Security Audit Report

AladdinETH (aETH) by AladdinDAO

SECBIT

December 22, 2022

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. The AladdinETH (aETH) protocol is dedicated to increasing the yield of users' assets and offers various yield strategies. SECBIT Labs conducted an audit from December 13 to December 22, 2022, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the AladdinETH (aETH) contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Adjust the code logic to allow users to withdraw the unlocked tokens at the time they are unlocked.	Info	Fixed
Design & Implementation	4.3.2 Discussion of the logic of `withdraw()` function.	Info	Discussed
Design & Implementation	4.3.3 Discussion of `execute()` function.	Info	Discussed
Design & Implementation	4.3.4 The administrator called the `migrateStrategy()` function to migrate funds without considering possible rewards.	Info	Discussed
Design & Implementation	4.3.5 There is no check on pool status in the `claimMulti()` and `claimAll()` functions.	Info	Discussed
Design & Implementation	4.3.6 Possible leftover rewards under the old contract should be handled when the administrator migrates the funds.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the AladdinETH (aETH) Protocol is shown below:

- Smart contract code
  - initial review commit *ed4eadc*
  - final review commit *a836c4f*

2.2 Contract List

The following content shows the contracts included in the AladdinETH (aETH) Protocol, which the SECBIT team audits:

Name	Lines	Description
FeeCustomization.sol	58	An abstract contract that deals with the fees.
AladdinETH.sol	21	Core contract about aETH strategy.
ConcentratorAladdinETHVault.sol	63	Manage contracts for manual compounding strategies.
AutoCompoundingConvexCurveStrategy.sol	53	Automatic compounding strategy of core contract for

		Convex protocol returns.
AutoCompoundingConvexFraxStrategy.sol	181	Automatic compounding strategy of core contract for Frax protocol returns.
AutoCompoundingStrategyBase.sol	34	Automatic compounding strategy of base class contracts.
ConcentratorStrategyBase.sol	41	Concentrator strategy base contract.
ManualCompounding- ConvexCurveStrategy.sol	50	Manual compounding strategy of core contract for Convex protocol returns.
ManualCompounding- CurveGaugeStrategy.sol	51	Manual compounding strategy of core contract for Curve protocol returns.
ManualCompounding- StrategyBase.sol	31	Manual compounding strategy of base class contracts.
AladdinCompounder.sol	239	Abstract contract that implements the core logic of curve LP token.
AladdinCompounder- WithStrategy.sol	113	Abstract contract that deals with automatic compounding strategy.

ConcentratorGeneralVault.sol	493	Abstract contract for depositing LP tokens into a general protocol.

Note: `AladdinCompounder.sol` *and* `ConcentratorGeneralVault.sol` *contracts have been modified from the original audited version, and only the changed codes are audited here.*

3. Contract Analysis

This part describes code assessment details, including "role classification" and "functional analysis".

3.1 Role Classification

Three key roles in the AladdinETH (aETH) Protocol are Governance Account, Common Account, and Operator Account.

- Governance Account
  - Description

    Contract Administrator
  - Authority
    - Update basic parameters
    - Migrate underlying tokens
    - Transfer ownership
    - Pause deposit for specific pool
  - Method of Authorization

    The contract administrator is the contract's creator or authorized by transferring the governance account.
- Common Account

    - Description

      Users participate in the AladdinETH (aETH) Protocol.

    - Authority
        - Deposit the underlying token(curve LP token) to receive a share token in the `AladdinETH.sol` contract
        - Harvest pending rewards and reinvest in the pool
        - Deposit strategy token to get rewards in the `ConcentratorAladdinETHVault.sol` contract
    - Method of Authorization

      No authorization required

- Operator Account
    - Description

      The actual caller of strategies

    - Authority
        - Call the core functions in each strategy, such as `deposit()`, `withdraw()`, and `harvest()`.
    - Method of Authorization

      The contract deployer determines the operator address when the contract is initialized.

3.2 Functional Analysis

The AladdinETH (aETH) protocol offers different strategies for users to reinvest their Ethers. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into six parts:

AladdinETH, AladdinCompounderWithStrategy, and AladdinCompounder

These three contracts cover the AladdinETH (aETH) protocol's core functionality. The main functions in these contracts are as below:

- `harvest()`

  Anyone could call this function to harvest rewards and notify it.

- `deposit()`

  Users can deposit underlying tokens to the contract, which will be transferred to the strategy contract. At the same time, the user can receive share tokens.

- `mint()`

  The code will calculate the number of underlying tokens the user should deposit based on the number of share tokens the user wants to obtain. It will then transfer these underlying tokens to the Convex protocol. At the same time, the user can receive share tokens.

- `withdraw()`

  The user withdraws a specified number of underlying tokens and burns the corresponding share tokens.

- `redeem()`

  The user specifies the number of share tokens to be burned and gets back the corresponding underlying tokens.

ConcentratorAladdinETHVault and ConcentratorGeneralVault

Users can deposit strategy tokens directly into the strategy contract and receive rewards. Different from the `AladdinETH` contract, the user of this contract can choose the type of rewards he wants to receive. The main functions in these contracts are as below:

- `deposit()`

The user deposits funds into the specified strategy, which will be recorded under the `_recipient` address.

- `withdraw()`

  The user withdraws funds from the specified strategy to the specified `_recipient` address.

- `claim()`

  This function allows the user to withdraw the earnings generated under the specified strategy.

- `claimAll()`

  This function allows the user to withdraw earnings already generated under all strategies.

- `harvest()`

  Users can call this function to retrieve revenue from the strategy contract and distribute it to all users.

AutoCompoundingConvexCurveStrategy, ConcentratorStrategyBase, and ConcentratorStrategyBase

These three contracts constitute the complete revenue strategy. The revenue earned under this strategy is automatically zapped to staking tokens and deposited in the Convex protocol. The main functions in these contracts are as below:

- `deposit()`

  Users cannot call this function directly. The `Operator` role will deposit underlying tokens under this contract into the Convex protocol.

- `withdraw()`

  Users cannot call this function directly. The `Operator` role withdraws underlying tokens from the Convex protocol and transfers them to the `_recipient` address.

- `harvest()`

Users cannot call this function directly. The `Operator` role harvests rewards, converts them into staking tokens, and deposits them into the Convex protocol.

AutoCompoundingConvexFraxStrategy, AutoCompoundingStrategyBase, and ConcentratorStrategyBase

These three contracts constitute the complete revenue strategy. The revenue earned under this strategy is automatically zapped to staking tokens. The main functions in these contracts are as below:

- `deposit()`

  Users cannot call this function directly. The `Operator` role will deposit underlying tokens under this contract into the Convex protocol, and these tokens will be locked scheduled period.

- `withdraw()`

  Users cannot call this function directly. The `Operator` role will record the amount of underlying token the user wants to withdraw.

- `claim()`

  Users can call this function to claim the unlocked tokens after the unlocking time has been reached.

- `harvest()`

  Users cannot call this function directly. The `Operator` role harvests rewards and converts them into staking tokens. These staking tokens will be locked into the Convex protocol if the contract is not paused.

ManualCompoundingConvexCurveStrategy, ConcentratorStrategyBase, and ManualCompoundingStrategyBase

These three contracts constitute the complete revenue strategy. The revenue earned under this strategy is automatically zapped to underlying tokens. The main functions in these contracts are as below:

- `deposit()`

Users cannot call this function directly. The `Operator` role will deposit underlying tokens under this contract into the Convex protocol.

- `withdraw()`

  Users cannot call this function directly. The `Operator` role withdraws underlying tokens from the Convex protocol and transfers them to the `_recipient` address.

- `harvest()`

  Users cannot call this function directly. The `Operator` role harvests rewards, converts them into staking tokens, and claims these rewards.

ManualCompoundingCurveGaugeStrategy, ConcentratorStrategyBase, and ManualCompoundingStrategyBase

These three contracts constitute the complete revenue strategy. The revenue earned under this strategy is automatically zapped to underlying tokens. The main functions in these contracts are as below:

- `deposit()`

  Users cannot call this function directly. The `Operator` role will deposit underlying tokens under this contract into the Convex protocol.

- `withdraw()`

  Users cannot call this function directly. The `Operator` role withdraws underlying tokens from the Convex protocol and transfers them to the `_recipient` address.

- `harvest()`

  Users cannot call this function directly. The `Operator` role harvests rewards, converts them into staking tokens, and claims these rewards.

4. Audit Detail

This part describes the process, and the detailed audit results also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bugs, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools, including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓

5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in the design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Adjust the code logic to allow users to withdraw the unlocked tokens at the time they are unlocked.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Description

The internal function `withdrawLockedAndUnwrap()` withdraws the tokens locked under the `_vault` contract. See the link https://etherscan.io/address/0x60391c9a3ebaace2a40b880ad03e95170675d7e4/advanced#code#L985. It shows that the token could be withdrawn when the user calls the function at a time exactly equal to the expected unlocked time.

```
//@audit reference:
https://etherscan.io/address/0x60391c9a3ebaace2a40b880ad03e951
70675d7e4/advanced#code#L985
function withdrawLockedAndUnwrap(bytes32 _kek_id) external
onlyOwner nonReentrant{
        //withdraw
        IFraxFarmERC20(stakingAddress).withdrawLocked(_kek_id,
address(this));

        //unwrap
IConvexWrapperV2(stakingToken).withdrawAndUnwrap(IERC20(stakin
gToken).balanceOf(address(this)));
        ......
    }

//@audit reference:
https://etherscan.io/address/0xa537d64881b84faffb9Ae43c951EEbF
368b71cdA#code#L2103
function withdrawLocked(bytes32 kek_id, address
destination_address) nonReentrant external returns (uint256) {
```

```
        require(withdrawalsPaused == false, "Withdrawals
paused");
        return _withdrawLocked(msg.sender,
destination_address, kek_id);
    }

    // No withdrawer == msg.sender check needed since this is
only internally callable and the checks are done in the
wrapper
    function _withdrawLocked(
        address staker_address,
        address destination_address,
        bytes32 kek_id
    ) internal returns (uint256) {
        // Collect rewards first and then update the balances
        _getReward(staker_address, destination_address, true);

        // Get the stake and its index
        (LockedStake memory thisStake, uint256 theArrayIndex)
= _getStake(staker_address, kek_id);

        require(block.timestamp >= thisStake.ending_timestamp
|| stakesUnlocked == true, "Stake is still locked!");
        uint256 liquidity = thisStake.liquidity;

       ......
    }
```

If the user withdraws tokens at a time equal to the unlock time, it will fail under the current code logic.

```
function claim(address _account) public {
    {
      LockData memory _locks = locks;
      // try to trigger the unlock and extend.
      _extend(vault, _locks);
      locks = _locks;
    }
```

```
    UserLockedBalance[] storage _userLocks =
userLocks[_account];
    uint256 _length = _userLocks.length;
    uint256 _nextIndex = nextIndex[_account];
    uint256 _unlocked;
    while (_nextIndex < _length) {
      UserLockedBalance memory _lock = _userLocks[_nextIndex];

      //@audit cannot withdraw tokens when block.timestamp ==
unlockAt
      if (_lock.unlockAt < block.timestamp) {
        _unlocked += _lock.balance;
        delete _userLocks[_nextIndex];
      } else {
        break;
      }
      _nextIndex += 1;
    }
    nextIndex[_account] = _nextIndex;

    IERC20(token).safeTransfer(_account, _unlocked);
  }

function _extend(address _vault, LockData memory _locks)
internal {

    if (_locks.unlockAt >= block.timestamp) return;

    if (_locks.pendingToUnlock > 0) {
      // unlock pending tokens
      _unlock(_vault, _locks, _locks.pendingToUnlock);
      _locks.pendingToUnlock = 0;
    } else if (!paused() && _locks.key != bytes32(0)) {
     ......
  }

  function _unlock(
    address _vault,
    LockData memory _locks,
```

```
    uint256 _amount
  ) internal {
    // all are unlocked.
    if (_locks.key == bytes32(0)) {
      // unlock token when paused
      pendingToLock -= _amount;
      return;
    }

    address _token = token;
    uint256 _unlocked =
IERC20(_token).balanceOf(address(this));

    //@audit withdraw unlocked funds

 IStakingProxyConvex(_vault).withdrawLockedAndUnwrap(_locks.ke
y);
    _unlocked = IERC20(_token).balanceOf(address(this)) -
_unlocked;
    .......
  }
```

Suggestion

Based on the `withdrawLockedAndUnwrap()` logic, adjust the current code judgment condition to allow unlocked funds to be retrieved when `_locks.unlockAt == block.timestamp`. The corresponding modification would be as follows.

```
function claim(address _account) public {
    {
      LockData memory _locks = locks;
      // try to trigger the unlock and extend.
      _extend(vault, _locks);
      locks = _locks;
    }

    UserLockedBalance[] storage _userLocks =
userLocks[_account];
```

```
    uint256 _length = _userLocks.length;
    uint256 _nextIndex = nextIndex[_account];
    uint256 _unlocked;
    while (_nextIndex < _length) {
      UserLockedBalance memory _lock = _userLocks[_nextIndex];

      //@audit modify as follows
      if (_lock.unlockAt <= block.timestamp) {
        _unlocked += _lock.balance;
        delete _userLocks[_nextIndex];
      } else {
        break;
      }
      _nextIndex += 1;
    }
    nextIndex[_account] = _nextIndex;

    IERC20(token).safeTransfer(_account, _unlocked);
  }

function _extend(address _vault, LockData memory _locks)
internal {

    //@audit modify as follows
    if (_locks.unlockAt > block.timestamp) return;

    if (_locks.pendingToUnlock > 0) {
      // unlock pending tokens
      _unlock(_vault, _locks, _locks.pendingToUnlock);
      _locks.pendingToUnlock = 0;
    } else if (!paused() && _locks.key != bytes32(0)) {
     ......
  }
```

Status

The team has adopted the suggestion and fixed this issue in commit *a836c4f*.

4.3.2 Discussion of the logic of `withdraw()` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

When a user wants to withdraw tokens, they need to call the `withdraw()` function, which will accumulate the funds they want to unlock during the current lock period. When the unlock time has arrived, the user calls relevant functions such as the `claim()` function, all the tokens in the period will be taken back, and the lock period and unlock time will be updated.

Consider the following scenario: if no one calls the relevant function after the unlock time has arrived, this will not update the unlock time. At this time, another user calls the `withdraw()` function to withdraw the tokens. Under the assumptions, the user can withdraw tokens immediately and then update the unlock time. At this point, the user does not need any wait time.

The locking period and unlock time should be updated first.

```
function withdraw(address _recipient, uint256 _amount)
external override onlyOperator {
    if (_amount > 0) {
      LockData memory _locks = locks;

      // add lock record
      userLocks[_recipient].push(
        UserLockedBalance({ balance: uint128(_amount),
unlockAt: _locks.unlockAt, _unused: 0 })
      );
```

```
        // increase the pending unlocks.
        _locks.pendingToUnlock =
uint128(uint256(_locks.pendingToUnlock) + _amount);

        // try extend lock duration
        _extend(vault, _locks);

        // update storage
        locks = _locks;
      }
    }

function _extend(address _vault, LockData memory _locks)
internal {
      // no need to extend now
      if (_locks.unlockAt >= block.timestamp) return;

      if (_locks.pendingToUnlock > 0) {
        // unlock pending tokens
        _unlock(_vault, _locks, _locks.pendingToUnlock);
        _locks.pendingToUnlock = 0;
      } else if (!paused() && _locks.key != bytes32(0)) {
        // Don't extend lock duration when paused or no lock
exists
        // _locks.key = bytes32(0) will happen when
        // 1. setPause(true)
        // 2. withdraw
        // 3. setPause(false)
        // 4. claim
        IStakingProxyConvex(_vault).lockLonger(_locks.key,
block.timestamp + _locks.duration);
        _locks.unlockAt = uint64(block.timestamp +
_locks.duration);
      }
    }
```

Status

This issue has been discussed. The team decides to keep the logic for the purpose of migrating tokens for convenience.

4.3.3 Discussion of `execute()` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

This function is designed to prevent unexpected emergencies from threatening the security of user funds. Only the `Operator` can call this function in the current code. Further analysis shows that the current `Operator` is either the `AladdinETH` contract address or the `ConcentratorAladdinETHvault` contract address, but neither of these addresses provides a function to call the `execute()` function. Therefore, it is important to check whether this function is retained for other reasons.

```
function execute(
    address _to,
    uint256 _value,
    bytes calldata _data
  ) external payable override onlyOperator returns (bool,
bytes memory) {
    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory result) = _to.call{ value:
_value }(_data);
    return (success, result);
  }
```

Status

This issue has been discussed. The team clarified that the function was retained to deal with unexpected situations.

4.3.4 The administrator called the `migrateStrategy()` function to migrate funds without considering possible rewards.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

The `migrateStrategy()` function is used to migrate funds from this contract to a new `_newStrategy` contract. It does not consider the possible rewards, and different strategies must be handled differently here.

- The `AutoCompoundingConvexCurveStrategy` strategy requires the administrator to call the `harvest()` function under this contract before retrieving any potentially unprocessed rewards.
- The `AutoCompoundingConvexFraxStrategy` strategy requires the administrator to call the `setPaused()` function to suspend the contract and then call the `harvest()` function to retrieve the rewards.

```
function migrateStrategy(address _newStrategy) external
onlyOwner {
    require(_newStrategy != address(0), "AladdinCompounder:
zero new strategy address");

    _distributePendingReward();

    uint256 _totalUnderlying = totalAssetsStored;
    RewardInfo memory _info = rewardInfo;
    if (_info.periodLength > 0) {
      if (block.timestamp < _info.finishAt) {
```

```
            _totalUnderlying += (_info.finishAt - block.timestamp)
* _info.rate;
        }
      }

      address _oldStrategy = strategy;
      strategy = _newStrategy;


   IConcentratorStrategy(_oldStrategy).prepareMigrate(_newStrate
gy);
      IConcentratorStrategy(_oldStrategy).withdraw(_newStrategy,
_totalUnderlying);

   IConcentratorStrategy(_oldStrategy).finishMigrate(_newStrateg
y);

      IConcentratorStrategy(_newStrategy).deposit(address(this),
_totalUnderlying);

      emit Migrate(_oldStrategy, _newStrategy);
    }
```

Status

This issue has been discussed. The team confirmed that special preparations would be made before the migration of funds.

4.3.5 There is no check on pool status in the `claimMulti()` and `claimAll()` functions.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

This contract code is a modification of the original code. In the original code, when the user calls the `claimAll()` function to claim a reward, it determines whether the corresponding reward pool has been deactivated and if so, the pool rewards will be skipped. The `claimMulti()` function and the `claimAll()` function in the current code do not determine the pool's state, so this design needs to be confirmed.

```
function claimMulti(
    uint256[] memory _pids,
    address _recipient,
    uint256 _minOut,
    address _claimAsToken
  ) public override nonReentrant returns (uint256) {
    uint256 _poolIndex = poolIndex;
    uint256 _rewards;
    for (uint256 i = 0; i < _pids.length; i++) {
      uint256 _pid = _pids[i];
      require(_pid < _poolIndex, "Concentrator: pool not
exist");

      UserInfo storage _userInfo = userInfo[_pid][msg.sender];
      // update if user has share
      if (_userInfo.shares > 0) {
        _updateRewards(_pid, msg.sender);
      }
      // withdraw if user has reward
      if (_userInfo.rewards > 0) {
        _rewards = _rewards.add(_userInfo.rewards);
        emit Claim(_pid, msg.sender, _recipient,
_userInfo.rewards);

        _userInfo.rewards = 0;
      }
    }

    return _claim(_rewards, _minOut, _recipient,
_claimAsToken);
```

```
    }

    /// @inheritdoc IConcentratorGeneralVault
    function claimAll(
      uint256 _minOut,
      address _recipient,
      address _claimAsToken
    ) external override nonReentrant returns (uint256) {
      uint256 _length = poolIndex;
      uint256 _rewards;
      for (uint256 _pid = 0; _pid < _length; _pid++) {
        UserInfo storage _userInfo = userInfo[_pid][msg.sender];
        // update if user has share
        if (_userInfo.shares > 0) {
          _updateRewards(_pid, msg.sender);
        }
        // withdraw if user has reward
        if (_userInfo.rewards > 0) {
          _rewards = _rewards.add(_userInfo.rewards);
          emit Claim(_pid, msg.sender, _recipient,
_userInfo.rewards);

          _userInfo.rewards = 0;
        }
      }

      return _claim(_rewards, _minOut, _recipient,
_claimAsToken);
    }
```

Status

The team confirmed this logic and decided to maintain the current design.

4.3.6 Possible leftover rewards under the old contract should be handled when the administrator migrates the funds.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Description

The administrator does not consider any possible leftover rewards when calling the `migrateStrategy()` function to migrate the funds under the `_oldStrategy` contract to the new `_newStrategy`. It is necessary to check whether the rewards need to be processed in parallel with the migration of the funds.

```
/// @notice Migrate pool assets to new strategy.
/// @param _pid The pool id.
/// @param _newStrategy The address of new strategy.
function migrateStrategy(uint256 _pid, address _newStrategy)
external onlyExistPool(_pid) onlyOwner {
    uint256 _totalUnderlying =
poolInfo[_pid].supply.totalUnderlying;
    address _oldStrategy = poolInfo[_pid].strategy.strategy;
    poolInfo[_pid].strategy.strategy = _newStrategy;


 IConcentratorStrategy(_oldStrategy).prepareMigrate(_newStrate
gy);
    IConcentratorStrategy(_oldStrategy).withdraw(_newStrategy,
_totalUnderlying);

 IConcentratorStrategy(_oldStrategy).finishMigrate(_newStrateg
y);
    IConcentratorStrategy(_newStrategy).deposit(address(this),
_totalUnderlying);

    emit Migrate(_pid, _oldStrategy, _newStrategy);
```

```
    }
```

Status

This issue has been discussed. The team confirmed that special preparations would be made before the migration of funds.

5. Conclusion

After auditing and analyzing the AladdinETH (aETH) Protocol contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock assets inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io