Dolomite

SMART CONTRACT AUDIT

zokyo

May 23th 2023 | v. 1.0

Security Audit Score

PASS

TECHNICAL SUMMARY

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

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

Contract Status

There were 0 critical issues found during the audit. (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 Leavitt Innovations 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 Leavitt Innovations repository: https://github.com/dolomite-exchange/dolomite-margin-modules

Last commit - 2be952b138f7a6a046620aa72871174193e92201

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

- GLPMathLib.sol
- GLPWrappedTokenUserVaultFactory.sol
- GmxRegistryV1.sol
- GLPPriceOracleV1.sol
- GLPWrappedTokenUserVaultV1.sol
- GLPUnwrapperTraderV1.sol
- GLPWrapperTraderV1.sol
- OnlyDolomiteMargin.sol
- ProxyContractHelpers.sol
- AccountActionLib.sol
- AccountBalanceLib.sol
- WrappedTokenUserVaultFactory.sol
- WrappedTokenUserVaultV1.sol
- WrappedTokenUserVaultUnwrapper.sol
- WrappedTokenUserVaultWrapperTrader.sol
- WrappedTokenUserVaultUpgradeableProxy.sol
- DolomiteMarginMath.sol
- Require.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 Leavitt Innovations 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

No critical issue was found during the audit, alongside one with high severity and some medium, low severity, and informational issues. However, these findings should only affect the contract owner and investors under specific conditions. You can find more information in 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 Leavitt Innovations team and the Leavitt Innovations team is 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 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	Non-Standard ERC20 tokens revert	High	Resolved
2	Potential DOS due to token approval logic	Medium	Resolved
3	Division by zero	Medium	Resolved
4	Logic edge case that allows for 2 account transfers taking place for the same vault	Medium	Resolved
5	Precision Loss in GLPPriceOracleV1	Low	Acknowledged
6	GLPWrappedTokenUserVaultV1 handleRewards() Function Can Accidentally Deposit WETH Into Dolomite Margin Exchange Without Claiming WETH First	Low	Resolved
7	Lack of Validations in GLPMathLib	Low	Resolved
8	Vault approves unlimited gmx	Low	Resolved
9	Unfulfilled todo	Low	Resolved
10	Potential division by zero in GLPMathLib.getGlpMintAmount()	Low	Resolved
11	Missing Input Validation	Low	Resolved
12	Wrong variable scope	Low	Resolved
13	No validation for immutable variables.	Low	Acknowledged
14	Wrong Guard Implementation.	Low	Acknowledged
15	No check for zero address for setters.	Low	Resolved

#	Title	Risk	Status
16	No check for zero amount.	Low	Resolved
17	Mixed assignment between maker and taker tokens	Informational	Resolved
18	Unneeded calculation	Informational	Resolved
19	Functions repeated logic	Informational	Resolved

COMPLETE ANALYSIS

HIGH | RESOLVED

Non-Standard ERC20 tokens revert

WrappedTokenUserVaultWrapperTrader.sol /GLPWrapperTraderV1.sol -

Some non-standard ERC20 tokens always force users to approve to zero first, otherwise they revert. The code does not follow this

The `GLPWrapperTraderV1` contract has a vulnerability related to the usage of USDT as the `_takerToken`. USDT expects each call to `approve` the first zero out of the allowance, and only then a new allowance can be set. However, in the `_exchangeIntoUnderlyingToken` function, the contract only uses `safeApprove` to set the allowance without first zeroing it out, which will cause a revert if the `_takerToken` is USDT.

If the `_takerToken` is USDT, the contract will fail to set the allowance, and thus it will not be able to execute the transaction successfully. This can lead to the user losing their gas fee and potentially losing their opportunity to execute a trade.

Recommendation:
To fix this vulnerability, the contract should first zero out the allowance before calling `safeApprove` to set the new allowance. In the case of USDT.

Note #1: The WrapperTrader contract always approves the amount that is spent by the external contract they call, based on this factor it does not represent a vulnerability.

MEDIUM | RESOLVED

Potential DOS due to token approval logic

GLPWrappedTokenUserVaultV1.sol - In method:

```
function _approveGmxForStaking(IERC20 _gmx) internal {
    address _sGmx = address(sGmx());
    if (_gmx.allowance(address(this), _sGmx) == 0) {
    _gmx.safeApprove(_sGmx, type(uint256).max);
    }
}
```

There is potential denial-of-service (DOS) if `_gmx.allowance(address(this), _sGmx)` is not equal to neither zero nor `type(uint256).max`. In this case, there might not be enough allowance to stake gmx because `safeApprove` is not invoked.

There is a possibility this takes place in a future setup because `GLPWrappedTokenUserVaultV1` is upgradeable (i.e. `WrappedTokenUserVaultUpgradeableProxy`). Assume there is `GLPWrappedTokenUserVaultV2` which implements a separate `approve` on gmx that can alter the amount of allowance to be something other than zero or `type(uint256).max` which leads to the DOS described above.

Recommendation:
Following the recommendation in issue, `Vault approves unlimited gmx` shall in turn solve this one which tells approving just the amount needed for transfer.

Fix - As of the state of commit `9092295` , issue is fixed

```
        uint256 allowance = _gmx.allowance(address(this), _sGmx);
        if (_amount > 0 && allowance > 0) {
            // reset the allowance to 0 if the approval is greater than zero and
there is a non-zero allowance
            _gmx.safeApprove(_sGmx, 0);
        }
```

Division by zero.

In some functions, there are the operations which are divided by the total supply amount of glp.
But there is no validation to ensure that total supply amount of glp is not zero

- GLPMathLib.sol -
  - In getUsdgAmountForSell

```
usdgAmount = _glpAmount * aumInUsdg / glpSupply;
```

- GLPPriceOracleV1.sol -
  - In _getCurrentPrice

```
uint256 rawPrice = glpManager().getAumInUsdg(false) * GLP_PRECISION / glp().totalSupply();
```

Recommendation:
Recommendation: please add validation for total supply amount of glp

Note #1 : This condition should never be triggered because the code is only ever executed when the user has GLP to redeem, which means the total supply is always > 0 when the code is executed. We can add an assert(totalSupply > 0) before calling it though to show acknowledgement of the remote possibility this happens

MEDIUM | RESOLVED

Logic edge case that allows for 2 account transfers taking place for the same vault.

The Dolomite Margin protocol is allowing account transfers between GMX and itself, in the feature specifications it clearly says that there should be only one transfer taking place between a gmx account and a dolomite margin vault, however that is not always true as an edge case in GMX's RewardRouterV2 allows to make 2 transfers, the issue is originating from the function _validateReceiver , this functionality have the role of checking if the receiver is in accordance with all the necessary criterias, however if the original sender is holding just glp tokens and is doing the first transfer, the staking for the tokens will be made at this lines , the updating of the variables averagedStakedAmounts and cummulativeRewards will only take place inside the _updateRewards function which is further called by _stake function, as the _updateRewards is only called before the stakedAmounts is updated, the condition that will allow averageStakedAmounts and cummulativeRewards to be updated will not be trigger for the first account transfer, they will only be updated starting with the second account transfer and only the 3rd account transfer moving forward will fail with error " RewardRouter: feeGlpTracker.averageStakedAmounts > 0 "

Recommendation:
Add a boolean check that will not allow a second account transfer to take place.

Precision Loss in GLPPriceOracleV1

`GLPPriceOracleV1 -`

The `GLPPriceOracleV1` contract implements the `IDolomitePriceOracle` interface, which is used to retrieve the price of a given token. The contract calculates the price of the token based on the price of the `DFS_GLP` token.

The `_getCurrentPrice()` function of the contract is used to calculate the current price of the token. The function calculates the price by multiplying the AUM (Assets Under Management) of the GLP Manager with a precision of 1e18 and dividing it by the total supply of GLP tokens. The function then subtracts the fee from the result to get the final price.

However, the function performs a multiplication on the result of a division, which can lead to precision loss. Specifically, it multiplies the result of `glpManager().getAumInUsdg(false) * GLP_PRECISION` with `fee / FEE_PRECISION`.

For example, suppose the AUM of the GLP Manager is 432670696578359312361727113313744979694 and the total supply of GLP tokens is 460186698499377418335757500. If we calculate the price using `(glpManager().getAumInUsdg(false) * GLP_PRECISION) / glp().totalSupply()`, the result would be 940206872535114501. However, if we use the `_getCurrentPrice()` function, the result would be 940206872535000000, which is a precision loss of 114501.

This precision loss could result in incorrect prices for the token

Recommendation:
To fix this vulnerability, we recommend changing the order of operations in the `_getCurrentPrice()` function to perform the multiplication first and then the division, like so: `glpManager().getAumInUsdg(false) * FEE_PRECISION / glp().totalSupply() * GLP_PRECISION / (FEE_PRECISION - fee)`.

Note #1:The contract uses GLP_PRECISION = 1e18, the division should not result in a significant precision loss in most cases.
When calculating the fee portion and understated price:
return rawPrice - (rawPrice * fee / FEE_PRECISION);
Again, with FEE_PRECISION = 10000, the precision loss should be minimal in most cases.
In conclusion, while there is a theoretical possibility of precision loss in these lines due to integer division, the impact of this Precision loss in GLPPriceOracleV1 should be negligible in practice

LOW | RESOLVED

GLPWrappedTokenUserVaultV1 handleRewards() Function Can Accidentally Deposit WETH Into Dolomite Margin Exchange Without Claiming WETH First

The GLPWrappedTokenUserVaultV1 contract's handleRewards() function can allow the vault owner to accidentally deposit WETH into the Dolomite margin exchange without first claiming the WETH. The issue arises when the “_shouldDepositWethIntoDolomite" parameter is set to true, but the "_shouldClaimWeth” parameter is set to false. This can result in a loss of WETH tokens for the vault owner.

The vulnerability can cause a loss of funds for the vault owner if they accidentally call handleRewards() with the wrong parameters. If "_shouldDepositWethIntoDolomite" is set to true, and "_shouldClaimWeth" is set to false, then WETH tokens will be deposited into the Dolomite margin exchange without first claiming them. The WETH tokens will then be stuck in the exchange, and the vault owner will not be able to recover them. The severity of the impact is medium, as the loss of funds is not guaranteed, but it is still a significant risk for the vault owner.

Recommendation:
To prevent this vulnerability, the handleRewards() function should be updated to check if "_shouldDepositWethIntoDolomite" is true, and if it is, then "_shouldClaimWeth" should also be required to be true. This will prevent the accidental deposit of WETH tokens into the Dolomite margin exchange without first claiming them. Alternatively, a separate function can be created for depositing WETH into the exchange, which would require the vault owner to first claim the WETH before depositing it into the exchange.

Fixed in commit 2be952b

LOW | RESOLVED

Lack of Validations in GLPMathLib

GLPMathLib.sol - Input data is not validated in this contract,

- In `getGlpMintAmount()`: Left not validated: `glpSupply`, having zero `glpSupply` leads to zero `glpAmount` given non-zero `_usdgAmount`
- In `getUsdgAmountForBuy`: Function argument `_inputAmount` is not validated. Also, it is recommended to validate `price` returned in this method's body to assure it is non-zero value.

In `getUsdgAmountForSell`: Variable `aumInUsdg` is not asserted to be non-zero. Given a zero `aumInUsdg` results in having non-zero `_glpAmount` producing zero `usdgAmount`.

Recommendation:
require statement for all the mentioned variables to be asserted that they are not equal to zero.

LOW | RESOLVED

Vault approves unlimited gmx

GLPWrappedTokenUserVaultV1.sol - The contract approves `sGmx` to transfer all the Gmx owned by it on calling

```
function _approveGmxForStaking(IERC20 _gmx) internal {
    address _sGmx = address(sGmx());
    if (_gmx.allowance(address(this), _sGmx) == 0) {
    _gmx.safeApprove(_sGmx, type(uint256).max);
    }
}
```

Unlimited approvals are not recommended, the issue is considered severe if the address of `spender` is an externally-owned-address (EOA) which is not the case here (ps. unless the admin changes the address value of `sGmx` to be an EOA).

Recommendation:
Add argument to the method to be: `_approveGmxForStaking(IERC20 _gmx, uint256 _amount)` in order to approve just the amount needed to be staked each time.

LOW | RESOLVED

Unfulfilled todo

GLPWrappedTokenUserVaultV1.sol - In `handleRewards` a `TODO` note that should address enabling the caller to determine the `toAccountNumber` is undone.

Recommendation:
Add an argument to the method `handleRewards` which can be decoded to put for `toAccountNumber`

LOW | RESOLVED

Potential division by zero in GLPMathLib.getGlpMintAmount()

The `getGlpMintAmount()` function in the `GLPMathLib` library may potentially perform a division by zero when calculating the `glpAmount` variable. Specifically, if the `_usdgAmount` input parameter is zero and the `aumInUsdg` variable returned from the `glpManager()` function is also zero, then the `glpAmount` variable will be set to zero as well. However, if `aumInUsdg` is non-zero, the division `_usdgAmount * glpSupply / aumInUsdg` will result in a division with zero error.

If this division by zero error occurs, it could potentially cause the contract to revert and the function call will not be completed successfully.

Recommendation:
To prevent this error, a check should be added to ensure that `aumInUsdg` is non-zero before performing the division. This can be achieved by adding an additional check at the start of the function as follows:

Note #1: In the contract code is a ternary operator in place that prevents the divide by zero error from occurring when aumInUsdg is `0`. This should not be an issue.

LOW | RESOLVED

Missing Input Validation

In the `GLPWrappedTokenUserVaultV1` contract, the `acceptFullAccountTransfer` function accepts an address parameter `_sender` . Currently, there is no check in place to validate if the `_sender` is not the zero address. This could lead to unexpected behavior or potential loss of funds if the zero address is mistakenly used as the `_sender`

In the GMXRegistry _sGlp, _sGmx, _sbfGmx, _vGlp, and _vGmx addresses should need to validate before setup

Recommendation:
It is strongly recommended to add the proposed validation check in the

Fixed in commit 2be952b

LOW | RESOLVED

Wrong variable scope.

WrappedTokenUserVaultUnwrapper.sol -
_ACTIONS_LENGTH is fixed at compile-time, but it was declared as an immutable

Recommendation:
declare it as constant

No validation for immutable variables.

There are immutable variables in some contracts. If invalid arguments are provided in the constructor, some transactions will be reverted.

- GLPWrappedTokenUserVaultFactory.sol -
  - WETH
- GLPWrapperTraderV1.sol -
  - USDC
- GLPUnwrapperTraderV1.sol -
  - USDC
- WrappedTokenUserVaultFactory.sol -
  - UNDERLYING_TOKEN

Recommendation:
Add the validation

Note: The team plans to add validation checks for immutable variables in future constructors. We've rechecked the provided GMX deployment script and found it satisfactory.
https://github.com/dolomite-exchange/dolomite-margin-modules/blob/master/scripts/deploy-gmx-ecosystem.ts

LOW | ACKNOWLEDGED

Wrong Guard Implementation.

The *assert* was used for validation in some functions.
In these cases, if the check wasn't passed, the user will lose all gas for executing that function.

Recommendation:
please use require instead of assert
GLPWrappedTokenUserVaultV1.sol -

- executeDepositIntoVault

```
assert(_amount1 == underlyingBalanceOf());
```

- executeWithdrawalFromVault

```
assert(_recipient1 != address(this));
```

LOW | RESOLVED

No check for zero address for setters.

In some functions, such as setter functions, there is no check for the zero address.

Recommendation:
please add the zero address check

- `GmxRegistryV1.sol` -
  - `setEsGmx`
  - `setSGlp`
  - `setSGmx`
  - `setSbfGmx`
- `WrappedTokenUserVaultFactory.sol` -
  - `setUserVaultImplementation`
  - `_setIsTokenConverterTrusted`

LOW | RESOLVED

No check for zero amount.

- GLPWrapperTraderV1.sol - in the `getExchangeCost`

There is no check to ensure that the _desiredMakerToken is not zero.
It can cause unnecessary gas consumption.

- GLPMathLib.sol - in the `getUsdgAmountForBuy`

There is no check to ensure that the _inputAmount is not zero.

Recommendation:
please add the zero amount check

Mixed assignment between maker and taker tokens

GLPUnwrapperTraderV1.sol - It is implied from the context of `GLPUnwrapperTraderV1::_exchangeUnderlyingTokenToOutputToken` and `WrappedTokenUserVaultUnwrapperTrader` that `_takerToken` refers to the `VAULT_FACTORY` while `_makerToken` refers to USDC. But we have `getExchangeCost` requiring that `_makerToken` be the VAULT_FACTORY while _takerToken be the USDC. It is worth noting though that the way references to maker and taker are used in this method are not problematic.
Here,

```
uint256 usdgAmount = GLPMathLib.getUsdgAmountForSell(GMX_REGISTRY,
_desiredMakerToken);
```

Library method `getUsdgAmountForSell` expect `_desiredMakerToken` to refer to the amount of glp or its derivatives (i.e. underlying token of factory).
And here,

```
GMX_REGISTRY.gmxVault().getGlpRedemptionAmount(_takerToken, usdgAmount);
```

External method `getGlpRedemptionAmount` expects `_takerToken` to refer to USDC, therefore, the return value is the proper expected value.

Recommendation:
Only minor refactoring without altering the logic of the method. Renaming variables suffice to clear the confusion, or adding comments that describe the roles of maker and taker in the context of this contract and the other related contracts.

INFORMATIONAL | RESOLVED

Unneeded calculation

DolomiteMarginMath.sol - `getPartialRoundUp` and `getPartialRoundHalfUp` in case of zero `numerator` or `target` it calculates `0/denominator` to make it the return value.

Recommendation:
The suggestion is to replace that by `return 0`

INFORMATIONAL | RESOLVED

Functions repeated logic

GLPWrappedTokenUserVaultV1.sol - Functions: `vestGlp, vestGmx` have the exact same logic. So is the case in `unvestGlp, unvestGmx.`

Recommendation:
Implement private functions one for vesting (i.e. `_vestGlpOrGmx`) and one for unvesting (i.e. `_unvestGlpOrGmx`) to replace the implementations in the bodies of those mentioned four methods.

Example:

```
function vestGlp(uint256 _esGmxAmount) external override
onlyVaultOwner(msg.sender) {
    _vestGlpOrGmx(glpOrGmx, _esGmxAmount);      // _vestToken
}
```

	GLPMathLib.sol GLPWrappedTokenUserVaultFactory.sol GmxRegistryV1.sol GLPPriceOracleV1.sol GLPWrappedTokenUserVaultV1.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

	GLPUnwrapperTraderV1.sol GLPWrapperTraderV1.sol OnlyDolomiteMargin.sol ProxyContractHelpers.sol AccountActionLib.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

	AccountBalanceLib.sol WrappedTokenUserVaultFactory.sol WrappedTokenUserVaultV1.sol WrappedTokenUserVaultUnwrapper.sol WrappedTokenUserVaultWrapperTrader.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

	WrappedTokenUserVaultUpgradeableProxy.sol DolomiteMarginMath.sol Require.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 Leavitt Innovations 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 Leavitt Innovations team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo