# SMART CONTRACT AUDIT REPORT

for

# BETA FINANCE

Prepared By: Shuxiao Wang

PeckShield

May 29, 2021

## Document Properties

| | |
|---|---|
| Client | Beta Finance |
| Title | Smart Contract Audit Report |
| Target | Beta |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jing Wang, Xuxian Jiang |
| Reviewed by | Shuxiao Wang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | May 29, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc1 | May 23, 2021 | Xuxian Jiang | Release Candidate #1 |
| 0.2 | May 12, 2021 | Xuxian Jiang | Add More Findings #1 |
| 0.1 | May 6, 2021 | Xuxian Jiang | Initial Draft |

## Contact

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



| | |
|---|---|
| Name | Shuxiao Wang |
| Phone | +86 173 6454 5338 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the **Beta** design document and related smart contract source code, 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 Beta

The `Beta` protocol is a permissionless money market for lending and borrowing crypto assets that is designed specifically to reduce systematic risk of price volatility for all yield farmers.

The basic information of Beta is as follows:

Table 1.1: Basic Information of Beta

| Item | Description |
|---:|---|
| Issuer | Beta Finance |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | May 29, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Beta assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/beta-finance/beta.git (3e84d6d)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/beta-finance/beta.git (0c55e06)

## 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

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

## 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

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 |

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.

## 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.

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. |

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Beta 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 | 3 | ■ ■ ■ |
| Informational | 0 | |
| 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, and 3 low-severity vulnerabilities.

Table 2.1: Key Beta Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | High | Incorrect Collateral Accounting In BetaBank::put() | Business Logic | Fixed |
| PVE-002 | Low | Improved Sanity Checks Of System/Function Parameters | Coding Practices | Fixed |
| PVE-003 | Low | Suggested Adherence Of Checks-Effects-Interactions Pattern | Time and State | Fixed |
| PVE-004 | Low | Improved Next Interest Rate Calculation | Business Logic | Fixed |
| 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 Incorrect Collateral Accounting In BetaBank::put()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `BetaBank`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

The `Beta` protocol is a permissionless money market for crypto-asset lending and borrowing. For borrowing users, they mainly interact with a core contract `BetaBank`, which provides a number of functionalities, e.g., `open()`, `borrow()`, `repay()`, `put()`, and `take()`. In the following, we examine the `put()` logic.

To elaborate, we show below the implementation of `put()`. It is designed to allow a payer to add more collateral to the given position. (Note the payer must be the position owner or the sender.) It comes to our attention the collateral-updating logic about the given position (line 256) is incorrect: `positions[_owner][_pid].collateralSize += pos.collateralSize`. The proper update logic should be `positions[_owner][_pid].collateralSize = pos.collateralSize`.

```
230  /// @dev Puts more collateral to the given position. Payer must be position owner or
         sender.
231  /// @param _owner The position owner to put more collateral.
232  /// @param _pid The position id to put more collateral.
233  /// @param _amount The amount of collateral to put via `transferFrom`.
234  /// @param _payer The payer to drain collateral token from.
235  function put(
236    address _owner,
237    uint _pid,
238    uint _amount,
239    address _payer
240  ) external override lock {
241    require(_pid < nextPositionIds[_owner], 'put/bad-pid');
```

```
    // 1. pre-conditions
    require(allowActionFor(_owner, msg.sender), 'put/bad-sender');
    require(_payer == msg.sender   _payer == _owner, 'put/bad-payer');
    // 2. transfer collateral tokens in
    Position memory pos = positions[_owner][_pid];
    uint amount;
    {
      uint balBefore = IERC20(pos.collateral).balanceOf(address(this));
      IERC20(pos.collateral).safeTransferFrom(_payer, address(this), _amount);
      uint balAfter = IERC20(pos.collateral).balanceOf(address(this));
      amount = balAfter - balBefore;
    }
    // 3. update the position - no collateral check required
    pos.collateralSize += amount;
    positions[_owner][_pid].collateralSize += pos.collateralSize;
    emit Put(_pid, _amount, _payer);
  }
```

Listing 3.1: BetaBank::put()

**Recommendation** Revise the current `BetaBank::put()` logic to properly reflect the added collateral.

**Status** The issue has been fixed by this commit: `43d232c`.

## 3.2 Improved Sanity Checks For System/Function Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Beta` protocol is no exception. Specifically, if we examine the `BetaConfig` contract, it has defined a number of protocol-wide risk parameters, such as `reserveRate` and `cFactors`. In the following, we show a representative routinethat allow for their changes.

```
  /// @dev Sets the global reserve information.
  function setReserveInfo(address _reserveBeneficiary, uint _reserveRate) external {
    require(msg.sender == governor, 'setReserveInfo/not-governor');
    require(_reserveRate < 1e18, 'setReserveInfo/bad-rate');
    reserveBeneficiary = _reserveBeneficiary;
    reserveRate = _reserveRate;
    emit SetReserveInfo(_reserveBeneficiary, _reserveRate);
```

```
}
```

Listing 3.2: BetaConfig:: setReserveInfo ()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `reserveBeneficiary` may revert every single `accrue()`, hence hurting the adoption of the protocol.

Moreover, the `BetaBank::open()` logic can be improved to validate the presence of the price oracle for the collateral: `require(IBetaOracle(oracle).getAssetETHPrice(_collateral)> 0, 'open/no-price')` . And the given position ID is suggested to be validated before it can be used in a number of position-updating routines, e.g., `_buy()/_repay()/short()` in `BetaRunnerBase`.

```
function open(
  address _owner,
  address _underlying,
  address _collateral
) external override whenNotPaused onlyOwner(_owner) returns (uint pid) {
  address bToken = bTokens[_underlying];
  require(bToken != address(0), 'open/bad-underlying');
  require(_underlying != _collateral, 'open/self-collateral');
  require(IBetaConfig(config).getCollFactor(_collateral) > 0, 'open/bad-collateral');
  require(IBetaOracle(oracle).getAssetETHPrice(_collateral) > 0, 'open/no-price');
  pid = nextPositionIds[_owner]++;
  Position storage pos = positions[_owner][pid];
  pos.bToken = bToken;
  pos.collateral = _collateral;
  emit Open(pid, _owner, bToken, _collateral);
}
```

Listing 3.3: BetaBank::open()

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

**Status** The issue has been fixed by the following commits: `9472543`, `6285ee2`, and `d59ab91`.

## 3.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BToken
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

### Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `BToken` as an example, the `accrue()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 95) starts before effecting the update on internal state (line 102), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
80    /// @dev Accrues interest rate and adjust the rate. Can be called by anyone at any
        time.
81   function accrue() public {
82     if (block.timestamp <= lastAccrueTime) return;
83     address betaBank_ = betaBank; // gas savings
84     require(!IBetaBank(betaBank_).isPaused(), 'BetaBank/paused');
85     // gas savings
86     uint totalLoan_ = totalLoan;
87     uint totalAvailable_ = totalAvailable;
88     uint interestRate_ = interestRate;
89     uint timePast = block.timestamp - lastAccrueTime;
90     IBetaConfig config = IBetaConfig(IBetaBank(betaBank_).config());
91     IBetaInterestModel model = IBetaInterestModel(IBetaBank(betaBank_).interestModel());
92     uint interest = (interestRate_ * totalLoan_ * timePast) / (365 days) / 1e18;
93     totalLoan_ += interest;
94     totalLoan = totalLoan_;
```

```
    interestRate = model.getNextInterestRate(interestRate_, totalAvailable_, totalLoan_,
        timePast);
    if (interest > 0) {
      uint toReserve = (interest * config.reserveRate()) / 1e18;
      address beneficiary = config.reserveBeneficiary();
      _mint(beneficiary, (toReserve * totalSupply()) / (totalLoan_ + totalAvailable_ -
          toReserve));
      emit Accrue(interest);
    }
    lastAccrueTime = block.timestamp;
  }
```

Listing 3.4: BToken::accrue()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions to block possible `re-entrancy`.

**Recommendation** Apply necessary reentrancy prevention by following the well-established `Checks-Effects-Interactions` best practice to block possible `re-entrancy`.

**Status** The issue has been fixed by this commit: `35d92e8`.

## 3.4 Improved Next Interest Rate Calculation

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BToken`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

In the `Beta` protocol, there is a dedicated interest rate contract to compute the new interest rate that immediately becomes effective after current interest accrual. Note the new interest rate is calculated fully on-chain, algorithmically deduced from the following self-explanatory states of the available pools: `prevRate`, `totalAvailable`, `totalLoan`, and `timePast`. In the following, we examine the interest rate calculation logic.

To elaborate, we show below the `accrue()` function that is designed to accrue pending interest and adjust the rate accordingly with the new states. The statement of our focus is the following: `interestRate = model.getNextInterestRate(interestRate, totalAvailable, totalLoan, timePast)` (line 91).

```
80    /// @dev Accrues interest rate and adjust the rate. Can be called by anyone at any
        time.
81    function accrue() public {
82      if (block.timestamp <= lastAccrueTime) return;
83      address betaBank_ = betaBank; // gas savings
84      require(!IBetaBank(betaBank_).isPaused(), 'BetaBank/paused');
85      // gas savings
86      uint totalLoan_ = totalLoan;
87      uint totalAvailable_ = totalAvailable;
88      uint interestRate_ = interestRate;
89      uint timePast = block.timestamp - lastAccrueTime;
90      IBetaConfig config = IBetaConfig(IBetaBank(betaBank_).config());
91      IBetaInterestModel model = IBetaInterestModel(IBetaBank(betaBank_).interestModel());
92      uint interest = (interestRate_ * totalLoan_ * timePast) / (365 days) / 1e18;
93      totalLoan_ += interest;
94      totalLoan = totalLoan_;
95      interestRate = model.getNextInterestRate(interestRate_, totalAvailable_, totalLoan_,
          timePast);
96      if (interest > 0) {
97        uint toReserve = (interest * config.reserveRate()) / 1e18;
98        address beneficiary = config.reserveBeneficiary();
99        _mint(beneficiary, (toReserve * totalSupply()) / (totalLoan_ + totalAvailable_ -
            toReserve));
100       emit Accrue(interest);
101     }
102     lastAccrueTime = block.timestamp;
103   }
```

Listing 3.5: BToken::accrue()

It comes to our attention that the `totalLoan` used for the interest rate calculation is based on the current loan without adding the pending interest. As a result, the utilization rate is slightly smaller to be reflected in the new interest rate calculation. With that, we suggest to replace `totalLoan` (line 91) with `totalLoan+interest`.

**Recommendation** Adjust the given arguments for the new interest rate calculation in `accrue()`.

**Status** The issue has been fixed by this commit: `5f9557a`.

## 3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BetaConfig`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

## Description

In the Beta protocol, there is a special administrative account, i.e., governor. This governor account plays a critical role in governing and regulating the system-wide operations (e.g., authorizing other roles, setting various parameters, and adjusting external oracles). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the governor account is indeed privileged. In the following, we show representative privileged operations in the Beta protocol.

```
/// @dev Sets the risk configurations of the given levels.
function setRiskConfigs(uint[] calldata levels, RiskConfig[] calldata configs) external {
  require(msg.sender == governor, 'setRiskConfigs/not-governor');
  require(levels.length == configs.length, 'setRiskConfigs/bad-length');
  for (uint idx = 0; idx < levels.length; idx++) {
    require(configs[idx].safetyLTV <= 1e18, 'setRiskConfigs/bad-safety-ltv');
    require(configs[idx].liquidationLTV <= 1e18, 'setRiskConfigs/bad-liquidation-ltv');
    require(
      configs[idx].safetyLTV < configs[idx].liquidationLTV,
      'setRiskConfigs/inconsistent-ltv-values'
    );
    require(configs[idx].killBountyRate <= 1e18, 'setRiskConfigs/bad-kill-reward-factor');
    rConfigs[levels[idx]] = configs[idx];
    emit SetRiskConfig(
      levels[idx],
      configs[idx].safetyLTV,
      configs[idx].liquidationLTV,
      configs[idx].killBountyRate
    );
  }
}

/// @dev Sets the global reserve information.
function setReserveInfo(address _reserveBeneficiary, uint _reserveRate) external {
  require(msg.sender == governor, 'setReserveInfo/not-governor');
  require(_reserveRate < 1e18, 'setReserveInfo/bad-rate');
  require(_reserveBeneficiary != address(0), 'setReserveInfo/bad-beneficiary');
  reserveBeneficiary = _reserveBeneficiary;
  reserveRate = _reserveRate;
  emit SetReserveInfo(_reserveBeneficiary, _reserveRate);
}
```

Listing 3.6: Various Setters in BetaConfig

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the governor is not governed by a DAO-like structure. The discussion with the team has confirmed that the governance is currently managed by a multi-sig account. We point out that a compromised governor account would allow the attacker to undermine necessary assumptions behind the protocol and subvert various protocol operations.

**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.

# 4 Conclusion

In this audit, we have analyzed the Beta design and implementation. The system presents a unique, robust offering as a decentralized non-custodial money market for lending and borrowing crypto assets that is designed specifically to reduce systematic risk of price volatility for all yield farmers. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.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.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.