabled
    ✓ toggles setScale, setStakeEnabled, setMultiplierStakeEnabled,
      setStakingWithdrawEnabled, depositLimit (272ms)
stakeMultiplier
    ✓ stakeMultiplier (744ms)
withdrawMultiplier
    ✓ withdrawMultiplier (309ms)
withdraw
    ✓ withdraw (517ms)
getAvailableTokenRewards
    ✓ getAvailableTokenRewards (280ms)
addToContractWhitelist
    ✓ addToContractWhitelist (514ms)
isContract()
    ✓ returns true/false if address is contract address or not  (114ms)
 recoverEth()
    ✓ sends all ether balance within contract to admin caller (96ms)
 migrateToken()
    ✓ transfers token from contract to receipient (155ms)
transfer()
    ✓ transfer tokens and calls after token transfer hook (312ms)
getAvailableTokenRewards
    ✓ getAvailableTokenRewards (499ms)
claimRewards
    ✓ claimRewards (205ms)

  25 passing (39s)
```

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
MarinateReceiver.sol	100	70	100	100
MarinateV2.sol	98.82	83.7	100	98.48
ContractWhitelist.sol	100	80	100	100
All files	**99.05**	**89.91**	**100**	**99.08**

We are grateful to have been given the opportunity to work with the Umami DAO team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo's Security Team recommends that the Umami DAO team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo