SMART CONTRACT AUDIT REPORT

for

Goledo

Prepared By: Xiaomi Huang

PeckShield

October 12, 2022

Document Properties

Client	Goledo
Title	Smart Contract Audit Report
Target	Goledo
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 12, 2022	Shulin Bie	Final Release
1.0-rc	August 9, 2022	Shulin Bie	Release Candidate

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 Goledo 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 Goledo

Goledo is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., AAVE. The protocol allows users to 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. The protocol extends the original version with new features for staking-based incentivization and fee distribution.

Table 1.1: Basic Information of Goledo

Item	Description
Target	Goledo
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 12, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Goledo/goledo-core.git (d2b6dd9)

1.2 About PeckShield

PeckShield Inc. [12] 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 OWASP Risk Rating Methodology [11]:

- 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 list of check 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 List of Check Items

Category	Check Item
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) [10], 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.

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 Logics	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 `Goledo` implementation. 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	0	
Medium	3	■■■
Low	2	■■
Informational	1	■
Total	6	

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 that 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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Goledo Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-002	Medium	Revisited Reentrancy Protection In Current Implementation	Time and State	Confirmed
PVE-003	Medium	Incentive Inconsistency Between AToken And StableDebtToken	Business Logic	Confirmed
PVE-004	Informational	Immutable States If Only Set At Constructor()	Coding Practices	Confirmed
PVE-005	Low	Fork-Compliant Domain Separator In AToken	Business Logic	Confirmed
PVE-006	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 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Goledo protocol, the LendingPool contract is designed to be the main entry for interaction with borrowing/lending users. In particular, one entry routine, i.e., deposit(), accepts asset transfer-in and mints the corresponding AToken to represent the depositor's share in the lending pool. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
  function deposit(
    address asset,
    uint256 amount,
    address onBehalfOf,
    uint16 referralCode
  ) external override whenNotPaused {
      DataTypes.ReserveData storage reserve = _reserves[asset];

      ValidationLogic.validateDeposit(reserve, amount);

      address aToken = reserve.aTokenAddress;

      reserve.updateState();
      reserve.updateInterestRates(asset, aToken, amount, 0);

      IERC20(asset).safeTransferFrom(msg.sender, aToken, amount);
```

```

        bool isFirstDeposit = IAToken(aToken).mint(onBehalfOf, amount, reserve.
            liquidityIndex);

        if (isFirstDeposit) {
            _usersConfig[onBehalfOf].setUsingAsCollateral(reserve.id, true);
            emit ReserveUsedAsCollateralEnabled(asset, onBehalfOf);
        }

        emit Deposit(asset, msg.sender, onBehalfOf, amount, referralCode);
    }
```

Listing 3.1: `LendingPool::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Goledo`. In `Goledo` protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Note that other routines, i.e., `MasterChef::deposit()` and `MultiFeeDistribution::stake()`, share the similar issue.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status The issue has been confirmed by the team. There is no need to support deflationary/rebasing tokens.

3.2 Revisited Reentrancy Protection In Current Implementation

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact:High
- Target: Multiple Contracts
- Category: Time and State [9]
- CWE subcategory: CWE-682 [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` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

In the `MasterChef` contract, we notice the `deposit()` routine has potential reentrancy risk. To elaborate, we show below the related code snippet of the `deposit()` routine. In the `deposit()` routine, we notice `IERC20(_token).safeTransferFrom(address(msg.sender), address(this), _amount)` (line 222) will be called to transfer the underlying assets into the `MasterChef` contract. If the `_token` faithfully implements the ERC777-like standard, then the `deposit()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining `GasTokens`).

In our case, the above hook can be planted in `IERC20(_token).safeTransferFrom(address(msg.sender), address(this), _amount)` (line 222) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `user.rewardDebt` intact (used for the calculation of pending rewards at line 217). With a lower `user.rewardDebt`, the re-entered `deposit()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains.

```
208     function deposit(address _token, uint256 _amount) external {
209         PoolInfo storage pool = poolInfo[_token];
```

```
        require(pool.lastRewardTime > 0);
        _updateEmissions();
        _updatePool(_token, totalAllocPoint);
        UserInfo storage user = userInfo[_token][msg.sender];
        uint256 userAmount = user.amount;
        uint256 accRewardPerShare = pool.accRewardPerShare;
        if (userAmount > 0) {
            uint256 pending = userAmount.mul(accRewardPerShare).div(1e12).sub(user.
                rewardDebt);
            if (pending > 0) {
                userBaseClaimable[msg.sender] = userBaseClaimable[msg.sender].add(
                    pending);
            }
        }
        IERC20(_token).safeTransferFrom(address(msg.sender), address(this), _amount);
        userAmount = userAmount.add(_amount);
        user.amount = userAmount;
        user.rewardDebt = userAmount.mul(accRewardPerShare).div(1e12);
        if (pool.onwardIncentives != IOnwardIncentivesController(0)) {
            uint256 lpSupply = IERC20(_token).balanceOf(address(this));
            pool.onwardIncentives.handleAction(_token, msg.sender, userAmount, lpSupply)
                ;
        }
        emit Deposit(_token, msg.sender, _amount);
    }
```

Listing 3.2: `MasterChef::deposit()`

We observe the current implementation of the `MasterChef` and `MultiFeeDistribution` contracts haven't considered reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been confirmed by the team.

3.3 Incentive Inconsistency Between AToken And StableDebtToken

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AToken/StableDebtToken`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Goledo` protocol extends the built-in `IncentivesController` framework to engage protocol users. While reviewing the logic to integrate the incentive mechanism, we observe unnecessary inconsistency that may introduce unwanted confusion and errors.

To elaborate, we show below the related code snippet of the `IncentivizedERC20::_mint()` and `StableDebtToken::_mint()` routines. It comes to our attention that the first routine uses the post-update balance in the invocation of `IncentivesController::handleAction()` (line 200), while the second routine uses the pre-update balance in the invocation of `IncentivesController::handleAction()` (line 408).

```
188     function _mint(address account, uint256 amount) internal virtual {
189         require(account != address(0), "ERC20: mint to the zero address");
190 
191         _beforeTokenTransfer(address(0), account, amount);
192 
193         uint256 currentTotalSupply = _totalSupply.add(amount);
194         _totalSupply = currentTotalSupply;
195 
196         uint256 accountBalance = _balances[account].add(amount);
197         _balances[account] = accountBalance;
198 
199         if (address(_getIncentivesController()) != address(0)) {
200             _getIncentivesController().handleAction(account, accountBalance,
                    currentTotalSupply);
201         }
202     }
```

Listing 3.3: `IncentivizedERC20::_mint()`

```
399     function _mint(
400         address account,
401         uint256 amount,
402         uint256 oldTotalSupply
403     ) internal {
404         uint256 oldAccountBalance = _balances[account];
405         _balances[account] = oldAccountBalance.add(amount);
406 
407         if (address(_incentivesController) != address(0)) {
408             _incentivesController.handleAction(account, oldAccountBalance,
                    oldTotalSupply);
409         }
410     }
```

Listing 3.4: `StableDebtToken::_mint()`

Recommendation Be consistent in using the account balance for incentivization measurement.

Status The issue has been confirmed by the team.

3.4 Immutable States If Only Set At Constructor()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `Goledo` protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
contract ChefIncentivesController is Ownable {
    ...

    address public poolConfigurator;

    IMultiFeeDistribution public rewardMinter;
}
```

Listing 3.5: `ChefIncentivesController`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been confirmed by the team.

3.5 Fork-Compliant Domain Separator In AToken

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `AToken`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `AToken` token contract strictly follows the widely-accepted ERC20 specification. In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `initialize()` function (lines 78-80).

```
61      function initialize(
62          ILendingPool pool,
63          address treasury,
64          address underlyingAsset,
65          IAaveIncentivesController incentivesController,
66          uint8 aTokenDecimals,
67          string calldata aTokenName,
68          string calldata aTokenSymbol,
69          bytes calldata params
70      ) external override initializer {
71          uint256 chainId;
72
73          //solium-disable-next-line
74          assembly {
75          chainId := chainid()
76          }
77
78          DOMAIN_SEPARATOR = keccak256(
79          abi.encode(EIP712_DOMAIN, keccak256(bytes(aTokenName)), keccak256(
                EIP712_REVISION), chainId, address(this))
80          );
81
82          ...
83      }
```

Listing 3.6: `AToken::initialize()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed

DOMAIN_SEPARATOR, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        require(owner != address(0), "INVALID_OWNER");
        //solium-disable-next-line
        require(block.timestamp <= deadline, "INVALID_EXPIRATION");
        uint256 currentValidNonce = _nonces[owner];
        bytes32 digest = keccak256(
        abi.encodePacked(
            "\x19\x01",
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, currentValidNonce, deadline))
        )
        );
        require(owner == ecrecover(digest, v, r, s), "INVALID_SIGNATURE");
        _nonces[owner] = currentValidNonce.add(1);
        _approve(owner, spender, value);
    }
```

Listing 3.7: AToken::permit()

Recommendation Recalculate the value of DOMAIN_SEPARATOR inside the permit() function.

Status The issue has been confirmed by the team.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the Goledo protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setOnwardIncentives(address _token, IOnwardIncentivesController _incentives
        ) external onlyOwner {
        require(poolInfo[_token].lastRewardTime != 0);
        poolInfo[_token].onwardIncentives = _incentives;
    }

    function setClaimReceiver(address _user, address _receiver) external {
        require(msg.sender == _user  msg.sender == owner());
        claimReceiver[_user] = _receiver;
    }
```

Listing 3.8: `ChefIncentivesController`

Moreover, the `LendingPoolAddressesProvider` contract allows the privileged `owner` to configure protocol-wide contracts, including `LENDING_POOL`, `LENDING_POOL_CONFIGURATOR`, `POOL_ADMIN`, `EMERGENCY_ADMIN`, `LENDING_POOL_COLLATERAL_MANAGER`, `PRICE_ORACLE`, and `LENDING_RATE_ORACLE`. These contracts play a variety of duties and are also considered privileged.

```
    contract LendingPoolAddressesProvider is Ownable, ILendingPoolAddressesProvider {
        string private _marketId;
        mapping(bytes32 => address) private _addresses;

        bytes32 private constant LENDING_POOL = 'LENDING_POOL';
        bytes32 private constant LENDING_POOL_CONFIGURATOR = 'LENDING_POOL_CONFIGURATOR'
            ;
        bytes32 private constant POOL_ADMIN = 'POOL_ADMIN';
        bytes32 private constant EMERGENCY_ADMIN = 'EMERGENCY_ADMIN';
        bytes32 private constant LENDING_POOL_COLLATERAL_MANAGER = 'COLLATERAL_MANAGER';
        bytes32 private constant PRICE_ORACLE = 'PRICE_ORACLE';
        bytes32 private constant LENDING_RATE_ORACLE = 'LENDING_RATE_ORACLE';
        ...
    }
```

Listing 3.9: `LendingPoolAddressesProvider`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the `Goledo` design and implementation. `Goledo` is a decentralized non-custodial liquidity markets protocol that is developed on top of one of the largest DeFi protocols, i.e., `AAVE`. The current implementation extends the original `AAVE` with new features for staking-based incentivization and fee distribution. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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