SMART CONTRACTS REVIEW

zokyo

November 9th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING VAULTKA

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 1 Critical issue: 1 resolved = 0 points deducted
- 2 High issues: 2 resolved = 0 points deducted
- 11 Medium issues: = 3 resolved and 8 acknowledged = 0 points deducted
- 4 Low issues: 4 issues acknowledged = - 4 points each unresolved = -0 points total.
- 9 Informational issues: 2 unresolved, 5 resolved, and 2 acknowledged = -2 points each unresolved = -2 points total.

Thus, 100 - 2 = 98

TECHNICAL SUMMARY

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

The scope of this audit was to analyze and document the Vaultka smart contracts codebase for quality, security, and correctness.

Contract Status

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

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts 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 Vaultka team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Vaultka repository:
Repo: https://github.com/Vaultka-Project/vodkaV2-contracts

Last commit - 44ae5f46d8bb6f950f4ab56067633527657b64de

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

- VodkaV2GMXHandler.sol
- VodkaVaultV2.sol
- Water.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 Vaultka 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. 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 Thorough manual review of the codebase line by line.

Executive Summary

In our audit, we identified a critical severity issue as well as two high-severity issues. Furthermore, we discovered vulnerabilities with medium and low severity, in addition to a few informational issues. For a comprehensive breakdown of these findings, please refer to the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Vaultka team and the Vaultka team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. 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

FINDINGS SUMMARY

#	Title	Risk	Status
1	Unrestricted position fulfillment due to lack of initiator validation	Critical	Resolved
2	Loss of funds by griefing debt repayment	High	Resolved
3	Liquidator Might Lose ETH If Received Tokens Are Less Than debtPortion	High	Resolved
4	User is granted less amount than they should	Medium	Acknowledged
5	Flags can be passed by in case of requestClosePosition	Medium	Resolved
6	Unfair liquidation mechanism adopted	Medium	Acknowledged
7	getLatestData will give wrong results of tokens with decimals !=18	Medium	Acknowledged
8	Protocol's Usability Becomes Limited When Access To Chainlink Oracle Data Feed Is Blocked	Medium	Acknowledged
9	Chainlink's latestRoundData May Return Stale Or Incorrect Result	Medium	Acknowledged
10	getLatestData Does Not Check If Arbitrum Sequencer Is Down In Chainlink Feeds	Medium	Resolved
11	call() Should Be Used Instead Of transfer()	Medium	Resolved
12	Deadline Set To block.timestamp	Medium	Acknowledged
13	amountOutMinimum Hardcoded to 0	Medium	Acknowledged
14	minMarketTokens Hardcoded To 0	Medium	Acknowledged
15	Upgradeability is malfunctioning due to false gap placement	Low	Acknowledged
16	No slippage protection on token swap	Low	Acknowledged

#	Title	Risk	Status
17	Set Upper/Lower Bounds While Assigning maxLeverage And minLeverage	Low	Acknowledged
18	Mapping inCloseProcess not updated	Low	Acknowledged
19	Function argument not validated	Informational	Acknowledged
20	Misleading revert message	Informational	Resolved
21	Lack of events	Informational	Acknowledged
22	Inline if statement does not give options as intended	Informational	Unresolved
23	Loose access modifier	Informational	Resolved
24	Swap fees ceiling limit is low	Informational	Resolved
25	Variable inconsistency	Informational	Unresolved
26	A bit more costly calculation chosen despite availability of alternative	Informational	Resolved
27	Remove/Update Unnecessary Comment	Informational	Resolved

CRITICAL-1 | RESOLVED

Unrestricted position fulfillment due to lack of initiator validation

The VodkaVaultV2 contract holds functions designed to initiate specific operations on the GMX system, essentially starting the whole relevant process. For instance, the user can call the requestOpenPosition function specifying the amount, leverage, and the asset they want to short. Then, this function creates a struct with required data and calls the createDeposit function on GMX's exchangeRouter, passing prepared parameters.

```
        IExchangeRouter.CreateDepositParams memory params =
IExchangeRouter.CreateDepositParams({
            receiver: address(this),
            callbackContract: strategyAddresses.VodkaHandler,
            uiFeeReceiver: msg.sender,
            market: gmp.marketToken,
            initialLongToken: gmp.longToken,
            initialShortToken: gmp.shortToken,
            longTokenSwapPath: new address[](0),
            shortTokenSwapPath: new address[](0),
            minMarketTokens: 0,
            shouldUnwrapNativeToken: false,
            executionFee: gmxOpenCloseFees,
            callbackGasLimit: 2000000
        });

        bytes32 key =
IExchangeRouter(gmxAddresses.exchangeRouter).createDeposit(params);
```

The createDeposit method caches the msg.sender address, which refers to the VodkaVaultV2 contract, and then invokes the createDeposit function on the depositHandler contract

```
  function createDeposit(
      DepositUtils.CreateDepositParams calldata params
  ) external payable nonReentrant returns (bytes32) {
```

```
        address account = msg.sender;

        return depositHandler.createDeposit(
            account,
            params
        );
    }
```

In the final step of the deposit creation routine, the sent parameters, along with the account address, are encoded into a Props struct within the DepositUtils contract.

```
        Deposit.Props memory deposit = Deposit.Props(
            Deposit.Addresses(
                account,
                params.receiver,
                params.callbackContract,
                params.uiFeeReceiver,
                market.marketToken,
                params.initialLongToken,
                params.initialShortToken,
                params.longTokenSwapPath,
                params.shortTokenSwapPath
            ),
```

Privileged actors on the GMX platform review the request and decide whether to accept or revoke it by calling executeDeposit or cancelDeposit functions respectively. If the deposit request is accepted and executed, a callback is made to the callbackContract (set during the request phase) via the afterDepositExecution function, providing a related Props struct as an argument.

```
    function afterDepositExecution(
        bytes32 key,
        Deposit.Props memory deposit,
        EventUtils.EventLogData memory eventData
    ) internal {
        if (!isValidCallbackContract(deposit.callbackContract())) { return; }
```

```
        try IDepositCallbackReceiver(deposit.callbackContract()).afterDepositExecution{
gas: deposit.callbackGasLimit() }(
            key,
            deposit,
            eventData
        ) {
        } catch {
            emit AfterDepositExecutionError(key, deposit);
        }
    }
```

On the Vaultka side, the afterDepositExecution function receives the callback, but without validation to ensure that VodkaVaultV2 was the initiator of the entire process, completely omitting the deposit parameter.

```
    function afterDepositExecution(
        bytes32 key,
        Deposit.Props memory deposit,
        EventUtils.EventLogData memory eventData
    ) external {
        require(
            IRoleStore(RoleStore).hasRole(msg.sender, Role.CONTROLLER),
            "Not proper role"
        );
        IVodkaV2(strategyAddresses.VodkaV2).fulfillOpenPosition(
            key,
            eventData.uintItems.items[0].value
        );
    }
```

Since there's no validation of the initiator in the afterDepositExecution function, an attacker can craft a malicious deposit request, with arbitrary data provided, and set the VodkaHandler contract's address as a callbackContract, effectively bypassing the accounting step included in requestDeposit.
The deposit routine was provided as an example. However, the described vulnerability equally relates to every flow that includes a callback to a function in the VodkaHandler contract

Recommendation:

Implement a check for all callbacks to ensure that the account parameter from the Props struct corresponds to the VodkaVaultV2 address.

Loss of funds by griefing debt repayment

When a deposit request is denied on the GMX side (privileged account triggers cancelDeposit from DepositHandler contract), the afterDepositCancellation callback from VodkaV2GMXHandler is invoked as part of the routine. Within this function, the leverage amount is repaid, and the user set in the deposit request receives native tokens.

```
        IWater(strategyAddresses.WaterContract).repayDebt(
ir.leverageAmount,
            dr.leverageAmount
        );

        payable(dr.user).transfer(eventData.uintItems.items[0].value);
```

However, if the user reverts on receiving the native tokens, it causes the entire transaction to revert. A malicious user could craft a deposit that is intended to be canceled on the GMX side, utilizing the maximum possible leverage, which would necessitate a lend from the water contract. As a result, repaying the outstanding debt becomes impossible, leading to a loss of assets.
Moreover, similar vulnerabilities can arise if the user is blacklisted by USDC token contracts, which can allow specific addresses to be blacklisted at the contract level. If a user is blacklisted, any transfer to or from that user's address will fail.

Recommendation:

- Implement a Pull over Push payment pattern, allowing users to withdraw tokens manually rather than sending them directly.
- Wrap the token transfer in a try/catch block, ensuring that errors are handled.
- Consider implementing a check if the receiver of USDC is not blacklisted.

Liquidator Might Lose ETH If Received Tokens Are Less Than debtPortion

A liquidator can request liquidation of a position by calling the function requestLiquidationPosition at L897 inside VodkaVaultV2. To call this function the liquidator needs to provide ETH (msg.value) which would be sent to the GMX withdraw vault.To fulfil this liquidation request the handler calls the fulfillLiquidation() and if the returnedUSDC is less than the debt then the liquidator is not rewarded with the liquidator reward.

The problem with the flow is that the liquidator is not refunded with the ETH that was sent with the requestLiquidationPosition call like we do in example - fulfillOpenPosition() through setting the tempPayableAddress.

Another edge case that might occur can be , after ETH value sent by the liquidator(say x) is stuck in the vault contract another request by another user for fulfillOpenPosition might occur (and that user sent say y ETH in the requestOpenPosition) , so that user will get x+y ETH back since inside the receive it sends the entire ETH balance to the temp payable user.

User is granted less amount than they should

VodkaVaultV2.sol - In function `fulffillClosePosition()`,`extraData.returnedValue` is being compared to `wr.fullDebtValue + waterProfits`, after that a bigger amount `wr.fullDebtValue + extraData.profits` is subtracted from `extraData.returnedValue`. Look at lines 829 & 834. User in turn is not receiving the proper amount because of the change in `extraData.toLeverageUser`. There is also a serious issue that the calculation in 834 is likely to revert because the subtraction is not protected from underflow (i.e. the if statement does not check correct subtracted amount), therefore function is likely to revert despite call being legitmate.

```
829 if (extraData.returnedValue < (wr.fullDebtValue + waterProfits)) {
830     _positionInfo.liquidator = msg.sender;
831     _positionInfo.liquidated = true;
832     waterRepayment = extraData.returnedValue;
833 } else {
834     extraData.toLeverageUser = (extraData.returnedValue -
wr.fullDebtValue - extraData.profits) + leverageUserProfits;
835     waterRepayment = extraData.returnedValue -
extraData.toLeverageUser - waterProfits;
```

Recommendation:
Either validate the amount with `extraData.profits`, or replace it by `waterProfits` .

Fix: Client made attempts to address the issue in commit `c557b03` .

Client stated the following regarding that issue :
If extraData.returnedValue is slightly greater than`wr.fullDebtValue` then there will be no profit. The return value needs to be greater than position value when position was opened before there will be profit for this the position will be liquidated. And i am sure before this time the keeper would have liquidated the position

Flags can be passed by in case of requestClosePosition

VodkaVaultV2.sol - Function `requestClosePosition` is called with the purpose to close an open position. After `functionfulfillClosePosition` is invoked, position is either liquidated or closed.
Function `requestClosePosition` should not pass through line 718 when called with the same arguments.

For example suppose position is marked liquidated. The issue here is that when position is liquidated, the function can still be invoked because `!_positionInfo.closed` evaluates to true while `!_positionInfo.liquidated` evaluates to false. Or operator in the end evaluates to true and execution goes through the require statement despite that it should not be the case.

```
   function requestClosePosition(
        uint256 _positionID,
        address _user
    ) external payable InvalidID(_positionID, _user) nonReentrant {
        PositionInfo storage _positionInfo = positionInfo[_user]
[_positionID];
718     require(
            !_positionInfo.liquidated || !_positionInfo.closed,
            "VodkaV2: position is closed or liquidated"
        );
```

Recommendation:
Refactor the require statement: `require(!_positionInfo.liquidated && !_positionInfo.closed)`.

MEDIUM-3 | ACKNOWLEDGED

Unfair liquidation mechanism adopted

VodkaVaultV2.sol - No incentive to liquidate a losing position if it is overwhelmed by debt. It is shown in function `fulfillLiquidation()` that liquidator receives reward only if `wr.returnedUSDC >= debtPortion` as when debt is higher (i.e. worse position), the liquidator takes no rewards because the protocol receives whatever it can get in order to compensate for the debt. But that in turn deincentivizes liquidators to get to liquidate such a position leaving the protocol to deal with severe losses.

Recommendation:
Give some portion to liquidator even if the position does not have sufficient return to pay for the whole debt. This in order to stop positions from losing even more.

Fix - Client pledges to deal with that as being part of the project is to have liquidator bots called by owners of project as according to client, they prioritizing the rewards for water users first.

MEDIUM-4 | ACKNOWLEDGED

getLatest Data Will Give Wrong Results for Tokens With Decimals != 18

Inside VaultkaV2GMXHandler to get the price of a token (usdc , index token and long token) we use the function getLatestData at L373. Since USD is represented in 30 decimals , the function normalises USDC as 24 decimals and in the else clause it normalises to 12 decimals , meaning a 18 decimal token.

Problems might arise if long token or index token is a token with decimals != 18 , for instance if the token has 22 decimals , then getLatestData would return the price in 1e12 whereas it should have been in 8 decimals,

This will lead to incorrect values returned by getEstimatedMarketTokenPrice.

Recommendation:
Properly normalise the price according to the decimal of the token.

MEDIUM-5 | ACKNOWLEDGED

Protocol's Usability Becomes Limited When Access To Chainlink Oracle Data Feed Is Blocked

The function getLatestData inside VaultkaV2GMXHandler.sol at L373 takes the address of the token as input , queries the chainlink aggregator to fetch the price and returns the price in correct decimal precision.

As https://blog.openzeppelin.com/secure-smart-contract-guidelines-the-dangers-of-price-oracles/ mentions, it is possible that Chainlink's multisigs can immediately block access to price feeds at will. When this occurs, executing latestRoundData reverts , which causes denial of service if the function getLatestData gets called to fetch the price of the token.

Recommendation:
The logic for getting the token's price (inside getLatestData) from the Chainlink oracle data feed should be placed in the try block while some fallback logic when the access to the Chainlink oracle data feed is denied should be placed in the catch block.

MEDIUM-6 | ACKNOWLEDGED

Chainlink's latestRoundData May Return Stale Or Incorrect Result

Chainlink's `latestRoundData` is used inside getLatestData at L373 to retrieve price feed data, however there is insufficient protection against price staleness.

Return arguments are necessary to determine the validity of the returned price, as it is possible for an outdated price to be received. See here for reasons why a price feed might stop updating.

The return value `updatedAt` contains the timestamp at which the received price was last updated, and can be used to ensure that the price is not outdated. See more information about `latestRoundID` in the Chainlink docs. Inaccurate price data can lead to functions not working as expected and/or lost funds.

Recommendation:
Introduce sufficient checks to ensure correct prices returned.

MEDIUM-7 | RESOLVED

getLatestData Does Not Check If Arbitrum Sequencer Is Down In Chainlink Feeds

When utilizing Chainlink in L2 chains like Arbitrum, it's important to ensure that the prices provided are not falsely perceived as fresh, even when the sequencer is down. This vulnerability could potentially be exploited by malicious actors to gain an unfair advantage.

Recommendation:
It is recommended to follow the code example of Chainlink: https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

MEDIUM-8 | RESOLVED

call() Should Be Used Instead Of transfer()

VaultkaV2GMXHandler makes use of the transfer function at L520 to transfer ETH back to the user.

The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

1. The claimer smart contract does not implement a payable function.
2. The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
3. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Recommendation:
It is recommended to use call() instead of transfer()

MEDIUM-9 | ACKNOWLEDGED

Deadline Set To block.timestamp

Inside VaultkaV2GMXHandler's afterWithdrawalExecution function the ExactInputParams is populated at L553 to perform the necessary swap but the deadline parameter is simply passed in currently as block.timestamp, in which the transaction occurs. This effectively means that the transaction has no deadline; which means that a swap transaction may be included anytime by validators and remain pending in mempool, potentially exposing users to sandwich attacks by attackers or MEV bots.

Recommendation:
The deadline should be set to a correct/safe value throughout the codebase.

MEDIUM-10 | ACKNOWLEDGED

amountOutMinimum Hardcoded to `0`

Inside VaultkaV2GMXHandler's afterWithdrawalExecution function the ExactInputParams is populated at L553 to perform the necessary swap but the amountOutMinimum has been hardcoded to `0`, this can expose users to sandwich attacks due to unlimited slippage.

Recommendation:
Set the slippage to a correct/safe value throughout the codebase.

minMarketTokens Hardcoded To `0`

The value for minMarketTokens at L638 inside VaultkaVaultV2 is hardcoded to `0` , the index token for the market could change in price and alter the price of the market token, making the market token price change unexpectedly causing the vault to receive less market tokens than expected.

Also any deposits/orders that might get executed by the keeper before the deposit may affect the balance of long/short backing tokens in the market and result in more negative impact than expected causing the vault to receive less market tokens than expected.

Recommendation:
Set the minMarketTokens to a correct/safe value throughout the codebase.

LOW-1 | ACKNOWLEDGED

Upgradeability is malfunctioning due to false gap placement

VodkaV2GMXHandler.sol - In that contract __gaps is not placed as the last variable of the declared variables which leads to having the storage mixed up with the upgraded versions of the contract.

```
uint256[50] private __gaps;
 address public WETH;
 uint24 public univ3Fee;
 address public RoleStore;
```

Recommendation:

Place _gaps at the end of variable declarations.

LOW-2 | ACKNOWLEDGED

No slippage protection on token swap

VodkaVaultV2.sol - In body of function `requestClosePosition()` there is a procedure that aims at closing an opened position while the amount of tokens to be received out via gmx is not protected against sudden price change. This is because we have parameters `minLongTokenAmount` and `minShortTokenAmount` are set to zero.

```
760     IExchangeRouter.CreateWithdrawalParams memory params =
IExchangeRouter
            .CreateWithdrawalParams({
                receiver: strategyAddresses.VodkaHandler,
                callbackContract: strategyAddresses.VodkaHandler,
                uiFeeReceiver: _user,
                market: gmp.marketToken,
                longTokenSwapPath: new address[](0),
                shortTokenSwapPath: new address[](0),
                minLongTokenAmount: 0,
                minShortTokenAmount: 0,
                shouldUnwrapNativeToken: false,
                executionFee: gmxOpenCloseFees,
                callbackGasLimit: 2000000
            });
```

Also same finding in VodkaV2GMXHandler.sol, In function afterWihtdrawalExecution() we have:

```
uint256 amountOut;
        if (wr.longToken != WETH) {
            ISwapRouter.ExactInputParams memory params = ISwapRouter
            .ExactInputParams({
                path: abi.encodePacked(wr.longToken, univ3Fee, WETH,
univ3Fee, strategyAddresses.USDC),
                recipient: address(this),
                deadline: block.timestamp,
                amountIn: longAmountFromGMX,
                amountOutMinimum: 0
            });

            amountOut =
ISwapRouter(strategyAddresses.univ3Router).exactInput(params);
        } else {
            ISwapRouter.ExactInputSingleParams memory params =
ISwapRouter
            .ExactInputSingleParams({
                tokenIn: wr.longToken,
                tokenOut: strategyAddresses.USDC,
                fee: univ3Fee,
                recipient: address(this),
                deadline: block.timestamp,
 amountIn: longAmountFromGMX,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            });

            amountOut =
ISwapRouter(strategyAddresses.univ3Router).exactInputSingle(params);
        }
amountOutMinimum is set to 0 in both cases of the if statement.
```

Recommendation:

Put non-zero values into the output amounts to avoid impact of undesirable change of prices. Those values can be passed as arguments in the calling functions.

LOW-3 | ACKNOWLEDGED

Set Upper/Lower Bounds While Assigning maxLeverage And minLeverage

The owner can call the function setMaxandMinLeverage inside VaultkaVaultV2 at L394 to assign or update the values of max and minLeverage. Assigning the MAX_LEVERAGE too low might result in reverts inside the function requestOpenPosition. The calculation for leveragedAmount at L608 would revert if say , amount * leverage < 1000 .

The same goes for assigning DTVLimit DTVSlippage at L334 , the statement at L914 will revert if DTVLimit * DTVslippage < 1000

Recommendation:
While assigning min/maxLeverage proper bounds should be followed.

LOW-4 | ACKNOWLEDGED

Mapping inCloseProcess not updated

VodkaVaultV2.sol - In function `fulfillClosePosition`, we have mapping `inCloseProcess` that allows or denies execution of the function. As `fulfillClosePosition` for a given key is intended to be executed successfully once. It is expected that `inCloseProcess` value for the given user/positionID be updated to `false` but it is not.

```
require(
    inCloseProcess[wr.user][wr.positionID],
    "VodkaV2: close position request not ongoing"
);
```

Recommendation:
Preferably have a three state for that process instead of representing by a bool we can have it represented as: NONE, ONGOING, DONE. Where NONE represents `false`, `ONGOING` represents `true` (i.e. after call to `requestClosePosition()`) and DONE represents the final fulfillment of closing the position.

INFORMATIONAL-1 | ACKNOWLEDGED

Function argument not validated

In VodkaVaultV2.sol - `_fixedFeeSplit` is not being validated to be within an accepted range value.

```
function setProtocolFee(
    address _feeReceiver,
    uint256 _withdrawalFee,
    address _waterFeeReceiver,
    uint256 _liquidatorsRewardPercentage,
    uint256 _fixedFeeSplit
)
```

Recommendation:

Add a `require` statement to validate `_fixedFeeSplit`.

INFORMATIONAL-2 | RESOLVED

Misleading revert message

VodkaVaultV2.sol - In function `setDebtValueRatio()`, a misleading revert message might confuse the operator.

```
require(
    _debtValueRatio <= 1e18,
    "Debt value ratio must be less than 1"
);
```

Recommendation:

Change the message to "Debt value ratio must be less than 1e18".

Lack of events

Some functions that carry out important changes to contracts state emit no events on changing significant parameters of the contracts by admin or other. This is taking place in a number of external and state changing functions in the following files below.

VodkaVaultV2.sol

```
function setDebtValueRatio(uint256 _debtValueRatio, uint256
_timeAdjustment) external onlyOwner
```

VodkaV2GMXHandler.sol

```
function setWETH(address _WETH) external onlyOwner
function setUniv3Fee(uint24 _univ3Fee) external onlyOwner
function setRoleStore(address _roleStore) external onlyOwner
```

Water.sol

```
function setAllowedVault(address _vault, bool _status) external onlyOwner
zeroAddress(_vault)
function setUtilRate(uint256 _utilRate) public onlyOwner
```

Recommendation:
Emit relevant events to announce the changes.

Inline if statement does not give options as intended

VodkaVaultV2.sol - Function `getEstimatedCurrentPosition` there is a distinction made between `_shares` and `_positionInfo.position` as shown here:

```
uint256 userShares = (_shares == 0) ? _positionInfo.position : _shares;
```

But in the only occurence in which this function is invoked we have the following:

```
(uint256 currentPosition, ) = getEstimatedCurrentPosition(
    _positionID,
    _positionInfo.position,
    _user
);
```

Hence, _shares refers to `_positionInfo.position` , therefore, no distinction in the inline if statement which makes the condition being checked not being utilized.

Recommendation:

Another look need to be taken to ensure if `_shares` should refer to `_positionInfo.position` or not.

INFORMATIONAL-5 | RESOLVED

Loose access modifier

VodkaVaultV2.sol - Access modifier is assigned to `public` despite that the following functions are not called in VodkaVaultV2 contract.

It is also mentioned in comments that function `getCurrentLeverageAmount()` is meant to be called by frontend.

```
function getCurrentLeverageAmount(uint256 _positionID, address _user)
public view returns (uint256,uint256)
function getAllUsers() public view returns (address[] memory)
function getTotalOpenPosition(address _user) public view returns
(uint256)
```

Recommendation:
Change access modifier to `external` when it is applicable.

INFORMATIONAL-6 | RESOLVED

Swap fees ceiling limit is low

VodkaV2GMXHandler.sol - The maximum value allowed for univ3Fee is relatively low. If this parameter is lower than what can be accepted by protocol, it leads to swaps ending up failing. As the protocol parameters can be changed from time to time, it shall be better for this contract to adapt to these changes.

```
function setUniv3Fee(uint24 _univ3Fee) external onlyOwner {
     require(_univ3Fee <= 5000, "Fee too high");
     univ3Fee = _univ3Fee;
 }
```

Recommendation:
Allow the parameter to take a higher value as long as it is tokenomically viable.

Fix - Client shows user they are not imposing at anytime a fee that have a high negative impact on them.

INFORMATIONAL-7 | UNRESOLVED

Variable inconsistency

Water.sol - In funcion `deposit()` , developer referred to the caller of the transaction as `msg.sender` , also they used `_msgSender()` (i.e. which returns `msg.sender`). This has no effect in logic of smart contract as they both refer to the same addres but regarding coding style, it is recommended to be consistent and use one way to refer to the caller's address.

```
_deposit(_msgSender(), msg.sender, actualAmount, shares);
```

Recommendation:
Use `msg.sender` in place of `_msgSender()` .

Fix - Addressed by client in commit `5b2f501` . Still there is one occurrence of _msgSender() that causes coding inconsistency .

A bit more costly calculation chosen despite availability of alternative

VodkaVaultV2.sol - At line 674 , a bit costly and repeated computation is being carried out.

```
669     PositionInfo memory _positionInfo = PositionInfo({
670         user: dr.user,
671         deposit: dr.depositedAmount,
672         leverageMultiplier: dr.leverageMultiplier,
673         position: dr.receivedMarketTokens,
674         price: ((dr.depositedAmount * dr.leverageMultiplier/1000) *
1e12) * 1e18 / dr.receivedMarketTokens,
675         liquidated: false,
```

It is worth noting though that the computation has already been carried out in `requestOpenPosition()` using also library function `mulDiv()` which the equation in 674 does not. see the following:

```
uint256 leveragedAmount = amount.mulDiv(_leverage, 1000) - amount;
```

This in turn means that computation in line 674 can be replaced by `dr.leverageAmount + dr.depositedAmount`.

Recommendation:
Same quantity can be retrieved from `dr.leveragedAmount + dr.depositedAmount`.

Remove/Update Unnecessary Comment

The comment at L333 inside VaultkaVaultV2 can be removed since the ownerOwner modifiers have already been applied.

Another example is at L468 inside VaultkaVaultV2 where the comment should be “Debt value ratio must be less than or equal to one” instead of “Debt value ratio must be less than one”

Recommendation:
Remove/Update the unnecessary TODO comment.

	VodkaV2GMXHandler.sol VodkaVaultV2.sol Water.sol
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

We are grateful for the opportunity to work with the Vaultka 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 Vaultka team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo