HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: LombardFi
Date: December 5, 2022

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 LombardFi
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	Lending Platform
Platform	EVM
Language	Solidity
Methodology	Manual Review, Automated Review, Architecture Review
Website	https://lombard.fi/
Changelog	15.11.2022 - Initial Review 05.12.2022 - Second 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 LombardFi (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.

Scope

The scope of the project is smart contracts in the repository:

Initial review scope

Repository	https://github.com/lombardfi/paladin-audit
Commit	5b8adc082ec53e0ee523d743b78a4af460faae56
Whitepaper	-
Functional Requirements	https://docs.lombard.fi/
Technical Requirements	https://docs.lombard.fi/
Contracts Addresses	-
Contracts	File: ./contracts/external/chainlink/Denominations.sol SHA3: 4a03690f5502e69b4896a15e94bcbcb6af8b6ab24489f435240a3277de422428 File: ./contracts/external/chainlink/IAggregatorV3.sol SHA3: 63ee8b551ef16c853ef695b44fecbdab972b23b1284a0919e8efdf11942d17ec File: ./contracts/external/chainlink/IFeedRegistry.sol SHA3: 14af9ef4c2648aeb0ffee4594983484d145b016912f5ec2592be761be399acc6 File: ./contracts/interfaces/IBasePriceOracle.sol SHA3: 312fa92be40efbd8a83427dcfd7c86f1c628061a1610f74529758cb11f205078 File: ./contracts/interfaces/IERC20.sol SHA3: 4b576e889567d39127415745dd39634a28682058697718e02df6dedcfededede05 File: ./contracts/interfaces/IOracleManager.sol SHA3: a9bb7ed37d46987582ae8f203fb453d96a877e1d05cc5cafb81944455d8f3bc6 File: ./contracts/interfaces/IOwnable.sol SHA3: ef8faf71f4999297a6209640ff48318e33a44a56da1f7cc3b22a239f3e87601b File: ./contracts/interfaces/IPool.sol SHA3: e0e0aad4ec4646713831f672af5163bd7ef2206a9900fcc3885dc01e16735076 File: ./contracts/interfaces/IPoolFactory.sol SHA3: 41ae44915ae37ef2f96d5fee8c696e2a1b1bd514bbedccdaf1c5360e785257e1 File: ./contracts/interfaces/IRouter.sol SHA3: 6f97df1d18d67f49e59a931e54ea7979499a3d51a30deb88a2ef27eb3195b628 File: ./contracts/OracleManager.sol SHA3: a0bb25efd208ab46803164836e545491ea11dea4700430ed0c72e64a098cae40 File: ./contracts/oracles/ChainlinkOracleAdapter.sol SHA3: 4d129052fe2fe1f2b50f22aafedee190674fa7c519633d99bab47242cd08bda1 File: ./contracts/Pool.sol

	SHA3: 1066bb5a228d378e3da774255cb6e5efdc7be2e56ff1197bf1536e37ae0f5075 File: ./contracts/PoolFactory.sol SHA3: fd8020e53f8a9921a4848a210578b181d7fd6401d3975b4925b97a2e1899f45c File: ./contracts/Router.sol SHA3: 1fb7594f6a031cce1884802cb98f07c1c0d1de0b35c8e94aa9665c20ead71a5f File: ./contracts/test/CustomERC20.sol SHA3: 1a2eca3651c5b97b707aee3fca9b11aff436214330b73349ccbca86b6f9f9de8 File: ./contracts/test/ERC20Test.sol SHA3: b194344c63c9225612ad9479feae764ab98c52c38ef0edf8e59222f4510cf52a File: ./contracts/test/TestOracle.sol SHA3: 7ad4ae5b1962696957ab17e9ccb19286622514b5d467fae9f948e205af69337d

Second review scope

Repository	https://github.com/lombardfi/paladin-audit
Commit	a8159bf32293cbbd2b9c8c49be8b1ebc49352a88
Whitepaper	-
Functional Requirements	https://docs.lombard.fi/
Technical Requirements	https://docs.lombard.fi/
Contracts Addresses	-
Contracts	File: ./contracts/external/chainlink/Denominations.sol SHA3: 4a03690f5502e69b4896a15e94bcbcb6af8b6ab24489f435240a3277de422428 File: ./contracts/external/chainlink/IAggregatorV3.sol SHA3: 63ee8b551ef16c853ef695b44fecbdab972b23b1284a0919e8efdf11942d17ec File: ./contracts/external/chainlink/IFeedRegistry.sol SHA3: 14af9ef4c2648aeb0ffee4594983484d145b016912f5ec2592be761be399acc6 File: ./contracts/interfaces/IBasePriceOracle.sol SHA3: 7e5130a3db9d5384c0942fa94da5ecb7dee14688ddd6bfa9173a60a64fc49899 File: ./contracts/interfaces/IERC20.sol SHA3: 4b576e889567d39127415745dd39634a28682058697718e02df6dedcfededede05 File: ./contracts/interfaces/IOracleManager.sol SHA3: a9bb7ed37d46987582ae8f203fb453d96a877e1d05cc5cafb81944455d8f3bc6 File: ./contracts/interfaces/IOwnable.sol SHA3: ef8faf71f4999297a6209640ff48318e33a44a56da1f7cc3b22a239f3e87601b File: ./contracts/interfaces/IPool.sol SHA3: fff93aafa41ac144e54bf54a441e37d54a808d8909c55f5b1346e1560087f847 File: ./contracts/interfaces/IPoolFactory.sol SHA3: 061595e6f9e6c978bb70dfb70c4832b4bec0b27ab9f9c8a934c251a20092f5a1 File: ./contracts/interfaces/IRouter.sol SHA3: 19f5f3f744a044a45a776937176d9600f3a696a575d5fe5fa68f1ac4c1504d1b

	File: ./contracts/OracleManager.sol SHA3: 1a0d688a91590fef47cc6814285acc63ffd6103d5f8a5d4423f0b72245d02131 File: ./contracts/oracles/ChainlinkOracleAdapter.sol SHA3: 9c262fa1503d23d297a8b9253b096bd54ff3f7aef59d75012bb2d59720eca7f3 File: ./contracts/Pool.sol SHA3: a4021a3398ff07602264bedc743d309fdad8fb8c92c304f8ff4417b61d9f39c5 File: ./contracts/PoolFactory.sol SHA3: a48df0b040ef4660d497291688bccdc980c455d821b243617100c4f3d51c775c File: ./contracts/Router.sol SHA3: f84cde9b5b085df9a1ac16aca1d332dfdd914aedc42abd1df641b5d3ea9ba21e File: ./contracts/test/CustomERC20.sol SHA3: 1a2eca3651c5b97b707aee3fca9b11aff436214330b73349ccbca86b6f9f9de8 File: ./contracts/test/ERC20Test.sol SHA3: b194344c63c9225612ad9479feae764ab98c52c38ef0edf8e59222f4510cf52a File: ./contracts/test/TestOracle.sol SHA3: 61434e3ca775af10378742f7a1e8b58e0cc4f23a0884c79319d8bff74e35ae04

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions.
Medium	Medium-level vulnerabilities are important to fix; however, they cannot lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution.

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 **9** out of **10**.

- Functional requirements and technical description are provided.
- NatSpec is present in the contracts and describes the code correctly.

Code quality

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

- The development environment is configured.

Test coverage

Test coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.

Security score

As a result of the audit, the code contains **0** critical, **0** high, **0** medium, **0** low severity issues. The security score is 1**0** 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: **9.5**.

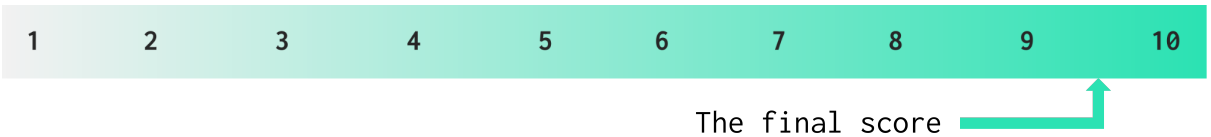

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
10 November 2022	8	4	3	1
05 December 2022	0	0	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and more specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Not Relevant
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Not Relevant
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Passed
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed

Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery	Not Relevant
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Passed
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP standards violation	EIP	EIP standards should not be violated.	Passed
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Passed
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant

Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style guide violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

System Overview

LombardFi is an DeFi protocol that provides functionality for custom, permissionless reputation-based undercollateralized loans. Smart contracts are meant to allow anyone to ask for an undercollateralized loan, leveraging the reputation of the borrower.

The protocol allows for 2 types of actors to interact: lenders and borrowers.
Anyone can become a borrower by deploying a Pool via the PoolFactory.
The borrower must supply a set of parameters, equivalent to a term sheet in the traditional OTC lending world.
All parameters in the term sheet are chosen by the borrower, with the exception of the origination fee (defined at protocol level by the governance) and the start timestamp, which is automatically set.

Lenders deposit funds into the pool, and at the end of the loan, they will be able to withdraw the original funds plus the earned yield. In case of loan default, they will receive the earned yield plus a pro rata portion of the collaterals.

- **OracleManager.sol** - Oracle aggregator that can get prices from multiple oracle adapters. It does not combine prices but has multiple oracle implementations. It currently supports Chainlink.
- **ChainlinkOracleAdapter.sol** - Adapted from the official implementation.
- **Pool.sol** - Immutable Pool implementation contract is created at the PoolFactory contract construction. Every pool is a clone of this contract.
- **PoolFactory.sol -** Deploys Pool contracts.

Privileged roles

- Router: Performs user actions on the pool.
- Governance: Owns Router, PoolFactory, OracleManager and performs administrative operations on them.
  Setters available to the governance on each contract:
  - Router
    - Set pool factory
    - Set oracle manager
    - Set treasury
  - OracleManager
    - Set oracles
  - PoolFactory
    - Set origination fee
    - Set max collateral assets

Risks

- Given the central role that borrower branding plays in this protocol, the client considers ‘malicious borrowers that do not repay’ not an area of concern. The Data Feeds used by the protocol are found through the Chainlink Feed Registry without any further check besides data availability. Currencies provided as collateral are not whitelisted, so the borrower can use as collateral any ERC20 for which exists a Data Feed in Chainlink’s registry.
- The liquidity of collateral assets is always subject to change. One should understand all **risks** of providing a loan secured by such collaterals. Losses may be inevitable in the case of low liquidity of collateral assets.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

1. Ambiguous Third Party Integration

Oracle's response data freshness check is implemented in the wrong way in the *getPrice* function:

```
if (answeredInRound != roundId) { return (false, 0); }
```

Checking *answeredInRound* was the correct way to check data freshness when Chainlink used to implement FluxMonitor jobs. After the switch to Off-Chain Reporting, it is a legacy value that can be ignored.

It is strongly recommended to implement some risk mitigation measure.

Path: ./contracts/ChainlinkOracleAdapter.sol : getPrice()

Recommendation: To check the freshness of the Oracle response, one should check that

block.timestamp - response.updatedAt <= aggregator.heartbeat

using a reasonable value if *aggregator.heartbeat* is not available.

It is suggested to implement risk mitigation measures, like a manual kill switch for selected feeds or currencies.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

2. Inconsistent Data

The contract owner can change an active factory that is used to retrieve pools.

As a result, funds on previous pools will be locked.

Path: ./contracts/Router.sol : setFactory()

Recommendation: Do not allow change of factories after at least one pool is created.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

3. Insufficient Balance

A pool is created by a borrower with arbitrary assets. There is no guarantee that pools with those assets have sufficient liquidity and are not overpriced or completely controlled by the borrower.

A lender can be tricked by such collateral assets.

Path: ./contracts/PoolFactory.sol : createPool()

Recommendation: Add a whitelist of assets that can be used as collateral.

Status: Mitigated (The documentation mention this risk, making it clear that frontends developers are responsible for whitelisting the assets to be used on their platform)

Medium

1. Inefficient Gas Model

Router.sol widely uses OZ ReentrancyGuard, but it can be replaced by an ERC20 whitelist. This would save Gas on lender's operations.

The external calls in Router.sol are all directed to:

- *lentAsset* and *collateralAssets* ERC20 contracts, calling *.decimals()* and IERC20 safeTransfers functions. If these ERC20 contracts cannot be trusted, then Reentrancy is only one of the possible issues, and currencies should be whitelisted.

  Note: risk section mentions this issue.

- Other contracts of this protocol, with no nested external dangerous calls (only Oracle calls or IERC20 transfers)

Path: ./contracts/Router.sol

Recommendation: Replace reentrancyGuard with a whitelist containing the ERC20 currencies allowed to be used on the platform.

Status: Mitigated (The documentation mention this risk, making it clear that frontends developers are responsible for whitelisting the assets to be used on their platform)

2. Requirements Violation

Requirements say that borrowing must happen all at once, while the contracts allow for multiple partial borrowing.

This unexpected feature blends well with the feature that allows repayments to be done in multiple tranches with no side effects. The borrower will pay the full amount of fees whether he borrows the full amount or just a part of it, as the fees are computed on the amount deposited by the lenders.

This feature is well coded and improves the protocol's usability, so the client most likely forgot to update NatSpec and documentation, so this issue is medium severity and not high.

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

Recommendation: Update NatSpec and documentation.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

3. Missing Event

The function *redeem()* handles both the case of a repaid loan and the case of a [partially] defaulted one.

The event emitted is only one, and its parameter does not explicitly include the information on whether the loan defaulted or not. To get this information one would need to check if *Repay* event has been emitted or *pool.borrowed == 0*.

Whether the lender redeemed the entire original sum or not is important information to encode in the event.

```
emit Redeem(_pid, pool.lentAsset());
```

Path: ./contracts/Router.sol : redeem()

Recommendation: For efficient data retrieval, it is advised to encode the missing information (loan fully repaid or not) in Redeem event's parameters.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

4. Redundant Require Statement

The requirement "Caller must not be the pool's borrower" is enforced by *_verifyCallerIsNotBorrower()* at line 317, which is equivalent to *require(notional > 0, "Router::no notional")* at line 330.

Since *_verifyCallerIsNotBorrower()* is used in the deposit function, it is assured that *notional[borrower]* will always be 0.

_verifyCallerIsNotBorrower() at line 317 is thus redundant and can be removed to save Gas.

Path: ./contracts/Router.sol : redeem()

Recommendation: Remove *_verifyCallerIsNotBorrower(pool)* at line 317.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

Low

1. Unused Variable

Variable *router* is set in the constructor and never used.

Unused variables should be removed from the contracts. Unused variables are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease the readability.

Path: ./PoolFactory.sol

Recommendation: Remove unused variable *router*.

Status: Mitigated (client prefers to keep the variable for informational purposes)

2. **Misplaced, Typo in Natspec**

Use of NatSpec that belongs to other blocks or has some typos.

In Router.sol, the NatSpec of *repay()* refers to the arrays *collateralAssets* and *amts*, which do not belong to the function.

In Pool.sol, the variable *originationFee* has a grammar issue in "when borrowing, the borrower the origination fee [...]". The variable *borrowed* has an incorrect description "Borrowers must pay the coupon upfront based on this amount [...]".

In *OracleManager.sol* there is a misplaced *maximum* number of supported oracles instead of *minimum*.

In PoolFactory.sol, the NatSpec in the function *getAllPools()* refers to the function *getAllPoolsUpToPid()*, which should be *getAllPoolsSlice()*.

Paths: ./Router.sol : repay()
./Pool.sol : originationFee, borrowed
./OracleManager.sol: MIN_SUPPORTED_ORACLES
./PoolFactory.sol: getAllPools()

Recommendation: Review and correct the mentioned NatSpec descriptions to describe the functions and variables with precision.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

3. **Redundant Variable**

In Pool.sol, unnecessary variable use/declaration will increase the Gas consumption of the code that should be removed.

Within the function *repay()*, *borrowed* can be used directly without the need to declare a new variable *_borrowed*.

Within the function *redeem()*, *notional* is used unnecessarily, since it can be declared that *uint256 amountToReturn = notional[_src].*

Path: ./Pool.sol : repay() - _borrowed, redeem() - notional

Recommendation: Remove redundant variables.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

4. **No Message in Error Condition**

A *require* statement is missing an error message in PoolFactory.sol, within the function *getAllPoolsSlice()*, in the condition *require(_to >= _from && _to <= pid)*. This makes code harder to test and debug.

Path: ./PoolFactory.sol: getAllPoolsSlice()

Recommendation: Add an error message to the *require* condition.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

5. **Incorrect Error Message**

In Router.sol, the *deposit()* function has the condition *require(block.timestamp < pool.activeAt(), "Router::not active")*. The error message suggests a pool that is not active cannot accept deposits, which contradicts the actual requirement: a deposit can only be made before the pool is active.

Path: ./Router.sol: deposit()

Recommendation: Review and modify the error message to represent the actual *require* condition.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

6. **Unindexed Events**

Having indexed parameters in the events makes it easier to search for these events using indexed parameters as filters. None of the indicated interfaces have indexed events, although they report relevant information.

Paths: ./interfaces/IPool.sol
./interfaces/IPoolFactory.sol
./interfaces/IRouter.sol

Recommendation: Use the *indexed* keyword for the relevant parameters in the events mentioned.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

7. **Missing Zero Address Validation**

Address argument *_quoteAsset* is used as an argument for the constructor of OracleManager.sol without checking against the possibility of *0x0*.

Path: ./OracleManager.sol: constructor().

Recommendation: Implement zero address check.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

8. **Style Guide Violation**

In Router.sol there is a violation of function ordering: *getBorrowingPower()* (*public* visibility) should be above (*private* visibility) *_assetsAreValidPoolCollateral().*

In PoolFactory.sol, there is a violation of function ordering: the function *_assetsAreValid()* has *private* visibility and, therefore, should be placed at the bottom of the contract after the functions with *external* visibility.

In Pool.sol, there is a violation of function ordering: the function *transfer()* has *private* visibility and, therefore, should be placed at the bottom of the contract after the functions with *external* visibility.

Paths: ./Router.sol: getBorrowingPower().
./PoolFactory.sol: _assetsAreValid().
./Pool.sol: transfer().

Recommendation: Implement the function ordering according to the Solidity style guide.

Status: Fixed (Revised commit : a8159bf32293cbbd2b9c8c49be8b1ebc49352a88)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date 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 to 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, Consultant cannot guarantee the explicit security of the audited smart contracts.