SMART CONTRACT AUDIT REPORT

for

Geist Protocol

Prepared By: Yiqun Chen

PeckShield

November 23, 2021

Document Properties

Client	Geist
Title	Smart Contract Audit Report
Target	Geist
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Stephen Bie, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 23, 2021	Xuxian Jiang	Final Release
1.0-rc	November 21, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
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 `Geist` 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 Geist

`Geist` 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.

The basic information of Geist is as follows:

Table 1.1: Basic Information of Geist

Item	Description
Name	Geist
Website	https://geist.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 23, 2021

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

- https://github.com/geist-finance/geist-protocol.git (b6b13bd)

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

- https://github.com/geist-finance/geist-protocol.git (1cfcb20)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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.

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

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) [11], 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

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.

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 `Geist` 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	0	
Medium	2	■■
Low	4	■■■■
Informational	1	■
Undetermined	1	■
Total	8	

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 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Geist Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper answer Type in ChainlinkResponse	Coding Practice	Resolved
PVE-002	Low	Improved Oracle Status in PriceFeed::_fetchPrice()	Business Logic	Resolved
PVE-003	Low	Inconsistent Use Of IncentivesController::handleAction()	Business Logic	Resolved
PVE-004	Informational	ERC20 Compliance Of GeistToken	Coding Practice	Resolved
PVE-005	Medium	Fork-Resistant Domain Separator in AToken	Business Logic	Resolved
PVE-006	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-007	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Proper answer Type in ChainlinkReponse

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PriceFeed
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

The Geist protocol has strengthened the oracle reliability and security with the so-called dual price oracle, i.e., it makes use of not only the Chainlink's live ETH:USD aggregator reference contract, but also bandOracle with the connection to the BandMaster contract. While reviewing the integration with the Chainlink-based oracle, we notice the internal ChainlinkResponse data structure needs to be improved.

In the following, we show the definition of the ChainlinkResponse data structure. Notice the answer member field is defined as uint256, not int256. If we examine the criteria used in _badChainlinkResponse () to determine whether the Chainlink response is considered bad, it explicitly considers non-positive answer. In other words, there is a type inconsistency in the answer member field.

```
    struct ChainlinkResponse {
        uint80 roundId;
        uint256 answer;
        uint256 timestamp;
        bool success;
        uint8 decimals;
    }
```

Listing 3.1: The PriceFeed::ChainlinkResponse Structure

```
    function _badChainlinkResponse(ChainlinkResponse memory _response) internal view
        returns (bool) {
         // Check for response call reverted
```

```
        if (!_response.success) {return true;}
        // Check for an invalid roundId that is 0
        if (_response.roundId == 0) {return true;}
        // Check for an invalid timeStamp that is 0, or in the future
        if (_response.timestamp == 0  _response.timestamp > block.timestamp) {return
            true;}
        // Check for non-positive price
        if (_response.answer <= 0) {return true;}

        return false;
    }
```

Listing 3.2: `PriceFeed::_badChainlinkResponse()`

Recommendation Resolve the type inconsistency in the `answer` member field by defining it as an `int256`, not `uint256`.

Status The issue has been fixed by this commit: `8702175`.

3.2 Improved Oracle Status in PriceFeed::_fetchPrice()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PriceFeed`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

As mentioned earlier, the `Geist` protocol is unique in supporting the dual price oracle, which necessitates the examination of current oracle states. In total, there are five different oracle states, i.e., `chainlinkWorking`, `usingBandChainlinkUntrusted`, `bothOraclesUntrusted`, `usingBandChainlinkFrozen`, and `usingChainlinkBandUntrusted`. While examining possible transition from the fourth state, we notice the transition logic can be revisited.

To elaborate, we show below the code snippet from the `_fetchPrice()` function. This function is designed to fetch the current price and adjust the current oracle state accordingly. Starting from the fourth state `usingBandChainlinkFrozen`, the current logic considers the conditions of `!_chainlinkIsFrozen(chainlinkResponse)` (line 263) and `_bandIsBroken(bandResponse)` (line 282) to still yield `usingBandChainlinkFrozen` as the next state, which in fact can be better adjusted as `usingChainlinkBandFrozen`.

```
248        if (status == Status.usingBandChainlinkFrozen) {
249            if (_chainlinkIsBroken(chainlinkResponse, prevChainlinkResponse)) {
250                // If both Oracles are broken, return last good price
```

```
                if (_bandIsBroken(bandResponse)) {
                    return (Status.bothOraclesUntrusted, lastGoodPrice);
                }

                // If Chainlink is broken, remember it and switch to using Band

                if (_bandIsFrozen(bandResponse)) {return (Status.usingBandChainlinkUntrusted, lastGoodPrice);}

                // If Band is working, return Band current price
                return (Status.usingBandChainlinkUntrusted, bandResponse.value);
            }

            if (_chainlinkIsFrozen(chainlinkResponse)) {
                // if Chainlink is frozen and Band is broken, remember Band broke, and return last good price
                if (_bandIsBroken(bandResponse)) {
                    return (Status.usingChainlinkBandUntrusted, lastGoodPrice);
                }

                // If both are frozen, just use lastGoodPrice
                if (_bandIsFrozen(bandResponse)) {return (Status.usingBandChainlinkFrozen, lastGoodPrice);}

                // if Chainlink is frozen and Band is working, keep using Band (no status change)
                return (Status.usingBandChainlinkFrozen, bandResponse.value);
            }

            // if Chainlink is live and Band is broken, remember Band broke, and return Chainlink price
            if (_bandIsBroken(bandResponse)) {
                return (Status.usingChainlinkBandUntrusted, chainlinkResponse.answer);
            }

             // If Chainlink is live and Band is frozen, just use last good price (no status change) since we have no basis for comparison
            if (_bandIsFrozen(bandResponse)) {return (Status.usingBandChainlinkFrozen, lastGoodPrice);}

            // If Chainlink is live and Band is working, compare prices. Switch to Chainlink
            // if prices are within 5%, and return Chainlink price.
            if (_bothOraclesSimilarPrice(chainlinkResponse, bandResponse)) {
                return (Status.chainlinkWorking, chainlinkResponse.answer);
            }

            // Otherwise if Chainlink is live but price not within 5% of Band, distrust Chainlink, and return Band price
            return (Status.usingBandChainlinkUntrusted, bandResponse.value);
        }
```

Listing 3.3: PriceFeed::_fetchPrice()

Recommendation Apply the proper state-transition logic in `_fetchPrice()` as elaborated earlier.

Status The issue has been confirmed.

3.3 Inconsistent Use Of IncentivesController::handleAction()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StableDebtToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The `Geist` 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 `_mint()` function from both `IncentivizedERC20` and `StableDebtToken` contracts. It comes to our attention that the first contract uses the post-update balance in the invocation of `IncentivesController::handleAction()` (line 212) while the second contract uses the pre-update balance in the invocation of `IncentivesController::handleAction()` (line 413)!

```
200  function _mint(address account, uint256 amount) internal virtual {
201    require(account != address(0), 'ERC20: mint to the zero address');
202
203    _beforeTokenTransfer(address(0), account, amount);
204
205    uint256 currentTotalSupply = _totalSupply.add(amount);
206    _totalSupply = currentTotalSupply;
207
208    uint256 accountBalance = _balances[account].add(amount);
209    _balances[account] = accountBalance;
210
211    if (address(_getIncentivesController()) != address(0)) {
212      _getIncentivesController().handleAction(account, accountBalance,
         currentTotalSupply);
213    }
214  }
```

Listing 3.4: `IncentivizedERC20::_mint()`

```
398  /**
399   * @dev Mints stable debt tokens to an user
400   * @param account The account receiving the debt tokens
401   * @param amount The amount being minted
402   * @param oldTotalSupply the total supply before the minting event
403   **/
```

```
function _mint(
  address account,
  uint256 amount,
  uint256 oldTotalSupply
) internal {
  uint256 oldAccountBalance = _balances[account];
  _balances[account] = oldAccountBalance.add(amount);

  if (address(_incentivesController) != address(0)) {
    _incentivesController.handleAction(account, oldAccountBalance, oldTotalSupply);
  }
}
```

Listing 3.5: `StableDebtToken::_mint()`

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

Status The issue has been confirmed and the team clarifies the `stable lending` is disabled in `Geist`, which resolves the above inconsistency.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

3.4 ERC20 Compliance Of GeistToken

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: GeistToken
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

The GeistToken protocol is designed as the governance token, which will be disseminated to community and protocol users. In the following, we examine the ERC20 compliance of the GEIST token contract. Specifically, the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue. Specifically, the current implementation has defined the decimals state with the uint256 type. The ERC20 specification indicates the type of uint8 for the decimals state. Note that this incompatibility issue does not necessarily affect the functionality of GEIST in any negative way.

In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the GEIST implementation to ensure its ERC20-compliance.

Status The issue has been fixed by this commit: de05115.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.5 Fork-Resistant Domain Separator in AToken

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `AToken`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The various tokens in `Geist` are designed to strictly follows the widely-accepted ERC20 specification (Section 3.4). 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` in `AToken` is initialized once inside the `initialize()` function (lines 81-89).

```
64   function initialize(
65     ILendingPool pool,
66     address treasury,
67     address underlyingAsset,
68     IAaveIncentivesController incentivesController,
69     uint8 aTokenDecimals,
70     string calldata aTokenName,
71     string calldata aTokenSymbol,
72     bytes calldata params
73   ) external override initializer {
74     uint256 chainId;
75
76     //solium-disable-next-line
```

```
    assembly {
      chainId := chainid()
    }

    DOMAIN_SEPARATOR = keccak256(
      abi.encode(
        EIP712_DOMAIN,
        keccak256(bytes(aTokenName)),
        keccak256(EIP712_REVISION),
        chainId,
        address(this)
      )
    );

    _setName(aTokenName);
    _setSymbol(aTokenSymbol);
    _setDecimals(aTokenDecimals);

    _pool = pool;
    _treasury = treasury;
    _underlyingAsset = underlyingAsset;
    _incentivesController = incentivesController;

    emit Initialized(
      underlyingAsset,
      address(pool),
      treasury,
      address(incentivesController),
      aTokenDecimals,
      aTokenName,
      aTokenSymbol,
      params
    );
  }
```

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 This issue has been confirmed.

3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterChef`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `emergencyWithdraw()` 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 297) starts before effecting the update on internal states (lines 299−300), 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. The same issue is also applicable to other functions, including `deposit()` and `withdraw()`.

```
292     // Withdraw without caring about rewards. EMERGENCY ONLY.
293     function emergencyWithdraw(address _token) external {
294         PoolInfo storage pool = poolInfo[_token];
295         UserInfo storage user = userInfo[_token][msg.sender];
296         uint256 amount = user.amount;
297         IERC20(_token).safeTransfer(address(msg.sender), amount);
298         emit EmergencyWithdraw(_token, msg.sender, amount);
299         user.amount = 0;
300         user.rewardDebt = 0;
301         if (pool.onwardIncentives != IOnwardIncentivesController(0)) {
302             uint256 lpSupply = IERC20(_token).balanceOf(address(this));
303             try pool.onwardIncentives.handleAction(_token, msg.sender, 0, lpSupply) {}
                    catch {}
304         }
305     }
```

Listing 3.8: MasterChef::emergencyWithdraw()

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 in making use of `nonReentrant` to block possible `re-entrancy` and the adherence of `checks-effects-interactions` best practice is highly recommended.

Recommendation Apply necessary reentrancy prevention by following the known checks-effects-interactions pattern in addition to the utilization of the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed by this commit: `78b1414`.

3.7 Staking Incompatibility With Deflationary Tokens

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `MasterChef`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `Geist` protocol, the `MasterChef` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `_updatePool()` routine. This routine calculates `pool.accRewardPerShare` via dividing `reward` by `lpSupply`, where the `lpSupply` is derived from `IERC20(_token).balanceOf(address (this))` (line 212). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accRewardPerShare` as the final result, which dramatically inflates the pool's reward.

```
206     // Update reward variables of the given pool to be up-to-date.
207     function _updatePool(address _token, uint256 _totalAllocPoint) internal {
208         PoolInfo storage pool = poolInfo[_token];
209         if (block.timestamp <= pool.lastRewardTime) {
210             return;
211         }
212         uint256 lpSupply = IERC20(_token).balanceOf(address(this));
213         if (lpSupply == 0) {
214             pool.lastRewardTime = block.timestamp;
215             return;
216         }
```

```
        uint256 duration = block.timestamp.sub(pool.lastRewardTime);
        uint256 reward = duration.mul(rewardsPerSecond).mul(pool.allocPoint).div(
            _totalAllocPoint);
        pool.accRewardPerShare = pool.accRewardPerShare.add(reward.mul(1e12).div(
            lpSupply));
        pool.lastRewardTime = block.timestamp;
    }
```

Listing 3.9: `MasterChef::_updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` 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 `Geist` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed. The team clarifies no plan to support deflationary tokens here.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

In the `Geist` 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). 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 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 3.10: Example Setters in 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 3.11: The LendingPoolAddressesProvider Contract

It is worrisome if the privileged `owner` account is a plain EOA account. The discussion with the team confirms that the `owner` account is currently managed by a 2-day timelock, owned by a multisig. Note that a multi-sig account indeed greatly alleviates this concern, though it is still not 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 if current contracts are to be deployed behind a proxy, 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. For the time being, it is being mitigated by a 2-day timelock (owned by a multi-sig account) to balance efficiency and timely adjustment. After the protocol becomes stable, it is expected to eventually migrate to be owned by community proposals for decentralized governance.

4 | Conclusion

In this audit, we have analyzed the `Geist` design and implementation. The system presents a unique, robust offering as a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. The `Geist` protocol extends the original `AaveV2` 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 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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

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

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

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

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

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