SMART CONTRACT AUDIT

ZOKYO.

July 13th 2022 | v. 1.0

PASS

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

TECHNICAL SUMMARY

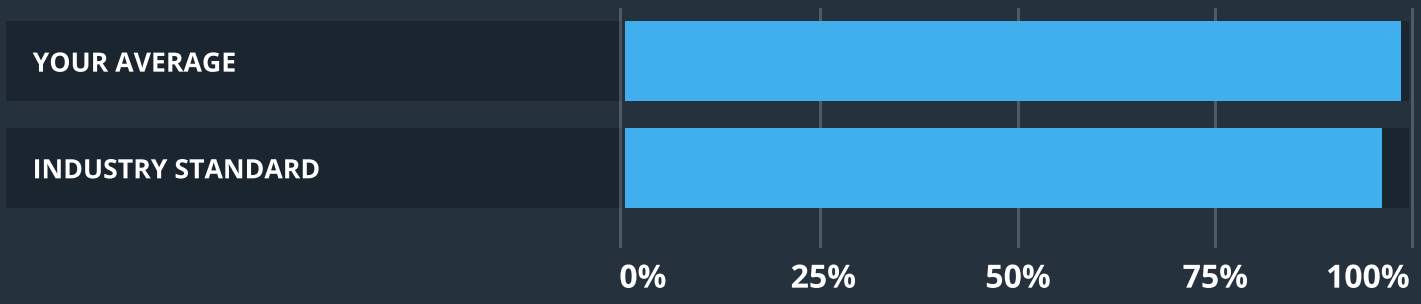
This document outlines the overall security of the Umami DAO smart contracts, evaluated by Zokyo's Blockchain Security team.

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

Contract Status

There were critical and high issues found during the audit.

Testable Code



The testable code is 98.92%, 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, 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.
https://github.com/Arbi-s/vault-strategies

Last audited commit: 0bd9c2a03e3c05cd25927d30ba71edc676184b1f

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

- BaseVault
- VaultStorage
- TracerGMXVault
- VaultLifeCycle
- Vault
- TcrPricing
- ShareMath
- GlpPricing
- L2Encoder

Throughout the review process, care was taken to ensure that the contract:

- Implements and adheres to existing standards appropriately and effectively;
- Documentation and code comments match logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices in efficient use of resources, 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 Truffle testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

1	Due diligence in assessing the overall code quality of the codebase.	3	Testing contract logic against common and uncommon attack vectors.
2	Cross-comparison with other, similar smart contracts by industry leaders.	4	Thorough, manual review of the codebase, line-by-line.

EXECUTIVE SUMMARY

There were a one critical issue, a one issue with the high severity and some with medium severity found during the audit. All the mentioned findings may have an effect only in case of specific conditions. They are described in detail in the “Complete Analysis” section.

The contracts are in a good condition. They are well written and structured. All the issues we found were successfully resolved by Umami DAO team. After a review of the fixes and comments from the Umami DAO team, we decided to mark a one issue as invalid.

Based on the status of issues, we can give a 96 security score.

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. Issues tagged “Verified” contain unclear or suspicious functionality that either needs explanation from the Customer’s side or it is an issue that the Customer disregards as an issue. 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

TracerGMXVault.sol – In body of swapToStable(...), call stack starts from external function commitAndClose(). This transaction can be spotted in pool and exposed to sandwich attack because of this snippet:

```
ISwapRouter.ExactInputParams memory params =  ISwapRouter.ExactInputParams({
    path: route,
    recipient: address(this),
    deadline: block.timestamp,
    amountIn: wethBalance,
    amountOutMinimum: 0
});
return router.exactInput(params)
```

setting amountOutMinimum to zero give a chance to the attacker to exploit that.
Severity of this explained by uniswap's official docs
https://docs.uniswap.org/protocol/guides/swaps/single-swaps
amountOutMinimum: we are setting to zero, but this is a significant risk in production. For a real deployment, this value should be calculated using our SDK or an onchain price oracle - this helps protect against getting an unusually bad price for a trade due to a front running sandwich or another type of price manipulation

Recommendation:
When trading from a smart contract, the most important thing to keep in mind is that access to an external price source is required. Without this, trades can be frontrun for considerable loss.
– uniswap's official docs

HIGH | RESOLVED

ShareMath.sol – In body of pricePerShare(...), initial case on zero totalSupply `pps` is initiated to 10**decimals where decimals refer to vault ERC20 decimals not the asset decimal.

```
    uint256 singleShare = 10**decimals;
    return totalSupply > 0 ? singleShare.mul(totalBalance).div(totalSupply) : singleShare;
```

Example how this function is used

```
    roundPricePerShare[_vaultRound] = ShareMath.pricePerShare(
      totalSupply(),
      IERC20(_vaultParams.asset).balanceOf(address(this)),
      _vaultParams.decimals
    );
```

_vaultParams.decimals refer to vault's decimals which messes up the calculation.
This works inconsistently since in a case where asset is , for instance, usdc (decimals = 6) and vault decimals = 18 this will end up having pricePerShare in 18 decimals.

MEDIUM | RESOLVED

TracerGMXVault.sol - There are multiple calls to the approve function form the IERC20, because the ERC20 standard does not require to fail inside the approve function and there are multiple implementation some of them will just return false if the approve is not successful, the contract does not treat this case and it will lead to unpredictable behaviour.

Recommendation:
The SafeERC20 library is already present in the project and it is used with safeTransferFrom and safeTransfer functions, it will be really easy to change the use of approve with the use of safeApprove all over the project

MEDIUM | RESOLVED

TracerGMXVault.sol – In body of openGlpPosition(...), last argument of function RewardRouterV2.mintAndStakeGlp called in last line of function's body represents the amount of glp to be minted in exchange for the assets transferred prior to this mint. Assigning this amount to Zero might lead to wasted funds.

Recommendation:
Priorly retrieve info about how much glp to have in exchange for the asset and make an assessment of the _minGlp amount to put in accordance with that.

MEDIUM | RESOLVED

GlpPricing.sol – usdToGlp multiplication usdAmount * glpPrice(maximise) does not seem to be correct in this case. According to definition of GMX: https://gmxio.gitbook.io/gmx/contracts#glp-price
Buy price: glpManager.getAum(true) / glp.totalSupply()
Assuming Asset Under Management to be usdc, hence we have an amount of usdc per each glp and the result represent glp Price. Multiplying this price to usdAmount does not give the proper result.

Recommendation:
usdAmount / glpPrice(maximise)

MEDIUM | RESOLVED

GlpPricing.sol – glpToUsd same issue as above — recommending replacing the div by mul.

MEDIUM | RESOLVED

BaseVault.sol – mint(uint256 shares, address receiver) gives the caller less shares than stated in function arguments (due to accruing fee). This is not a critical issue because the user is paying the proper amount of asset for the shares s/he receives. Nonetheless, it “looks like” s/he receives less than s/he expected from his/her perspective.
PS: this might be okay if it is considered in the dapp to show to the end user how many shares s/he actually receives.

	BaseVault	VaultStorage
Re-entrancy	Pass	Pass
Access Management Hierarchy	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass
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

	TracerGMXVault	VaultLifeCycle
Re-entrancy	Pass	Pass
Access Management Hierarchy	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass
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

	Vault	TcrPricing
Re-entrancy	Pass	Pass
Access Management Hierarchy	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass
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

	ShareMath	GlpPricing
Re-entrancy	Pass	Pass
Access Management Hierarchy	Pass	Pass
Arithmetic Over/Under Flows	Pass	Pass
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

L2Encoder	
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Secured 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.

```
vault branches
  constructor require statements
    ✓ revert; asset = ZERO_ADDRESS (178ms)
    ✓ revert; feeRecipient = ZERO_ADDRESS (111ms)
    ✓ revert; keeper = ZERO_ADDRESS (75ms)
    ✓ revert; glpManager = ZERO_ADDRESS (126ms)
    ✓ revert; feeRecipient = ZERO_ADDRESS (119ms)
    ✓ revert; keeper = ZERO_ADDRESS (98ms)
    ✓ revert; hedgePricing = ZERO_ADDRESS (99ms)
    ✓ revert; glpPricing = ZERO_ADDRESS (92ms)
    ✓ revert; managemenetFee exceeds limit  (53ms)
    ✓ revert; performanceFee exceeds limit  (52ms)
    ✓ revert; depositFee exceeds limit  (49ms)
    ✓ revert; ZERO minimumSupply (54ms)
    ✓ revert; ZERO cap (51ms)
    ✓ revert; cap <= minimumSupply (54ms)
    ✓ revert; name is empty string (46ms)
    ✓ revert; symbol is empty string
BaseVault
    ✓ #setLeveragePool  (93ms)
    ✓ #setLeveragePool  (42ms)
    ✓ #setLeveragePool  (60ms)
    ✓ #setLeveragePool  (65ms)
    ✓ #deposit -- revert; amount =0  (100ms)
    ✓ #deposit -- revert; amount =0  (39ms)
    ✓ #deposit -- revert; amount =0
    ✓ #initiateWithdraw -- revert; numShares=0
    ✓ #_completeWithdraw -- revert; Not initiated (137ms)
    ✓ #_completeWithdraw -- revert; round not closed (64ms)
```

✓ #initRounds -- revert; numRounds can not be zero (47ms)
✓ #initRounds -- revert; already initialized round (101ms)
✓ #updatePPS -- verify updating price do not take place if flag is false (39ms)

BaseVault

Views

✓ #accountValueBalance (734ms)
✓ #shares (779ms)
✓ #pricePerShare
✓ #decimals
✓ #totalPending
✓ #convertToShares
✓ #convertToAssets
✓ #previewDeposit

External methods

✓ #depositFor (876ms)
✓ #depositFor -- revert; exceeded cap (780ms)
✓ #depositFor -- revert; Insufficient balance (831ms)
✓ #mint -- revert; exceeded cap (824ms)
✓ #mint -- revert; Insufficient balance (815ms)
✓ #mint -- revert; !shares (545ms)
✓ #mint -- revert; address is zero for recipient (517ms)
✓ #migrateToken (157ms)
✓ #migrateToken (137ms)
✓ #migrateToken - revert; not admin (86ms)
✓ #recoverEth - revert; not admin (52ms)
✓ #recoverEth - (74ms)
✓ #transferAsset -- (204ms)
✓ #recoverEth -- revert; failed transfer (318ms)
✓ #transferAsset -- revert; failed transfer (303ms)

Setters

✓ #setNewKeeper -- set by owner (84ms)
✓ #setNewKeeper -- set by non-owner (56ms)
✓ #setNewKeeper -- set ZERO Address (45ms)
✓ #setFeeRecipient -- set by owner (77ms)
✓ #setFeeRecipient -- set by non-owner (50ms)
✓ #setFeeRecipient -- set ZERO Address (45ms)
✓ #setFeeRecipient -- set to old fee recipient (46ms)
✓ #setManagementFee -- set by owner (91ms)
✓ #setManagementFee-- set by non-owner (51ms)
✓ #setManagementFee-- set Invalid value (49ms)
✓ #setPerformanceFee-- set by owner (96ms)
✓ #setPerformanceFee-- set by non-owner (52ms)

✓ #setPerformanceFee-- set Invalid value (57ms)
✓ #setDepositFee-- set by owner (103ms)
✓ #setDepositFee -- set by non-owner (62ms)
✓ #setDepositFee -- set Invalid value (77ms)
✓ #setCap -- set by owner (110ms)
✓ #setCap-- set by non-owner (73ms)
✓ #setCap -- set Invalid value (86ms)
✓ #setCap -- set to very large value (not suited for uint104) (97ms)

DSMath

✓ #add -- normal operation (65ms)
✓ #add -- get its Max (53ms)
✓ #sub-- normal operation (58ms)
✓ #mul-- normal operation (56ms)
✓ #mul-- get its max (56ms)
✓ #min-- normal operation (58ms)
✓ #min-- normal operation (56ms)
✓ #max-- normal operation (71ms)
✓ #max-- normal operation (53ms)
✓ #imin-- normal operation (47ms)
✓ #imin-- normal operation (52ms)
✓ #imax-- normal operation (44ms)
✓ #imax-- normal operation (51ms)
✓ #imin-- signed operation (55ms)
✓ #imin-- signed operation (58ms)
✓ #imax-- signed operation (48ms)
✓ #imax-- signed operation (59ms)
✓ wmul (63ms)
✓ rmul (47ms)
✓ wdiv (57ms)
✓ rdiv (44ms)
✓ dsRpow -- (70ms)
✓ dsRpow -- (63ms)

GlpPricing

✓ #usdToGlp (155ms)
✓ #glpToUsd (157ms)
✓ #assertUint128 -- revert; Overflow uint104
✓ #assertUint128 -- (62ms)
✓ #assertUint128 -- revert; Overflow uint128
✓ #assertUint128 -- (67ms)

L2Encoder

✓ #encodeAddressArray (69ms)

Libraries

ShareMath
- ✓ #assetToShares -- revert; Invalid assetPerShare
- ✓ #assetToShares -- (59ms)
- ✓ #sharesPerAsset -- revert; Invalid assetPerShare
- ✓ #assertUint128 -- revert; Overflow uint128

SafeMath
- ✓ #add -- normal operation (48ms)
- ✓ #add -- get its Max (55ms)
- ✓ #sub-- normal operation (46ms)
- ✓ #mul-- normal operation (56ms)
- ✓ #mul-- get its max (46ms)
- ✓ #add -- revert overflow
- ✓ #mul-- revert overflow
- ✓ #sub-- revert overflow
- ✓ inconsistency due to initial `assetPerShare` being in decimals of `share`

Possible Issues
- ✓ can open a short position using the params
- ✓ User receiving her shares minus fees - unexpectedly
- ✓ initRounds not updating value of numRounds

Proof of Concept Tests -- show issues with potential findings
- ✓ inconsistency due to initial `assetPerShare` being in decimals of `share`

Possible Issues in TracerGMXVault
- ✓ #mint() - User receiving her shares minus fees - unexpectedly (2176ms)
- ✓ #initRounds -- not updating value of numRounds (130ms)

Dealing with Perpetual Pools
- ✓ #claimShorts() -- a supposedly normal case but AutoClaim.claim() fails
- ✓ #queueHedgeRebalance - do queueTracerClose (Hedge Decrease) -- (5756ms)

TcrPricing
- ✓ #sBtcToUsd (394ms)
- ✓ #sEthToUsd (403ms)
- ✓ #usdToSbtc (413ms)
- ✓ #usdToSeth (406ms)
- ✓ #setSEthPrice (171ms)
- ✓ #setSbtcPrice (171ms)

TracerGMXVault

Views
- ✓ #totalAvailableAssets -- no balances yet (1270ms)
- ✓ #totalAvailableAssets -- with balances and prices mocked (1452ms)
- ✓ #getNextLockedQueued -- shareSupply = 0 (1667ms)
- ✓ #rollToNextPosition -- normal operation (5669ms)
- ✓ #rollToNextPosition -- revert; not enough allocation due to initiateWithdraw (5421ms)
- ✓ #initiateWithdraw -- Doing initiateWithdraw on two different rounds (5720ms)

✓ #initiateWithdraw -- Doing initiateWithdraw on two different rounds (6687ms)
✓ #rollToNextPosition -- revert; !allocation (6014ms)
✓ #_rollToNextPosition -- with no vaultfees
✓ #getNextLockedQueued -- shareSupply != 0 | queuedWithdraw = 0
✓ #getNextLockedQueued -- shareSupply != 0 | queuedWithdraw != 0
✓ #getNextLockedQueued -- shareSupply != 0 | queuedWithdraw != 0

Setters

✓ #setLeverageSetIndex -- set by admin (507ms)
✓ #setLeverageSetIndex -- set by non-admin (383ms)
✓ #setLeverageSetIndex -- revert as index has no poolCommitter (110ms)
✓ #setLeverageSetIndex -- revert as index has no poolCommitter (86ms)
✓ #updateHedgePricing -- set by admin (142ms)
✓ #updateHedgePricing -- set by non-admin (86ms)
✓ #updateHedgePricing -- invalid value (111ms)
✓ #updateGlpPricing -- set by admin (150ms)
✓ #updateGlpPricing -- set by non-admin (91ms)
✓ #updateGlpPricing -- invalid value (110ms)
✓ #migrateVault -- called by non-admin (88ms)
✓ #migrateVault -- called by admin (243ms)
✓ #migrateLeverage -- called by non-admin (122ms)
✓ #migrateLeverage -- called by admin (105ms)

short burn and short mint functions

✓ an open a short position using the params (4833ms)
✓ can open a short position using the params using state -- (19671ms)
✓ can close a short position using the params (8507ms)
✓ #queueHedgeRebalance - do queueTracerOpen (Hedge Increase) -- init activeAllocation =0 (5866ms)
✓ #queueHedgeRebalance - revert; !allocation (3006ms)
✓ #queueHedgeRebalance - revert; over allocation (2953ms)
✓ #queueHedgeRebalance - hedge decrease - revert; !allocation (3000ms)
✓ #queueTracerOpen - revert due to unavailable balance (1848ms)
✓ #queueTracerOpen - revert; zero allocation (1778ms)
✓ #queueTracerOpen - revert; zero allocation (1935ms)

functions interacting with RewardRouter

✓ #openGlpPosition -- (3694ms)
✓ #commitAndClose -- without settlePositions (2939ms)
✓ #commitAndClose -- without settlePositions -- new prices verified (4858ms)
✓ #commitAndClose -- settlePositions -- allocation increase but no weth balance (5747ms)
✓ #commitAndClose -- settlePositions -- allocation increase with Weth balance (7076ms)
✓ #commitAndClose -- settlePositions -- allocation decrease (19129ms)

161 passing (29m)

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% Uncovered lines
BaseVault	98.8	95.6	96.97	98.82	430, 539
VaultStorage	100	100	100	100	
TracerGMXVault	98.54	96.15	95.65	98.52	349, 350
VaultLifeCycle	100	100	100	100	
Vault	100	100	100	100	
TcrPricing	92.31	50	100	92.31	
ShareMath	100	62.5	100	100	
GlpPricing	100	100	100	100	
L2Encoder	100	100	100	100	
All files	**98.92**	**95.68**	**97.5**	**98.94**	

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 investment or legal advice, nor should its authors be held accountable for 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.