UMAMI DAO

SMART CONTRACT AUDIT

zokyo

November 23rd, 2022 | v. 1.0

Security Audit Score

PASS

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

TECHNICAL SUMMARY

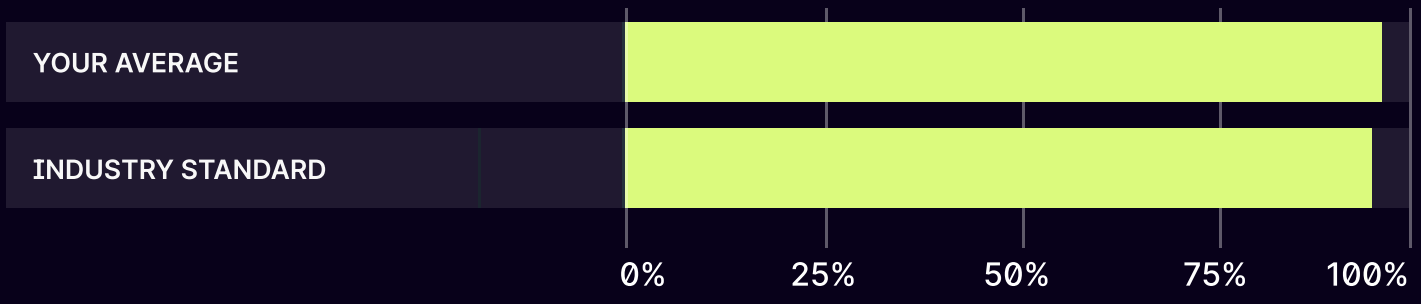
This document outlines the overall security of the Umami DAO smart contracts evaluated by the Zokyo Security team.

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

Contract Status

There were 3 critical issues found during the audit. (See Complete Analysis)

Testable Code



97.9% of the code is testable, which is above the industry standard of 95%.

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

Table of Contents

Auditing Strategy and Techniques Applied 3

Executive Summary 4

Structure and Organization of Document 5

Complete Analysis 6

Code Coverage and Test Results for all files written by Zokyo Secured team 23

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Umami DAO repository: https://github.com/UmamiDAO/auto-compounders

Last commit: fe23624efcd7b94e757be0d2f2d1f5e7b5af5e42

Within the scope of this audit, Zokyo auditors have reviewed the following contract(s):

- MarinateStrategyFarm.sol
- MarinateAutoCompounders.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Umami DAO smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. A part of this work included writing a unit test suite using the Hardhat testing framework. In summary, our strategies consisted mostly 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

The Zokyo team has conducted a security audit of the given codebase. The contracts provided for an audit are well written and structured. All the findings within the auditing process are presented in the “Complete Analysis” section.

There were three critical issue found during the audit, alongside three with high severity, some of medium severity and a couple of informational issues . All 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 the ease of navigation, document's sections are arranged from the most critical to the least critical. Issues are tagged as "Resolved" or "Unresolved" depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the client team and the client team are aware of it, but they have chosen to not solved it. The issues that are tagged as "Verified" contain unclear or suspicious functionality that either needs further explanation from the Customer or remains disregarded by the Customer. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or 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

SYSTEM OVERVIEW

The main use case of the contracts audited by the Zokyo team is to be an investment platform for Umami/mUmami ecosystem. During the manual and testing stages of the contracts audit, multiple security issues were found. All those can be found in the following sections. Beside these findings, there are also remarks that have to be made about the overall security of the contracts submitted for audit. Decision makers decided that one of the two contracts being audited will not be deployed, hence all the issues related to that contract became irrelevant and considered resolved.

The contracts receive deposit from investors to be staked and action is taken by reinvesting the deposits. The deposits can be invested on any tokens chosen by decision makers, and the investment takes place within the marinate ecosystem. Rewards harvested from the investment are distributed to the investors. During the initial assessment of the protocol, it has been discovered that generic tokens being a deposit can cause critical faulty calculations that lead to lost funds. But it is revealed by our partner that deposit tokens aren't meant to be arbitrary, for instance, the intention is assigned depositToken to be Umami and in this case there are no miscalculations in the way described in following sections. Other highly severity issues include absence of protection against slippage, which takes place during reinvestment of deposits. The partner acknowledged the issue and stated that the related attacks are mitigated by the choice to deploy the ecosystem on arbitrum. Despite that, the concern remains in case unintentional unfavourable swaps still can take place, partner took the responsibility to handle the reinvestment process wisely without being exposed to high frequency price changes as the contract does not protect against those. Hence, Zokyo advises the team to acknowledge the design decisions and take extra care while operating the contracts in their current design.

FINDINGS SUMMARY

#	Title	Risk
1	Staking without owner's agreement	Critical
2	Possible miscalculation mixing tokens	Critical
3	Decimals mismatch	Critical
4	SCALE parameter messes up defi calculations	High
5	No slippage set	High
6	Method messes up reward collection	High
7	Lack of validation of address of beneficiary	Medium
8	Token residuals remain (non-withdrawable)	Medium
9	Token withdrawal is centralized	Medium
10	Centralization Risk	Medium
11	Unsafe casting	Low
12	Method should be a view	Low
13	Inefficient loop	Low
14	Non-validated entry data on deployment	Low
15	Argument in method might cause inconsistent router path	Low
16	Method argument is not validated	Informational
17	Lock Solidity version	Informational
18	Storage used incurring unnecessary gas cost	Informational
19	Value recommended be constant	Informational
20	_depositToken arguments lack validation	Informational

#	Title	Risk
21	_deposit arguments lack validation	Informational
22	Method recommended not be public (limit access as needed)	Informational
23	Modifier will block multisig	Informational
24	Misinformation within incode documentation	Informational
25	Not using SafeERC20	Informational

CRITICAL | RESOLVED

Staking without owner's agreement

MarinateStrategyFarm.sol - in body of stakeFor Can be misused by ANY caller (msg.sender) as s/he can stake STOKEN for any user without other user's consent (ps. after approval). Despite that it is not considered a loss as users are receiving something that they can redeem, but this would lead to many upsets by users. Recommendation - Typical way of implementing functions similar to stakeFor is to make the caller pay the deposit from his/her own tokens while the beneficiary referred to by for is harvesting the stakes.

Fix-1:

According to partner's dev: `the Arbi's emission strategy will no longer be used and should be removed from scope, that includes the following file: MarinateStrategyFarm.sol.`. Hence, issues in MarinateStrategyFarm.sol are no longer relevant.

CRITICAL | RESOLVED

Possible miscalculation mixing tokens

MarinateAutoCompounder - in body of _reinvest: totalDeposit refers to depositToken which is different from toRedeposit that refers to UMAMI. This inconsistency leads to the presence of non-withdrawable quantities, as shown in PoC test.

Fix-1:

I might not fully fathom the tokenomics of your project as much as you, hence my description of the finding might not be very explanatory. Despite that though, this finding is based on a proof of concept (poc) test in which the investor ends up having AutoCompounder ERC20 that are not withdrawable because the underlying asset is zeroed.

```
let autoCompounder = ctx.ct.mockAutoCompounder;
// adminFee = reinvestFee = 0
await autoCompounder.setVariable("ADMIN_FEE_BIPS", 0);
await autoCompounder.setVariable("REINVEST_REWARD_BIPS", 0);
let rewardAmount = ether(10);
let umamiAmountSwapped = bitcoin(20); // 2e9
```

```
ctx.WETH.balanceOf.returns(0); // Smock balance of WETH
await depositToken.approve(autoCompounder.address, bitcoin(80));
// 8e9
await autoCompounder.deposit(bitcoin(80));
await ctx.ct.rewardToken.transfer(autoCompounder.address,
rewardAmount);
// smock UMAMI balance of AutoCompounder to emulate effect of
uniswap
ctx.ct.routerMock.exactInput.returns(umamiAmountSwapped);
await autoCompounder.reinvest();
// we have: totalDeposits = totalDeposits + umamiAmountSwapped =
1e11
expect(await
autoCompounder.totalDeposits()).to.be.eq(bitcoin(100));
expect(await
depositToken.balanceOf(autoCompounder.address)).to.be.eq(bitcoin(80));
expect(await autoCompounder.totalSupply()).to.be.eq(bitcoin(80));
// Withdraw all ERC20 of autoCompounder
// HENCE ISSUE totalDeposits > actual depositToken balance of
AutoCompounder
// Despite that totalSupply is 8e9 , we can only withdraw 6.4e9
await autoCompounder.withdraw(bitcoin(64));
// balance of depositToken is ZERO now
expect(await
depositToken.balanceOf(autoCompounder.address)).to.be.eq(bitcoin(0));
// Still there's totalySupply left that should be withdrawable
expect(await autoCompounder.totalSupply()).to.be.eq(bitcoin(16));
```

The test here does the following:

- Stake depositToken (deposit).
- Add rewards to autoCompounder.
- Reinvest: swapping rewards just added for depositToken and staking them in marinate.
- Withdraw: issue happens here as user won't be able to replace all his tokens for the
- underlying asset.

Fix-2:
According to dev team explanation, having depositToken asserted to refer to mUmami matters a lot, hence this constraint shall be assured to be configured. This could have been asserted in the way the contract is implemented, but if wiring the contracts together goes the way it should, there shall be no issue anyway. Things become more related to project tokenomics which is not the intention of a security audit.

```
totalDeposits = totalDeposits + toRedeposit;
```

both quantities `totalDeposits & toRedeposit` refer to `mUMAMI`. Given also that staked, `UMAMI` gives the same amount staked in `mUMAMI` to staker. Therefore, as long as dev team is aware and assuring the way the contracts are configured, the issue should be irrelevant.

CRITICAL | RESOLVED

Decimals mismatch
MarinateAutoCompounder.sol - in body of `_deposit` the amount minted is not considering the decimals of the `depositToken`. For example assume `depositToken` initiated by constructor is dai with 18 decimals, consequently the `amount` minted `getSharesForDepositTokens(amount)` will output `amount` (i.e. since totalSupply =0), considering that decimals of `MarinateAutoCompounder` is 9. This shall lead to lost funds in subsequent calls that are building on this.

Recommendation
This issue can be solved in several ways, one of them is to restrict assig nment of `depositToken` to tokens with decimals = 9. Or, preferably, adjust `amount` according to the ratios of decimals between `MarinateAutoCompounder` and `depositToken`.

fix-1:
Issue is conditionally resolved since dev team stated that they will avoid utilizing any decimals other than 9. If decimals for token used = 9, then the issue shall be irrelevant.

HIGH | RESOLVED

SCALE parameter messes up defi calculations

MarinateStrategyFarm.sol - Admin changing the value of `SCALE` after contract is already deployed and receiving rewards will mess up things recommendation—while changing value of `SCALE`, you need to rescale values of `totalTokenRewardsPerStake` & `paidTokenRewardsPerStake`.

fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

HIGH | ACKNOWLEDGED

No slippage set

MarinateAutoCompounder.sol - in body of `convertRewardTokensToDepositTokens`, putting `amountOutMinimum:0` in swaps will make this call susceptible to attacks that might lead to unnecessary lost funds for users.

fix-1:

Even though Arbitrum possesses a fair timing sequencer that protects from front running. It is still risky to leave this line of code as no check on price while swapping leads to a chance to lose assets to an accidental sudden price change without necessarily having an attacker aiming to target that.
After discussing the issue with the team, it is established that changing the current state of the contract is not possible. The team though reassured that work on a newer version in which this issue is avoided by receiving oracle price update is ongoing. Finally, the risk is acknowledged by both parties and work to encounter it is expected in a newer version.

HIGH | RESOLVED

Method messes up reward collection

MarinateStrategyFarm.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 informed us they ditched the development of MarinateStrategyFarm at the moment.

MEDIUM | RESOLVED

Lack of validation of address of beneficiary

MarinateStrategyFarm.sol - in body of `stakeFor - user` (the beneficiary of the stake) is not validated to be a non-zero address.

fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

MEDIUM | RESOLVED

Token residuals remain (non-withdrawable)

MarinateStrategyFarm.sol - `removeApprovedRewardToken` MarinateStrategyFarm still holds balances of reward tokens after removal and can not be dealt with using normal circumstances.

Recommendation

better off having a require statement that validate MarinateStrategyFarm does not have balance of those assets before removing them from their respective lists.

fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

MEDIUM | RESOLVED

Token withdrawal is centralized

MarinateStrategyFarm.sol & MarinateAutoCompounder - `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:

Partner stated that `The Umamo DAO Multisig is used as the only admin for our deployment.` hence there's no centralization risk issue.

MEDIUM | RESOLVED

Centralization Risk

MarinateStrategyFarm.sol & MarinateAutoCompounder.sol - Admin enjoys too much authority. The general theme of the repo is that admin has power to call several functions like adding/removing reward tokens , migrating/recovering tokens/eth, general setters/ mutators. Some functions can be more highly severe to be left out controlled by one wallet more than other functions; depending on the intentions behind the project.

Recommendation

Apply governance / use multisig wallets

fix-1:

Partner stated that `The Umamo DAO Multisig is used as the only admin for our deployment.` hence there's no centralization risk issue.

LOW | RESOLVED

Unsafe casting

MarinateStrategyFarm.sol - in body of `stakeFor` casting `block.timestamp + lockDuration` to uint32 seems unsafe. This will take effect in about 84 years.

fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

LOW | RESOLVED

Method should be a view

`checkReward` better be a view function, that's based on info from priorly audited `MarinateV2` repo which contained `function getAvailableTokenRewards(address staker, address token) external view returns (uint256 totalRewards)` which is a view function, hence all external & internal calls in the body of `checkReward` are just views no state changing gas-costing transaction is needed.

LOW | ACKNOWLEDGED

Inefficient loop

MarinateStrategyFarm.sol & MarinateAutoCompounder.sol - `removeApprovedRewardToken` & `removeRewardToken` - 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

despite it is recommended to avoid loops as much as possible in implementation given that there is a way to use the mapping O(1) to avoid loop. We acknowledge that the issue is very unlikely to cause a blocker for this method.

LOW | RESOLVED

Non-validated entry data on deployment

MarinateStrategyFarm.sol - in constructor - not verifying addresses `STOKEN` & `feeDestination` to be non-zero and `lockDuration` is not validated to be within valid range

Fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

LOW | RESOLVED

Argument in method might cause inconsistent router path

MarinateAutoCompounder.sol - in body of `addRewardToken`, the argument `swapRouter` can pose a possibility of inconsistency arises as admin possibly mistakenly add a route that is inconsistent with rewardToken and Umami path. Knowing that UMAMI should be the `tokenOut` of the swap process, then constructing the path in function body seems to be the recommended way to do it. This is deduced from the implementation of `convertRewardTokensToDepositTokens`.

Recommendation

construct the route in function body rather than taking it as an argument from admin.

INFORMATIONAL | RESOLVED

Method argument is not validated

MarinateStrategyFarm.sol - `addApprovedRewardToken` argument is not validated before setting.

Recommendation

verify address is non-zero address.

Fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

INFORMATIONAL | RESOLVED

Lock Solidity version

Both 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:

According to partner's dev: `the Arbi's emission strategy will no longer be used and should be removed from scope, that includes the following file: MarinateStrategyFarm.sol.` MarinateAutoCompounder is locked to 0.8.4, hence it is resolved.

INFORMATIONAL | RESOLVED

Storage used incurring unnecessary gas cost

MarinateStrategyFarm.sol - Extra unnecessary storage `stakedBalance` carries redundant information in storage. Since `farmerInfo[user].amount` holds the same information.

Fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

INFORMATIONAL | RESOLVED

Value recommended be constant

MarinateStrategyFarm.sol - `BIPS_DIVISOR` acts as a constant as it's not changeable after smart contract is deployed hence it is recommended to be declared as a predefined constant.

Fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

INFORMATIONAL | RESOLVED

_depositToken arguments lack validation

MarinateAutoCompounder.sol - constructor - Addresses `_depositToken`, `_marinateContract` & `_router` are not validated to be non-zero.

INFORMATIONAL | RESOLVED

_deposit arguments lack validation

MarinateAutoCompounder.sol - in body of `_deposit, amount` not validated to be non-zero which leads to unnecessary computation going on if zero input is present. Also, in the body of `setFeeDestination,` new `feeDestination` is not validated as non-zero address.

INFORMATIONAL | RESOLVED

Method recommended not be public (limit access as needed)

MarinateAutoCompounder.sol - function `setAllowances` can be `external` rather than `public,` it is recommended as one of the best practices to limit function access according to the usage.

INFORMATIONAL | RESOLVED

Modifier will block multisig

MarinateAutoCompounder.sol - in body of `reinvest,` modifier `onlyEOA` meant to limit calls to non-contract addresses by applying `msg.sender == tx.origin`, but this will end up by having no place for multisig calls.

INFORMATIONAL | RESOLVED

Misinformation within incode documentation

MarinateStrategyFarm.sol - misguiding documentation of the function `setAllowEarlyUnlock(bool _earlyUnlock) @param _earlyUnlock the duration in seconds`

`_earlyUnlock` is a bool not a number

Fix-1:

Partner informed us they ditched the development of MarinateStrategyFarm at the moment.

INFORMATIONAL | ACKNOWLEDGED

Not using SafeERC20

MarinateStrategyFarm.sol & MarinateAutoCompounder.sol - throughout the contracts, token transfer is frequently executed in a require statement applied on the return of the transfer function called on the external token, one example:

```
require(IERC20(STOKEN).transfer(feeDestination, withdrawFee),
"withdraw fee transfer failed");
```

It is more preferred to use `safeTransfer` to using this pattern.

Fix:

According to a discussion, we acknowledge the point of this implementation, hence it shall not become a blocker for the readiness of the contracts.

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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a 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 Hardhat testing framework.

The 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.

MarinateStrategyFarm

addReward()
- ✓ It should revert if not approved reward token (63ms)
- ✓ It should set totalTokenRewardsPerStake correctly for 6 decimals (4855ms)
- ✓ It should set totalTokenRewardsPerStake correctly for 18 decimals (1097ms)
- ✓ small rewards (889ms)

stake()
- ✓ should revert if amount is invalid
- ✓ it should set total staked to the correct value (48ms)
- ✓ Should create the Farmer struct with the correct values in farmerInfo (53ms)
- ✓ Should stake, add rewards then stake again for another user (1013ms)

withdraw( )
- ✓ Should revert if the timelock is not complete (66ms)
- ✓ should revert if the user has not deposited
- ✓ should collect and pay pending rewards to the user (1385ms)
- ✓ Should return the share tokens (96ms)
- ✓ Should set global variables to appropriate values (91ms)
- ✓ Should take the fee in the correct period (107ms)

claimRewards()
- ✓ reverts if the user has no stake
- ✓ pays all rewards to the user

recoverETH
- ✓ recoverETH (515ms)

migrateToken
- ✓ migrateToken (524ms)

setLockDuration()
- ✓ setLockDuration
- ✓ setFeePeriod
- ✓ setFeeDestination()
- ✓ setWithdrawalFee()

✓ setScale()
✓ setAllowEarlyUnlock()
removeApprovedRewardToken()
✓ Should remove addApprovedRewardToken
claimRewards()
✓ claimRewards
Autocompounder
deposit
✓ deposit (188ms)
recoverETH
✓ recoverETH (428ms)
initialization
✓ intialization (914ms)
migrateToken
✓ migrateToken (67ms)
updateMinTokensToReinvest
✓ updateMinTokensToReinvest
updateMinTokensToReinvest
✓ updateMinTokensToReinvest (1461ms)
updateAdminFee
✓ updateAdminFee
updateReinvestReward
✓ updateReinvestReward
updateAdminFee
✓ updateAdminFee
rewardTokensLength
BigNumber { value: "1" }
✓ ewardTokensLength
decimals
✓ decimals
setFeeDestination
✓ setFeeDestination
rewardTokensLength
✓ Should withdraw the correct amount of mUMAMI after rewards have been compounded over time (24095ms)

39 passing (83s)

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
MarinateAutoCompo under.sol	98,11	68,33	100	98.15	
MarinateStrategyF arm.sol	97.7	61.9	100	97.7	
All files	**97.9**	**65.1**	**100**	**97.9**	

We are grateful for 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 Security recommends the Umami DAO team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo