SMART CONTRACT AUDIT REPORT

for

HopeLend

Prepared By: Xiaomi Huang

PeckShield

July 30, 2023

Document Properties

Client	HopeLend
Title	Smart Contract Audit Report
Target	HopeLend
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2023	Xuxian Jiang	Final Release
1.0-rc	June 26, 2023	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the HopeLend protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About HopeLend

HopeLend protocol forks from AaveV3 – the popular decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. HopeLend maintains the same core business logic, but reconstructs rewards-related logic, customizes price oracle implementation, as well as redistributes the treasury income. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of HopeLend

Item	Description
Name	HopeLend
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2023

In the following, we show the Git repositories of reviewed files and the commit hash value used in this audit. Note that HopeLend assumes a trusted price oracle with timely market price feeds for supported assets.

- https://github.com/Light-Ecosystem/lend-core.git (97eb3ef)
- https://github.com/Light-Ecosystem/lend-periphery.git (04ba5ce)
- https://github.com/Light-Ecosystem/hope-oracle.git (18ff6bb)

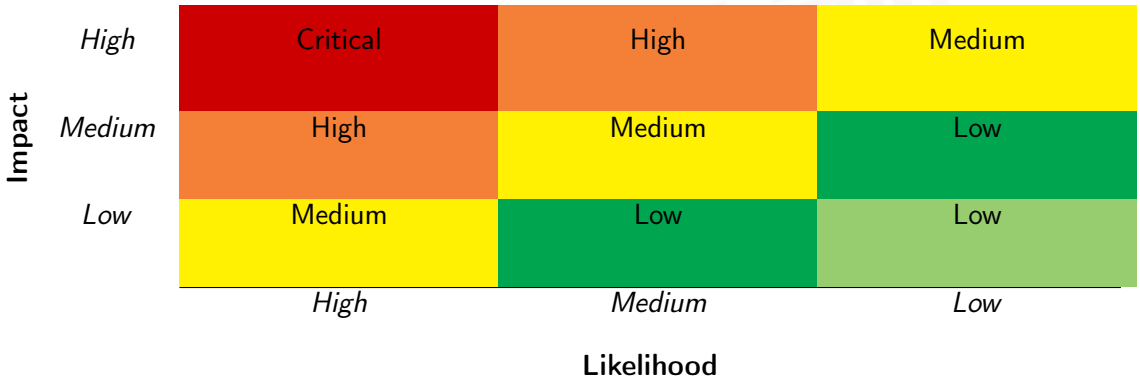
And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Light-Ecosystem/lend-core.git (f53383c)
- https://github.com/Light-Ecosystem/lend-periphery.git (b06b8d)
- https://github.com/Light-Ecosystem/hope-oracle.git (10f42a0)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the HopeLend protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	2	■■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key HopeLend Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Underflow Avoidance in BorrowLogic And UserConfiguration	Coding Practices	Resolved
PVE-002	Low	Timely Rate Adjustment Upon Pool Interest Rate Strategy Change	Business Logic	Confirmed
PVE-003	Informational	Redundant Logic Removal in LendingGauge	Coding Practices	Resolved
PVE-004	High	Incentive Allocation Manipulation with Flashloans	Business Logic	Mitigated
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Underflow Avoidance in BorrowLogic And UserConfiguration

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BorrowLogic, UserConfiguration`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [2]

Description

The `HopeLend` protocol is a decentralized non-custodial money market protocol where there is a constant need of accommodating various precision issues when transfering relevant tokens. And `SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. Since the version `0.8.0`, `Solidity` includes checked arithmetic operations by default, and this largely renders `SafeMath` unnecessary. While reviewing arithmetic operations in current implementation, we notice occasions that may introduce unexpected overflows/underflows.

For example, if we examine the `updateIsolatedDebtIfIsolated()` function, this function updates the isolated debt whenever a position collateralized by an isolated asset is repaid or liquidated. However, when the underlying asset of a reserve has an unusual decimal, which may revert the following calculation of `reserveCache.reserveConfiguration.getDecimals()- ReserveConfiguration.DEBT_CEILING_DECIMALS` (lines 46 – 47). Note this calculation appears in a number of routines. Its revert may bring in unnecessary frictions and cause issues for integration and composability.

```
30  function updateIsolatedDebtIfIsolated(
31    mapping(address => DataTypes.ReserveData) storage reservesData,
32    mapping(uint256 => address) storage reservesList,
33    DataTypes.UserConfigurationMap storage userConfig,
34    DataTypes.ReserveCache memory reserveCache,
35    uint256 repayAmount
```

```
36  ) internal {
37    (bool isolationModeActive, address isolationModeCollateralAddress, ) = userConfig
38      .getIsolationModeState(reservesData, reservesList);
39
40    if (isolationModeActive) {
41      uint128 isolationModeTotalDebt = reservesData[isolationModeCollateralAddress]
42        .isolationModeTotalDebt;
43
44      uint128 isolatedDebtRepaid = (repayAmount /
45        10 **
46          (reserveCache.reserveConfiguration.getDecimals() -
47            ReserveConfiguration.DEBT_CEILING_DECIMALS)).toUint128();
48
49      // since the debt ceiling does not take into account the interest accrued, it
          might happen that amount
50      // repaid > debt in isolation mode
51      if (isolationModeTotalDebt <= isolatedDebtRepaid) {
52        reservesData[isolationModeCollateralAddress].isolationModeTotalDebt = 0;
53        emit IsolationModeTotalDebtUpdated(isolationModeCollateralAddress, 0);
54      } else {
55        uint256 nextIsolationModeTotalDebt = reservesData[isolationModeCollateralAddress
            ]
56          .isolationModeTotalDebt = isolationModeTotalDebt - isolatedDebtRepaid;
57        emit IsolationModeTotalDebtUpdated(
58          isolationModeCollateralAddress,
59          nextIsolationModeTotalDebt
60        );
61      }
62    }
63  }
```

Listing 3.1: `IsolationModeLogic::updateIsolatedDebtIfIsolated()`

Recommendation Revise the above calculation to avoid the unnecessary overflows and underflows.

Status The issue has been fixed by the following commit: `c15d3af`.

3.2 Timely Rate Adjustment Upon Pool Interest Rate Strategy Change

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `Pool`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `HopeLend` protocol allows the governance to dynamically configure the interest rate strategy for current reserves. The supported interest rate strategy implements the calculation of the interest rates depending on the reserve state. While reviewing current configuration logic, we notice the update of the interest rate strategy warrants the need of refreshing the latest stable borrow rate, the latest variable borrow rate, as well as the latest liquidity rate.

To elaborate, we show below the `setReserveInterestRateStrategyAddress()` function. It implements a rather straightforward logic in validating and applying the new `interestRateStrategyAddress` contract. It comes to our attention that the internal accounting for various rates is not timely refreshed to make it immediately effective. As a result, even if the interest rate strategy is already updated, current rates are not updated yet. In other words, the latest stable/variable borrow rate and the latest liquidity rate are still based on the replaced interest rate strategy.

```
561   function setReserveInterestRateStrategyAddress(address asset, address
          rateStrategyAddress)
562     external
563     virtual
564     override
565     onlyPoolConfigurator
566   {
567     require(asset != address(0), Errors.ZERO_ADDRESS_NOT_VALID);
568     require(_reserves[asset].id != 0  _reservesList[0] == asset, Errors.ASSET_NOT_LISTED
            );
569     _reserves[asset].interestRateStrategyAddress = rateStrategyAddress;
570   }
```

Listing 3.2: `Pool::setReserveInterestRateStrategyAddress()`

Recommendation Revise the above logic to apply the give `interestRateStrategyAddress` for the give reserve.

Status

3.3 Redundant Logic Removal in LendingGauge

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: LendingGauge
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The HopeLend protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, and Address, to facilitate its code implementation and organization. For example, the LendingGauge smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the LendingGauge contract, there are two if statements (lines 115 and 121) and the second if statement is redundant (and can be safely removed). The reason is that the second if statement executes after the first one, which implies totalDebt>0. In other words, the following statement is always true: availableLiquidityPlusDebt = availableLiquidity + totalDebt>0, which eliminates the need for the second if statement.

```
111  function updateAllocation() external override returns (bool) {
112    uint256 stableDebtTokenTotalSupply = IERC20(stableDebtToken).totalSupply();
113    uint256 variableDebtTokenTotalSupply = IERC20(variableDebtToken).totalSupply();
114    uint256 totalDebt = stableDebtTokenTotalSupply + variableDebtTokenTotalSupply;
115    if (totalDebt == 0) {
116      borrowAllocation = 0;
117      return true;
118    }
119    uint256 availableLiquidity = IERC20(underlyingAsset).balanceOf(hToken);
120    uint256 availableLiquidityPlusDebt = availableLiquidity + totalDebt;
121    if (availableLiquidityPlusDebt == 0) {
122      borrowAllocation = 0;
123      return false;
124    }
125    borrowAllocation = _getAllocationByUtilizationRate(totalDebt.rayDiv(
         availableLiquidityPlusDebt));
126    return true;
127  }
```

Listing 3.3: LendingGauge::updateAllocation()

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by the following commit: `81ffa34`.

3.4 Incentive Allocation Manipulation with Flashloans

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `LendingGauge`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Deposit and borrowing operations in `HopeLend` will receive corresponding `LT` incentives. Specifically, each reserve in `HopeLend` will have a corresponding deposit certificate `HToken` and a debt token for borrowing. Debt tokens are divided into stable-rate debt and variable-rate debt. By design, the incentives are only for `HToken` and variable-rate debt, not stable-rate debt. The allocation between `HToken` and variable-rate debt is adjusted according to the current utilization rate. Our analysis shows that the current allocation scheme may be vulnerable to flashloan-based manipulation.

To elaborate, we show below the related `updateAllocation()` routine. As the name indicates, this routine updates the debt token allocation ratio based on the current fund utilization rate, which is computed as `totalDebt.rayDiv(availableLiquidityPlusDebt)`. The denominator of `availableLiquidityPlusDebt` is calculated as `IERC20(underlyingAsset).balanceOf(hToken)+ totalDebt`. Apparently, the denominator is affected when a flashloan is borrowed to affect the available `IERC20(underlyingAsset).balanceOf(hToken)`.

```
111   function updateAllocation() external override returns (bool) {
112     uint256 stableDebtTokenTotalSupply = IERC20(stableDebtToken).totalSupply();
113     uint256 variableDebtTokenTotalSupply = IERC20(variableDebtToken).totalSupply();
114     uint256 totalDebt = stableDebtTokenTotalSupply + variableDebtTokenTotalSupply;
115     if (totalDebt == 0) {
116       borrowAllocation = 0;
117       return true;
118     }
119     uint256 availableLiquidity = IERC20(underlyingAsset).balanceOf(hToken);
120     uint256 availableLiquidityPlusDebt = availableLiquidity + totalDebt;
121     if (availableLiquidityPlusDebt == 0) {
122       borrowAllocation = 0;
123       return false;
124     }
125     borrowAllocation = _getAllocationByUtilizationRate(totalDebt.rayDiv(
          availableLiquidityPlusDebt));
126     return true;
127   }
```

Listing 3.4: `LendingGauge::updateAllocation()`

Recommendation Revisit the current approach to compute the fund utilization rate so that it is immune to the above flashloan manipulation.

Status The issue has been mitigated by the following commit: `81ffa34`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `HopeLend` protocol, there are a few privileged `admin` accounts that play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
416   function setUnbackedMintCap(address asset, uint256 newUnbackedMintCap)
417     external
418     override
419     onlyRiskOrPoolAdmins
420   {
421     DataTypes.ReserveConfigurationMap memory currentConfig = _pool.getConfiguration(
          asset);
422     uint256 oldUnbackedMintCap = currentConfig.getUnbackedMintCap();
423     currentConfig.setUnbackedMintCap(newUnbackedMintCap);
424     _pool.setConfiguration(asset, currentConfig);
425     emit UnbackedMintCapChanged(asset, oldUnbackedMintCap, newUnbackedMintCap);
426   }
427
428   /// @inheritdoc IPoolConfigurator
429   function setReserveInterestRateStrategyAddress(address asset, address
        newRateStrategyAddress)
430     external
431     override
432     onlyRiskOrPoolAdmins
433   {
434     DataTypes.ReserveData memory reserve = _pool.getReserveData(asset);
435     address oldRateStrategyAddress = reserve.interestRateStrategyAddress;
436     _pool.setReserveInterestRateStrategyAddress(asset, newRateStrategyAddress);
437     emit ReserveInterestRateStrategyChanged(asset, oldRateStrategyAddress,
          newRateStrategyAddress);
438   }
```

```
439
440   /// @inheritdoc IPoolConfigurator
441   function setPoolPause(bool paused) external override onlyEmergencyAdmin {
442     address[] memory reserves = _pool.getReservesList();
443
444     for (uint256 i = 0; i < reserves.length; i++) {
445       if (reserves[i] != address(0)) {
446         setReservePause(reserves[i], paused);
447       }
448     }
449   }
450
451   /// @inheritdoc IPoolConfigurator
452   function updateBridgeProtocolFee(uint256 newBridgeProtocolFee) external override
          onlyPoolAdmin {
453     require(
454       newBridgeProtocolFee <= PercentageMath.PERCENTAGE_FACTOR,
455       Errors.BRIDGE_PROTOCOL_FEE_INVALID
456     );
457     uint256 oldBridgeProtocolFee = _pool.BRIDGE_PROTOCOL_FEE();
458     _pool.updateBridgeProtocolFee(newBridgeProtocolFee);
459     emit BridgeProtocolFeeUpdated(oldBridgeProtocolFee, newBridgeProtocolFee);
460   }
```

Listing 3.5: Example Privileged Functions in the `PoolConfigurator` Contract

If these privileged `admin` accounts are managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the HopeLend protocol, which forks from AaveV3 – the popular decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. HopeLend maintains the same core business logic, but reconstructs rewards-related logic, customizes price oracle implementation, as well as redistributes the treasury income. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.