HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Real Wagmi
Date: 15 Dec, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Real Wagmi
Approved By	Eren Gönen | Solidity SC Auditor at Hacken OÜ Arda Usman | Lead Solidity SC Auditor at Hacken OÜ Grzegorz Trawiński | Lead Solidity SC Auditor at Hacken OÜ
Tags	Borrowing; ERC20; Leverage
Platform	EVM
Language	Solidity
Methodology	Link
Website	http://wagmi.com/
Changelog	01.11.2023 - Initial Review 27.11.2023 - Second Review 14.12.2023 - Third Review 15.12.2023 - Fourth Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Real Wagmi (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Real Wagmi provides a comprehensive leverage contract set, utilizing concentrated liquidity technology. Users can Borrow/Leverage and provide liquidity. It contains the following contracts:

- *Keys* – a library that manages the functionality for adding and removing keys, as well as for computing unique keys based on address inputs, such as borrowing keys and pair keys.
- *ApproveSwapAndPay* – an abstract contract in Solidity is designed to facilitate token swaps and payment processing, with a focus on Uniswap V3. It defines functions for approving tokens for spending, executing token swaps, and handling callback functions from Uniswap V3 swaps. The contract provides mechanisms to ensure approval, validate swap slippage, and make payments, and it also includes functions for Uniswap V3 specific operations, such as calculating pool addresses and handling callback functions.
- *Constants* – a library that contains system constants.
- *ErrLib* – a library that handles system errors.
- *ExternalCall* – a library that handles external call logic used in external swaps.
- *DailyRateAndColleteral* – an abstract contract provides functions to manage and calculate daily loan rates and collateral balances for a lending system. It includes features such as retrieving and updating token rate information, calculating collateral balances, and determining fees based on loan rates.
- *LiquidityManager* – a library that provides functions for computing liquidity amounts when dealing with token amounts and price ranges in a Uniswap V3 pool.
- *Vault* – an implementation of a vault where the owner can transfer tokens to a specified address, and it can also retrieve the balances of multiple tokens held by the contract.
- *LiquidityBorrowingManager* – contract that manages the borrowing liquidity functionality for WAGMI Leverage protocol.

Privileged roles

- The owner of the OwnerSettings contract can update the dailyRateOperator address as well as the liquidations and platform bonuses using the updateSettings() function.

- The owner of the Vault contract can transfer tokens using the transferToken() function.
- The owner of the LiquidityBorrowingManager can collect protocol fees for multiple tokens and can add or remove swap call parameters from the whitelist.
- The operator of the LiquidityBorrowingManager can update the hold token's daily rate.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **6** out of **10**.

- Functional requirements are partially provided.
- Technical description is provided.

Code quality

The total Code Quality score is **8** out of **10**.

- The code contains redundant logic and functions.
    - See the low and informational sections below.
- The development environment is configured.

Test coverage

- The test coverage metric was exceptionally excluded from the final evaluation score due to a bug in the forge coverage tool.

Security score

As a result of the audit, the code contains **3** low severity issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.2**. The system users should acknowledge all the risks summed up in the risks section of the report.

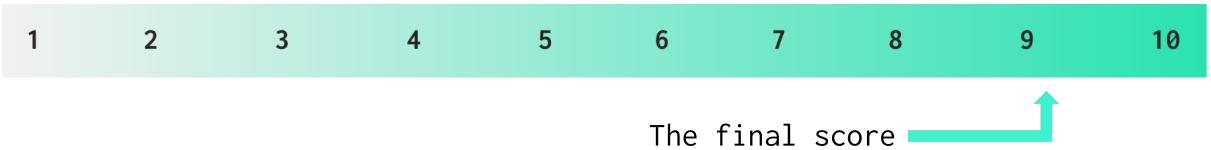

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
01 November 2023	4	4	1	1
27 November 2023	4	1	0	1
14 December 2023	4	0	0	0
15 December 2023	3	0	0	0

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

Critical

No critical issues were found.

High

H01. Temporary Funds Lock Caused by Zero Amount Borrowing

Impact	High
Likelihood	Medium

The *borrow()* function is intended to allow a user to borrow tokens by providing collateral and taking out loans. The function calculates the borrowing details, initializes borrowing variables, and adds borrowing keys and loans information to storage. However, The protocol is designed to prevent borrowing of a zero amount under normal circumstances. If a user provides an empty params.loans list to the *borrow()* function, the protocol will still charge the user and register them with zero values in the borrowing mapping.

This would create a situation where the protocol takes the user's funds and gives zero in return. In other words, this would create a funds lock situation. The only solution would be, in such a case, that the user must re-execute the borrow function with an adjusted params.loan variable so that one will get all the funds, current and pre deposited, back.

Proof of Concept:

1. Bob calls the *borrow()* function with valid parameters but provides an empty params.loans list.
2. The protocol will process the borrowing without reverting, even though the borrowed amount is zero. The protocol will successfully register Bob with zero values to the *borrowingsInfo* mapping.

Path: ./contracts/LiquidityBorrowingManager.sol : borrow()

Recommendation: Add a check in the borrow() function to ensure that the loans array length is greater than zero. If the provided loans array length is zero, the function should revert.

Found in: 72b3360

Status: Fixed (Revised commit: 88b6f14)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Medium

M02. Silent Failures in External Calls due to Inadequate Return Data Size Check

Impact	Medium
Likelihood	Medium

The `_patchAmountAndCall` function within the `ExternalCall` library is designed to execute external calls to other contracts. An oversight in the function's implementation can lead to silent failures under certain conditions. Specifically, the function checks if the external call was unsuccessful and if the returned data size is less than 256. However, there is a scenario where the call might be unsuccessful with a return data size greater than 256, which would not trigger the revert condition.

```
if and(not(success), and(gt(returndatasize(), 0),
lt(returndatasize(), 256))) {
returndatacopy(ptr, 0, returndatasize())
revert(ptr, returndatasize())
}
```

In such cases, the function will not enter the control statement and will not revert, leading to a silent fail. This behavior can have unintended consequences, especially when this function is invoked within the `ApproveSwapAndPay`: `_patchAmountsAndCallSwap()` function, which does not check the return value of `_patchAmountAndCall()`. Even `_patchAmountAndCall()` returns fail since there is no check for the return value function will continue to work.

```
// Patching the amount and calling the external swap
externalSwap.swapTarget._patchAmountAndCall(
externalSwap.swapData,
externalSwap.maxGasForCall,
externalSwap.swapAmountInDataIndex,
amountIn
);
```

Path: ./contracts/libraries/ExternalCall.sol: _patchAmountAndCall(),

./contracts/abstract/ApproveSwapAndPay.sol: _patchAmountsAndCallSwap()

Recommendation: Modify the control statement to check for unsuccessful calls regardless of the return data size. This can be achieved by separating the success check from the return data size

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

check. Additionally, in the ApproveSwapAndPay: *_patchAmountsAndCallSwap()* function, verify the return value of *_patchAmountAndCall()*. If the return value is false, revert the entire call.

Found in: 72b3360

Status: Fixed (Revised commit: c098a5)

M03. Bypass of Maximum Collateral Check in Borrow() Function

Impact	Medium
Likelihood	Medium

The *borrowing()* function in the contract is designed to check if the user is borrowing more collateral than the allowed maximum amount. However, the check is flawed as it relies on a user-provided input *params.maxCollateral*, allowing malicious users to bypass the check by providing the maximum value for a uint256 variable.

In the *borrowing()* function, there is a check to ensure that the borrowing collateral does not exceed the maximum allowed:

```
(borrowingCollateral > params.maxCollateral).revertError(
   ErrLib.ErrorCode.TOO_BIG_COLLATERAL);
```

However, the value of *params.maxCollateral* is provided by the user when they call the function. A malicious user can provide the maximum possible value for a *uint256* variable, effectively bypassing the check and borrowing more than the intended limit.

Path: ./contracts/LiquidityBorrowingManager.sol : borrow()

Recommendation: Instead of relying on user-provided input for the maximum collateral check, use an internally set maximum collateral value. This value should be set during contract deployment or through a secure admin function.

Found in: 72b3360

Status: Fixed (Revised commit: 88b6f14)

Low

L01. Missing Zero Address Validation

Impact	Low
Likelihood	Low

The zero address validation check is not implemented for the following functions:

1. *_patchAmountsAndCallSwap()*
2. *constructor()*
3. *setSwapCallToWhitelist()*

Setting one of aforementioned parameters to zero address (0x0) results in breaking main business flow.

Paths: ./contracts/abstract/ApproveSwapAndPay.sol: _patchAmountsAndCallSwap()

./contracts/abstract/LiquidityManager.sol: constructor(),

./contracts/LiquidityBorrowingManager.sol: setSwapCallToWhitelist()

Recommendation: Implement zero address validation for the given parameters.

Found in: 72b3360

Status: Acknowledged. The client has stated that a fix is not required for this issue.

L03. Missing Event for Critical Value Updation

Impact	Low
Likelihood	Low

Critical state changes should emit events for tracking things off-chain.

This may lead to inability for users to subscribe events and check what is going on with the project.

Paths: ./contracts/abstract/OwnerSettings.sol : updateSettings()

Recommendation: Emit events on critical state changes.

Found in: 72b3360

Status: Fixed (Revised commit: 2c1c80)

L04. Redundant Approval Attempts in _maxApproveIfNecessary() Function

Impact	Low
Likelihood	Low

The *_maxApproveIfNecessary()* function, as it can be seen below, in the contract attempts to set the maximum possible allowance for a spender if the current allowance is less than a specified amount. The function tries to approve the maximum possible value and, if that fails, retries with the maximum possible value minus one. If both

attempts fail, it resets the allowance to zero and repeats the same process. This redundancy is inefficient and leads to unnecessary gas consumption. After the first two attempts to set the maximum allowance fail, the function resets the allowance to zero and tries the same operations again. This is redundant since the same operations have already failed once.

```
function _maxApproveIfNecessary(address token, address spender,
uint256 amount) internal {
    if (IERC20(token).allowance(address(this), spender) <
amount) {
        if (!_tryApprove(token, spender, type(uint256).max)) {
            if (!_tryApprove(token, spender, type(uint256).max
- 1)) {
                require(_tryApprove(token, spender, 0));
                if (!_tryApprove(token, spender,
type(uint256).max)) {
                    if (!_tryApprove(token, spender,
type(uint256).max - 1)) {

true.revertError(ErrLib.ErrorCode.ERC20_APPROVE_DID_NOT_SUCCEED
);
                    }
                }
            }
        }
    }
}
```

Paths: ./contracts/abstract/ApproveSwapAndPay.sol : _maxApproveIfNecessary()

Recommendation: Remove the second code part where the process is tried all over again.

Found in: 72b3360

Status: Acknowledged.

L05. Incorrect Validation in takeOverDebt() Function Leads to Unintended Collateral Rejections

Impact	Medium
Likelihood	Medium

The *takeOverDebt()* function in the contract is designed to allow a user to take over an existing debt of another user. During this process, the new debt owner needs to pay a calculated minimum debt amount, a threshold value, to the system in order to take the ownership.

In the *takeOverDebt()* function, the aforementioned functionality is validated with the following check:

```
(collateralAmt <= minPayment).revertError(
ErrLib.ErrorCode.COLLATERAL_AMOUNT_IS_NOT_ENOUGH);
```

This check means that the collateral amount that is provided by the new borrower, always has to be greater than the minimum calculated value by the system. This means that the minimum amount calculated is not enough to cover the collateral, although it is actually enough.

Since the minimum value is calculated by the system in each debt takeover for each borrowing, the system should accept the collateral amounts that are greater than or equal to the minimum value.

Path: ./contracts/LiquidityBorrowingManager.sol : takeOverDebt()

Recommendation: Replace the “<=” with an “<” in the check above, so that the check would become:

```
(collateralAmt < minPayment).revertError(
ErrLib.ErrorCode.COLLATERAL_AMOUNT_IS_NOT_ENOUGH);
```

Found in: 72b3360

Status: Acknowledged. The client stated that they will not make any changes to this issue.

Informational

I01. Retained NFT Approval After Full Debt Repayment

The contracts maintain approval over a user's NFT position even after the user has fully repaid their debt. This behavior can lead to unintended consequences, especially in emergency situations. Ideally, once a user's debt is fully repaid, the contract should no longer have control over the user's NFT.

When a user repays their debt, the expectation is that any collateral or position they had locked up would be fully returned to them, and any control the contract had over those assets would be relinquished. However, in the current implementation, the contract retains approval over the user's NFT position, which means:

Path: ./contracts/LiquidityBorrowingManager.sol : repay()

Recommendation: Implement a mechanism in the contract to revoke the NFT approval once a user's debt is fully repaid.

Found in: 72b3360

Status: Acknowledged

I02. Commented Code Parts

Following commented code parts were observed:

1. *LiquidityBorrowingManager.sol* lines 450

The presence of commented-out code indicates an unfinished implementation, potentially causing confusion for both developers and users and decreasing code readability.

Path: ./contracts/LiquidityBorrowingManager.sol: takeOverDebt()

Recommendation: Remove commented parts of code.

Found in: 72b3360

Status: Fixed (Revised commit: 88b6f14)

I03. Redundant Logic in Collateral Calculation

The function *calculateCollateralAmtForLifetime()* in the contract is designed to calculate the collateral amount required for a borrowing's lifetime in seconds. However, there is a redundant logic check that ensures the collateral amount is at least 1, even after all the calculations have been performed.

The problematic code snippet is:

```
if (collateralAmt == 0) collateralAmt = 1;
```

The redundant logic can lead to confusion for developers and auditors, and might result in misleading results for the function, such as more Gas consumption.

Path: ./contracts/LiquidityBorrowingManager.sol: calculateCollateralAmtForLifetime()

Recommendation: Remove the redundant code part.

Found in: 72b3360

Status: Acknowledged.

I04. Unnecessary Use of block.timestamp Through a Function

The contract contains a function that returns the *block.timestamp* value. This is redundant since block.timestamp is a global variable in Solidity and can be directly accessed without the need for a

function. Using a function to obtain this value can lead to unnecessary Gas costs.

Path: ./contracts/LiquidityBorrowingManager.sol: _blockTimestamp()

Recommendation: Remove the _blockTimestamp function and directly use block.timestamp wherever needed in the contract.

Found in: 72b3360

Status: Acknowledged

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/RealWagmi/wagmi-leverage
Commit	72b3360
Whitepaper	-
Requirements	-
Technical Requirements	-
Contracts	File: contracts/LiquidityBorrowingManager.sol SHA3: d9ec1c89ac208806acd54d0229f6b205005efe39ef5a6b19f8d87e0615c6c473 File: contracts/Vault.sol SHA3: ba567e17d49ee8d4fec2fb9b1f3dd4b9175310144370c9e85524ae391d69a320 File: contracts/abstract/ApproveSwapAndPay.sol SHA3: 3316fc52accd3e45e57ac108a3a7057b6246fe0781e7d44ecbd7e00d0571c1cc File: contracts/abstract/DailyRateAndCollateral.sol SHA3: bd889ad982fcf7131675c6ae9648f4961557503b78a73e0f9b78678d2ee91c26 File: contracts/abstract/LiquidityManager.sol SHA3: fea05e15e9b1db14ee04469e3844c0ae0107bec840718cd881d3fae6a490daed File: contracts/abstract/OwnerSettings.sol SHA3: cb2533d32eab9091e1102d2b834c62ba499ab03929273eee806c50cd595bf729 File: contracts/libraries/Constants.sol SHA3: 65e1501b19341d0ee7e2f0a2c0a12aec821861055f666d1ed807358468a4a8fb File: contracts/libraries/ErrLib.sol SHA3: 44c4c8ffac2fddc3e30950767d83a08ce4ece6bf1da5dd5d6c8a0360fa16b16c File: contracts/libraries/Keys.sol SHA3: 7a3f8fe5065041a6c91e6531159b73b8ab25e579aa1af11b63b49373c33e79f5 File: contracts/vendor0.8/uniswap/LiquidityAmounts.sol SHA3: c9e1cea28b1ac7cd0029c515481db1d87425450f97c5e85aed4ced6a4177be35

Second review scope

Repository	https://github.com/RealWagmi/wagmi-leverage
Commit	88b6f14
Whitepaper	-
Requirements	-

Technical Requirements	-
Contracts	File: contracts/LiquidityBorrowingManager.sol SHA3: 9005145b8b5046ef41e58f17f07caad00d5c3378ce5854d17cca9e475ba9a23f File: contracts/Vault.sol SHA3: 67ece86ec42df9f7e61870c04ece1f2ffb4a9d231e95d1511fd27df7f9b952ce File: contracts/abstract/ApproveSwapAndPay.sol SHA3: 5ea99f9fcf5d55dc52b593f89b82c8a2f31dd9838a8a0f162e6b60eaf3126360 File: contracts/abstract/DailyRateAndCollateral.sol SHA3: bd889ad982fcf7131675c6ae9648f4961557503b78a73e0f9b78678d2ee91c26 File: contracts/abstract/LiquidityManager.sol SHA3: 2e6ee1acef5005064a3110c9ddb19ec684ea029139ab5f1aa561df1d31f052b2 File: contracts/abstract/OwnerSettings.sol SHA3: cb2533d32eab9091e1102d2b834c62ba499ab03929273eee806c50cd595bf729 File: contracts/libraries/Constants.sol SHA3: 11e0f2a826c9814aa4b81589a0b4fa843c82e3fc4940ffe3814bc1d7b25bf2d0 File: contracts/libraries/ErrLib.sol SHA3: f728ba641bdcc43bb4f986aa20f9cd1869395594d1e711c15566b4c1faccafde File: contracts/libraries/Keys.sol SHA3: 86fe8c42fef3b2941b37264f02f1f507b1242f291cf8e286b0acbb5bd5feca1d File: contracts/libraries/ExternalCall.sol SHA3: 0d69f3c8c3be22c679be91a74cb7a6d4a4dd1d277dd6a8b68251b6b7e887edfd File: contracts/libraries/TransferHelper.sol SHA3: b20a9ccad7e2c5311690c2150563e1d0180e0f856018abf3a8fad8da675c37e8 File: contracts/vendor0.8/uniswap/LiquidityAmounts.sol SHA3: 48b586939c2ccb48953039a6e5f4be7b369440a0695b8c410a8320151ec38a26

Third review scope

Repository	https://github.com/RealWagmi/wagmi-leverage
Commit	c098a59
Whitepaper	-
Requirements	-
Technical Requirements	-
Contracts	File: contracts/LiquidityBorrowingManager.sol SHA3: 9005145b8b5046ef41e58f17f07caad00d5c3378ce5854d17cca9e475ba9a23f File: contracts/Vault.sol SHA3: 67ece86ec42df9f7e61870c04ece1f2ffb4a9d231e95d1511fd27df7f9b952ce File: contracts/abstract/ApproveSwapAndPay.sol

	SHA3: ecff2b245f065e60103557d7044800f9054dede2bc3ac5574e794e6f62f4a275 File: contracts/abstract/DailyRateAndCollateral.sol SHA3: bd889ad982fcf7131675c6ae9648f4961557503b78a73e0f9b78678d2ee91c26 File: contracts/abstract/LiquidityManager.sol SHA3: 2e6ee1acef5005064a3110c9ddb19ec684ea029139ab5f1aa561df1d31f052b2 File: contracts/abstract/OwnerSettings.sol SHA3: cb2533d32eab9091e1102d2b834c62ba499ab03929273eee806c50cd595bf729 File: contracts/libraries/Constants.sol SHA3: 11e0f2a826c9814aa4b81589a0b4fa843c82e3fc4940ffe3814bc1d7b25bf2d0 File: contracts/libraries/ErrLib.sol SHA3: 527e4e13692177468481c54ba75bcf12cc726f1c19564c424c4d2c9324647a30 File: contracts/libraries/ExternalCall.sol SHA3: 0d69f3c8c3be22c679be91a74cb7a6d4a4dd1d277dd6a8b68251b6b7e887edfd File: contracts/libraries/Keys.sol SHA3: 86fe8c42fef3b2941b37264f02f1f507b1242f291cf8e286b0acbb5bd5feca1d File: contracts/libraries/TransferHelper.sol SHA3: b20a9ccad7e2c5311690c2150563e1d0180e0f856018abf3a8fad8da675c37e8 File: contracts/vendor0.8/uniswap/LiquidityAmounts.sol SHA3: 48b586939c2ccb48953039a6e5f4be7b369440a0695b8c410a8320151ec38a26

Fourth review scope

Repository	https://github.com/RealWagmi/wagmi-leverage
Commit	2c1c80e
Whitepaper	-
Requirements	-
Technical Requirements	-
Contracts	File: contracts/LiquidityBorrowingManager.sol SHA3: 9005145b8b5046ef41e58f17f07caad00d5c3378ce5854d17cca9e475ba9a23f File: contracts/Vault.sol SHA3: 67ece86ec42df9f7e61870c04ece1f2ffb4a9d231e95d1511fd27df7f9b952ce File: contracts/abstract/ApproveSwapAndPay.sol SHA3: ecff2b245f065e60103557d7044800f9054dede2bc3ac5574e794e6f62f4a275 File: contracts/abstract/DailyRateAndCollateral.sol SHA3: bd889ad982fcf7131675c6ae9648f4961557503b78a73e0f9b78678d2ee91c26 File: contracts/abstract/LiquidityManager.sol SHA3: 2e6ee1acef5005064a3110c9ddb19ec684ea029139ab5f1aa561df1d31f052b2 File: contracts/abstract/OwnerSettings.sol SHA3: a2cdfff31110b052dffb506e2568ad4b4356e32e8a7d703679a52876f32a8c3e

	File: contracts/libraries/Constants.sol SHA3: 11e0f2a826c9814aa4b81589a0b4fa843c82e3fc4940ffe3814bc1d7b25bf2d0 File: contracts/libraries/ErrLib.sol SHA3: 527e4e13692177468481c54ba75bcf12cc726f1c19564c424c4d2c9324647a30 File: contracts/libraries/ExternalCall.sol SHA3: 0d69f3c8c3be22c679be91a74cb7a6d4a4dd1d277dd6a8b68251b6b7e887edfd File: contracts/libraries/Keys.sol SHA3: 86fe8c42fef3b2941b37264f02f1f507b1242f291cf8e286b0acbb5bd5feca1d File: contracts/libraries/TransferHelper.sol SHA3: b20a9ccad7e2c5311690c2150563e1d0180e0f856018abf3a8fad8da675c37e8 File: contracts/vendor0.8/uniswap/LiquidityAmounts.sol SHA3: 48b586939c2ccb48953039a6e5f4be7b369440a0695b8c410a8320151ec38a26