Cyberscope

Audit Report

AIA Power

December 2024

Repository https://github.com/aiachain/aia-power

Commit 84b75ea86cb7c3b642df1207b6900d0f9a5665be

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/aiachain/aia-power
Commit	84b75ea86cb7c3b642df1207b6900d0f9a5665be
Testing Deploy	https://testnet.bscscan.com/address/0xfafed47824dcf5588761c9e74f8f724d1f478558

Audit Updates

Initial Audit	16 Dec 2024

Source Files

Filename	SHA256
contracts/Pool.sol	356bdd3cec54d2a366328296d8a0cbd7b34784997adcc8323df00035ddc75e3f
@openzeppelin/contracts/utils/Strings.sol	cb2df477077a5963ab50a52768cb74ec6f32177177a78611ddbbe2c07e2d36de
@openzeppelin/contracts/utils/Context.sol	b2cfee351bcafd0f8f27c72d76c054df9b571b62cfac4781ed12c86354e2a56c
@openzeppelin/contracts/utils/math/SignedMath.sol	420a5a5d8d94611a04b39d6cf5f02492552ed4257ea82aba3c765b1ad52f77f6
@openzeppelin/contracts/utils/math/Math.sol	85a2caf3bd06579fb55236398c1321e15fd524a8fe140dff748c0f73d7a52345
@openzeppelin/contracts/utils/introspection/IERC165.sol	701e025d13ec6be09ae892eb029cd83b3064325801d73654847a5fb11c58b1e5
@openzeppelin/contracts/utils/cryptography/ECDSA.sol	445963619903cee339e49aa2d7a0b07cfad90959529fff136394429c4a92d554

@openzeppelin/contracts/token/ERC721/IERC721.sol	c8d867eda0fd764890040a3644f5ccf5db92f852779879f321ab3ad8b799bf97
@openzeppelin/contracts/token/ERC20/IERC20.sol	7ebde70853ccafcf1876900dad458f46eb9444d591d39bfc58e952e2582f5587
@openzeppelin/contracts/token/ERC20/ERC20.sol	d20d52b4be98738b8aa52b5bb0f88943f62128969b33d654fbca731539a7fe0a
@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol	af5c8a77965cc82c33b7ff844deb9826166689e55dc037a7f2f790d057811990

Overview

The contract defines a complex system encompassing ownership, administrative roles, and staking mechanisms. It leverages inheritance and modular design to implement functionalities for role management and asset staking while maintaining flexibility for upgrades.

Ownership and Administrative Structure

The contract introduces an ownable structure through AdminOwnerUpgradable, enabling the assignment of an owner and additional administrators with permissions to perform specific functions. Owners can transfer ownership or renounce it entirely, while administrators can be added or removed with historical records maintained for audit purposes. This system ensures robust access control and traceability of administrative actions.

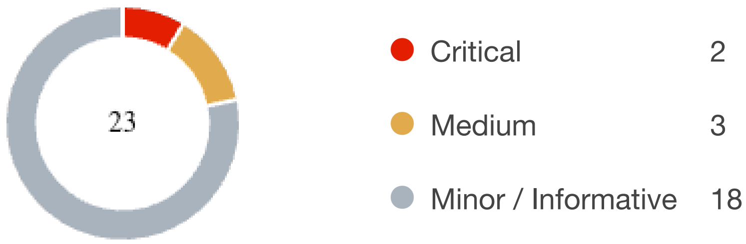
Core Functionalities

1. Staking System:
    - Users can stake tokens (or native currency) and optionally provide NFTs as part of the staking process. Staked NFTs are transferred to a designated fee address, and users' staking records are stored for transparency.
    - Stake validation considers several criteria, including minimum stake amounts and optional NFT requirements, with pool settings configurable by the owner.
2. Upgradeability:
    - Upgradeability is achieved using the Proxiable pattern, allowing the owner to update the implementation logic. This ensures the system can evolve without disrupting existing state or user data.
3. Price Oracle Integration:
    - The contract integrates a MultiPriceOracle for token price data, enabling dynamic pricing and asset valuation.
4. Event Logging:
    - Comprehensive event logging is in place to provide transparency for ownership transfers, administrative modifications, and staking or reward operations.

Contract Readability Comment

The assessment of the smart contract has revealed a deeply concerning issue – the codebase is overly complicated, tangled, and deviates significantly from fundamental coding principles. The complexity has reached a level where the code becomes almost unreadable and unintelligible. Even if the identified findings are addressed and rectified, the contract would still remain far from being production-ready due to its convoluted and non-standard structure. This inherent complexity not only hampers the contract's security but also presents a considerable maintenance challenge. To ensure the contract's stability, security, and long-term viability, it is essential to conduct a comprehensive code refactor. Simplifying and restructuring the code to adhere to best practices and coding standards will be imperative for making the contract production-ready and maintainable.

Findings Breakdown



Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	2	0	0
Medium	0	3	0	0
Minor / Informative	0	18	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
● Critical	TSI	Tokens Sufficiency Insurance	Acknowledged
● Critical	UWV	Unlimited Withdrawals Vulnerability	Acknowledged
● Medium	DPI	Decimals Precision Inconsistency	Acknowledged
● Medium	MSVP	Missing Signature Verification Property	Acknowledged
● Medium	SW	Stops Withdrawals	Acknowledged
● Minor / Informative	AAO	Accumulated Amount Overflow	Acknowledged
● Minor / Informative	CR	Code Repetition	Acknowledged
● Minor / Informative	CCR	Contract Centralization Risk	Acknowledged
● Minor / Informative	MMN	Misleading Method Naming	Acknowledged
● Minor / Informative	MEE	Missing Events Emission	Acknowledged
● Minor / Informative	PBV	Percentage Boundaries Validation	Acknowledged
● Minor / Informative	POSD	Potential Oracle Stale Data	Acknowledged
● Minor / Informative	RSM	Redundant State Modification	Acknowledged
● Minor / Informative	RVD	Redundant Variable Declaration	Acknowledged

●	TSD	Total Stakes Diversion	Acknowledged
●	L02	State Variables could be Declared Constant	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L08	Tautology or Contradiction	Acknowledged
●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L16	Validate Variable Setters	Acknowledged
●	L17	Usage of Solidity Assembly	Acknowledged
●	L19	Stable Compiler Version	Acknowledged
●	L20	Succeeded Transfer Check	Acknowledged

TSI - Tokens Sufficiency Insurance

Criticality	Critical
Location	contracts/Pool.sol#L415,431
Status	Acknowledged

Description

The tokens are held within the contract. However, the contract owner has the authority to withdraw the users' tokens. The owner may take advantage of it by calling the either the `flashReceive` or `destroy` function. While these function can provide flexibility in case an address sends tokens to the contract by mistake, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
function flashReceive(address _account, uint256 _realAmount, uint256
_feeAmount) external onlyAdmin {
    if (tokenAddr == address(0)) {
        if (_feeAmount > 0) {
            payable(feeAddr).transfer(_feeAmount);
        }
        payable(_account).transfer(_realAmount);
    } else {
        if (_feeAmount > 0) {
            IERC20(tokenAddr).transfer(feeAddr, _feeAmount);
        }
        IERC20(tokenAddr).transfer(_account, _realAmount);
    }

    totalReceives += _realAmount;
}

function destroy(uint256 _amount) external onlyAdmin {
    if (tokenAddr == address(0)) {
        payable(feeAddr).transfer(_amount);
    } else {
        IERC20(tokenAddr).transfer(feeAddr, _amount);
    }
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

Team Update

The team replied with the following statement:

There is no need to worry about this. We will ensure that there are sufficient tokens in the mining pool contract, and only the owner has the permission to withdraw coins, and only the admin has the permission to call relevant interfaces. The AIA coins in this pool mainly consist of three parts:

1. *AIA coins pledged by users*
2. *The AIA coin reward we deposited*
3. *AIA coins pledged through flash exchange*

UWV - Unlimited Withdrawals Vulnerability

Criticality	Critical
Location	contracts/Pool.sol#L439
Status	Acknowledged

Description

The `receiveA` function has a critical vulnerability that allows malicious users to repeatedly withdraw tokens without reducing their recorded stake in `userMapping` . Since the withdrawn amount is not deducted from the totalStake field of the user, a malicious user can call the function multiple times to withdraw funds far exceeding their actual stake, potentially draining the contract's balance and leading to significant losses.

```
function receiveA(uint256 _amount, Signature memory _signature) external {
    ...
}
```

Recommendation

The team is strongly advised to update the `userMapping[msg.sender].totalStake` value by subtracting the amount after the withdrawal is processed to ensure the user's stake reflects the updated balance. Additionally, the function should restrict users from withdrawing an amount that is greater than their staked amount.

The team could also introduce reentrancy protection using a mutex (`nonReentrant` modifier from OpenZeppelin's ReentrancyGuard library) to ensure the function cannot be recursively called within the same transaction.

Team Update

The team replied with the following statement:

This is because our logic is relatively complex, and data processing is centralized. Users may receive more AIA coins than they have pledged, so the contract does not have any reference data to make relevant restrictions. Therefore, users can receive AIA coins from the contract as long as they pass the backend signature.

DPI - Decimals Precision Inconsistency

Criticality	Medium
Location	contracts/Pool.sol#L402,440
Status	Acknowledged

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
userMapping[msg.sender].totalStake += amount;
require(userMapping[msg.sender].totalStake >= 1319*10**18, "invlid
address");
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

Team Update

The team replied with the following statement:

There is no need to worry about this, as our project's staking is only for AIA coins, with a fixed precision of 18 digits, and the restriction on receiving has been removed under the product's recommendation. Users who have not participated in regular staking may also receive AIA coin rewards if they participate in flash staking.

MSVP - Missing Signature Verification Property

Criticality	Medium
Location	contracts/Pool.sol#L476
Status	Acknowledged

Description

The `_verifySignature` function lacks the inclusion of the `chainId` parameter in the hash computation used for signature verification. Without incorporating the chainId, the same signature could potentially be replayed across different blockchain networks, posing a security risk. This omission undermines the uniqueness and safety of the signature verification process, making the contract vulnerable to cross-chain replay attacks.

```
function _verifySignature(address _account, uint256 _amount, uint256
_signId, uint256 _nonce, uint256 _deadline, bytes memory _data) private
view returns(uint8 _ret) {
    uint256 nonce = nonceMapping[_account];
    if (nonce != _nonce) {
        return 1;
    } else if (block.timestamp > _deadline) {
        return 2;
    }

    bytes32 hash = keccak256(abi.encodePacked(_signId, _account, _amount,
_nonce, _deadline));
    bytes32 message = ECDSA.toEthSignedMessageHash(hash);

    address _signer = ECDSA.recover(message, _data);

    if (_signer == address(0) || _signer != signAddr) {
        return 3;
    }

    return 0;
}
```

Recommendation

The team is advised to integrate the `chainId` into the hash calculation to ensure signatures are valid only on the intended blockchain network. Additionally, the off-chain signing mechanism should align with the updated hash structure by including the chainId. Tests should also be conducted to confirm that signatures valid on one network are rejected on another. By adding chainId to the signature verification process, the contract becomes resistant to cross-chain replay attacks, enhancing overall security.

Team Update

The team replied with the following statement:

Don't worry about this, because our project is only deployed on the AIA chain, and each signature ID is different, as well as the signature time limit. The backend will also match the signature ID after scanning it.

SW - Stops Withdrawals

Criticality	Medium
Location	contracts/Pool.sol#L440
Status	Acknowledged

Description

The contract enforces a restriction that prevents users from withdrawing their stakes unless their total stake amount exceeds `1319*10**18` tokens. This limitation can cause severe usability issues, as users who do not have sufficient funds to stake more than this amount will never meet the requirement, effectively locking their staked tokens. Furthermore, the contract assumes the staked token operates with 18 decimals, which might not align with the actual decimal places of the token (see DPI section). If the token has a different decimal configuration, this restriction becomes even more problematic, further preventing users from withdrawing their funds.

```
require(userMapping[msg.sender].totalStake >= 1319*10**18, "invlid
address");
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so that all users can withdraw their funds.

Team Update

The team replied with the following statement:

Don't worry, the total staking here is cumulative and will not decrease because the minimum staking amount previously restricted was this value. Therefore, as long as you have staked before, it will definitely be greater than or equal to this value. However, this code has also been commented out because users who have not staked regularly may also receive rewards if they participate in flash staking.

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	contracts/Pool.sol#L403,412
Status	Acknowledged

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

```
totalStakes += _amount;
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

Team Update

The team replied with the following statement:

Don't worry, the uint256 type is already large enough.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/Pool.sol#L277,306,416,451
Status	Acknowledged

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
require(_start >= 0 && _start < _end, "index error");

uint256 len;
uint256 realLen = stakeRecords[_account].length;
if (_start > realLen) {
    _start = realLen;
}

if (_end > realLen) {
    len = realLen - _start;
} else {
    len = _end - _start;
}

uint256 index;
RecordInfo[] memory tmpRecords = new RecordInfo[](len);
for (uint8 i = 0; i < len; i++) {
    index = realLen - (_start + i);
    if (index > 0) {
        index -= 1;
    }

    tmpRecords[i] = stakeRecords[_account][index];
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

Team Update

The team replied with the following statement:

There's no need to worry about this, it's just a reading record, it doesn't involve writing.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/Pool.sol#L245,252
Status	Acknowledged

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setAddrs(address _oracleAddr, address _signAddr, address
_feeAddr, address _tokenAddr) external onlyOwner {
    oracle = MultiPriceOracle(_oracleAddr);
    signAddr = _signAddr;
    feeAddr = _feeAddr;
    tokenAddr = _tokenAddr;
}
function setBase(uint256 _minStake, uint256 _feeRate, bool _poolState,
bool _needNft) external onlyOwner {
    minStake = _minStake;
    feeRate = _feeRate;
    poolState = _poolState;
    needNft = _needNft;
}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

Team Update

The team replied with the following statement:

Don't worry about this, only the owner has permission.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	contracts/Pool.sol#L415,431
Status	Acknowledged

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

The `flashReceive` method is an extended implementation of the `destroy` method. However, both methods transfer a given amount from the contract to a given address. As a result, the method names may cause confusion.

```
function flashReceive(address _account, uint256 _realAmount, uint256
_feeAmount) external onlyAdmin {
    ...
}
function destroy(uint256 _amount) external onlyAdmin {
    ...
}
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

Team Update

The team replied with the following statement:

Don't worry about this, it's determined based on business needs, one is flash redemption and the other is daily destruction.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/Pool.sol#L246,247,248,249,253,254,255,256
Status	Acknowledged

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
oracle = MultiPriceOracle(_oracleAddr);
signAddr = _signAddr;
feeAddr = _feeAddr;
tokenAddr = _tokenAddr;
minStake = _minStake;
feeRate = _feeRate;
poolState = _poolState;
needNft = _needNft;
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

Team Update

The team replied with the following statement:

Don't worry about this, only the owner has permission.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	contracts/Pool.sol#L254,448
Status	Acknowledged

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
uint256 feeAmount = _amount * feeRate / 10000;
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

Team Update

The team replied with the following statement:

Don't worry, the uint256 type is large enough not to exceed its boundaries.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	contracts/Pool.sol#L272
Status	Acknowledged

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
function getPrice(address _tokenAddr) public view returns (uint256 _price)
{
    uint256 price = oracle.assetPrices(_tokenAddr);
    return price;
}
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

Team Update

The team replied with the following statement:

Don't worry about this, we will ensure that the price in the oracle contract is normal. This mining pool contract only takes the price from the oracle and cannot determine whether the price is past or not.

RSM - Redundant State Modification

Criticality	Minor / Informative
Location	contracts/Pool.sol#L62
Status	Acknowledged

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract modifies the state of certain variables even when their current state matches the state passed as an argument. As a result, the state modification is redundant.

```
function modificationAdmin(address admin, bool state) public virtual
onlyOwner {
    emit ModificationAdmin(admin,  _admins[admin], state);
    _admins[admin] = state;

    if (adminInfo[admin].time == 0) {
        AdminInfo memory info = AdminInfo(msg.sender, admin,
block.timestamp);
        adminInfos.push(info);

        adminInfo[admin] = info;
    }
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

Team Update

The team replied with the following statement:

Don't worry about this, we just want to keep track of which admin is available for easy query and management.

RVD - Redundant Variable Declaration

Criticality	Minor / Informative
Location	contracts/Pool.sol#L102,118
Status	Acknowledged

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

Certain contracts are declared that are not used in a meaningful way by the main contract. As a result, these contracts are redundant.

```
abstract contract AdminOwner is OwnableEx {
    ...
}
contract Proxy {
    ...
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

Team Update

The team replied with the following statement:

Don't worry, this is the universal code we use for permission management and upgrade management.

TSD - Total Stakes Diversion

Criticality	Minor / Informative
Location	contracts/Pool.sol#L412
Status	Acknowledged

Description

The `totalStakes` is the total number of tokens that have been staked in the contract.The `totalStakes` and the stakes of individual accounts are two separate concepts that are managed by different variables in a smart contract. These two entities should be equal to each other. As a result, the sum of user stakes is diverse from the `totalStakes`.

```
totalStakes += _amount;
```

Recommendation

The `totalStakes` and the users' stake variables are separate and independent from each other. The `totalStakes` represents the total number of tokens that have been staked, while the stakes mapping stores the number of tokens that each account has staked. The sum of stakes should always equal the `totalStakes`.

Team Update

The team replied with the following statement:

This is not necessarily the case, as I mentioned in my first point, the AIA coin in the contract is composed of three parts and not necessarily all of them are pledged.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/Pool.sol#L123,134,155,159,164
Status	Acknowledged

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

Team Update

The team replied with the following statement:

This is not necessarily the case, basically everything can change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Pool.sol#L85,90,237,245,252,259,272,277,306,335,371,408,415,431,439,472
Status	Acknowledged

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _addr
uint256 _index
uint256 _amount
address _signAddr
address _oracleAddr
address _feeAddr
address _tokenAddr
uint256 _minStake
bool _needNft
uint256 _feeRate
bool _poolState
uint256[] memory _times
address[] memory _nfts
uint256 _start

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Team Update

The team replied with the following statement:

This is not necessarily the case, usually distinguishing between parameters starting with an underscore and variables.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	contracts/Pool.sol#L278,307
Status	Acknowledged

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(_start >= 0 && _start < _end, "index error")
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

Team Update

The team replied with the following statement:

This is a parameter required for querying records, and the starting point index must be greater than or equal to 0 and less than the ending point index.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/Pool.sol#L336,337
Status	Acknowledged

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint8 state
uint256 time
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

Team Update

The team replied with the following statement:

If these uint8 and uint256 are not initialized, they default to 0 values.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/Pool.sol#L126,247,248,249,420
Status	Acknowledged

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
(bool success, ) = contractLogic.delegatecall(constructData)
signAddr = _signAddr
feeAddr = _feeAddr
tokenAddr = _tokenAddr
payable(_account).transfer(_realAmount)
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

Team Update

The team replied with the following statement:

There is no need to worry about this, as the owner only has permission for the relevant settings. Flash redemption has been determined to be greater than 0 in another admin contract.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	contracts/Pool.sol#L122,133,158
Status	Acknowledged

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly { // solium-disable-line

sstore(0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7,
contractLogic)
        }

...
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

Team Update

The team replied with the following statement:

Don't worry about this, it's a universal solution.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/Pool.sol#L2
Status	Acknowledged

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.17;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Team Update

The team replied with the following statement:

Don't worry, the contract deployed by the owner account is considered valid.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/Pool.sol#L241,383,423,425,435,458,460
Status	Acknowledged

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(_addr).transfer(msg.sender, _amount)
IERC20(tokenAddr).transferFrom(msg.sender, address(this), amount)
IERC20(tokenAddr).transfer(feeAddr, _feeAmount)
IERC20(tokenAddr).transfer(_account, _realAmount)
IERC20(tokenAddr).transfer(feeAddr, _amount)
IERC20(tokenAddr).transfer(feeAddr, feeAmount)
IERC20(tokenAddr).transfer(msg.sender, realAmount)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Team Update

The team replied with the following statement:

Don't worry about this, it will definitely be successful, and we will also ensure that the pool has sufficient AIA coins.

Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
OwnableEx	Implementation	Context		
	initOwner	Internal	✓	
	owner	Public		-
	isAdmin	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
	modificationAdmin	Public	✓	onlyOwner
	getAllAdmins	Public		-
	getAdmin	Public		-
	getAdminByIndex	Public		-
AdminOwner	Implementation	OwnableEx		
		Public	✓	-
AdminOwnerUpgradable	Implementation	OwnableEx		
		Public	✓	-
	constructor1	Public	✓	-
Proxy	Implementation			

		Public	✓	-
		External	Payable	-
		External	Payable	-
Proxiable	Implementation			
	updateCodeAddress	Internal	✓	
	proxiableUUID	Public		-
MultiPriceOracle	Interface			
	assetPrices	External		-
Pool	Implementation	AdminOwnerUpgradable		
		Public	✓	-
		External	Payable	-
	constructor1	Public	✓	-
	withdrawCoin	External	✓	onlyOwner
	setAddrs	External	✓	onlyOwner
	setBase	External	✓	onlyOwner
	setNfts	External	✓	onlyOwner
	getBase	Public		-
	getPrice	Public		-
	getStakeRecords	Public		-
	getReceiveRecords	Public		-
	check	Public		-

	stakeA	External	Payable	-
	flashStake	External	✓	onlyAdmin
	flashReceive	External	✓	onlyAdmin
	destroy	External	✓	onlyAdmin
	receiveA	External	✓	-
	verifySignature	Public		onlyOwner
	_verifySignature	Private		
PoolUpdateable	Implementation	Pool, Proxiable		
	updateCode	Public	✓	onlyOwner
Strings	Library			
	toString	Internal		
	toString	Internal		
	toHexString	Internal		
	toHexString	Internal		
	toHexString	Internal		
	equal	Internal		
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		

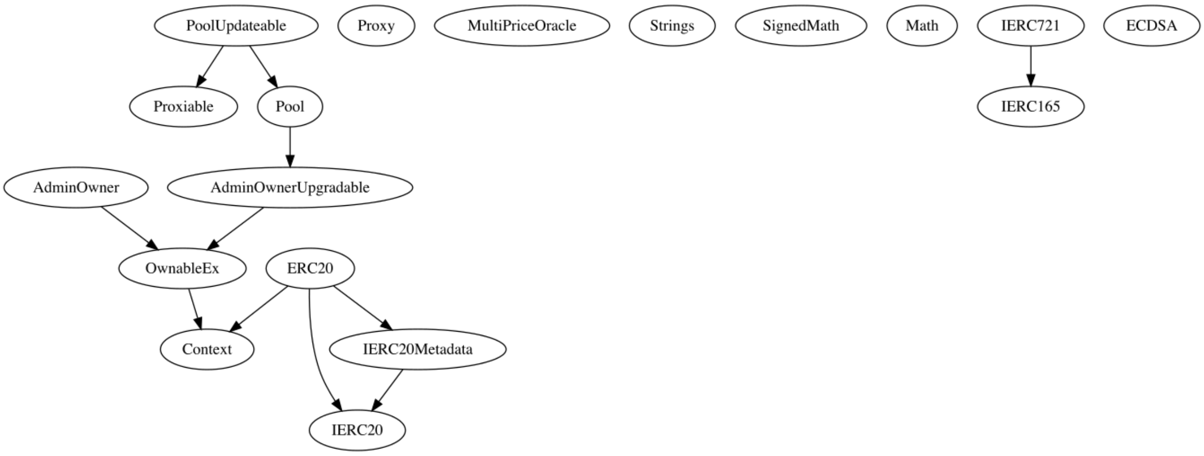
SignedMath	Library			
	max	Internal		
	min	Internal		
	average	Internal		
	abs	Internal		
Math	Library			
	max	Internal		
	min	Internal		
	average	Internal		
	ceilDiv	Internal		
	mulDiv	Internal		
	mulDiv	Internal		
	sqrt	Internal		
	sqrt	Internal		
	log2	Internal		
	log2	Internal		
	log10	Internal		
	log10	Internal		
	log256	Internal		
	log256	Internal		
IERC165	Interface			
	supportsInterface	External		-

ECDSA	Library			
	_throwError	Private		
	tryRecover	Internal		
	recover	Internal		
	tryRecover	Internal		
	recover	Internal		
	tryRecover	Internal		
	recover	Internal		
	toEthSignedMessageHash	Internal		
	toEthSignedMessageHash	Internal		
	toTypedDataHash	Internal		
	toDataWithIntendedValidatorHash	Internal		
IERC721	Interface	IERC165		
	balanceOf	External		-
	ownerOf	External		-
	safeTransferFrom	External	✓	-
	safeTransferFrom	External	✓	-
	transferFrom	External	✓	-
	approve	External	✓	-
	setApprovalForAll	External	✓	-
	getApproved	External		-
	isApprovedForAll	External		-
IERC20	Interface			

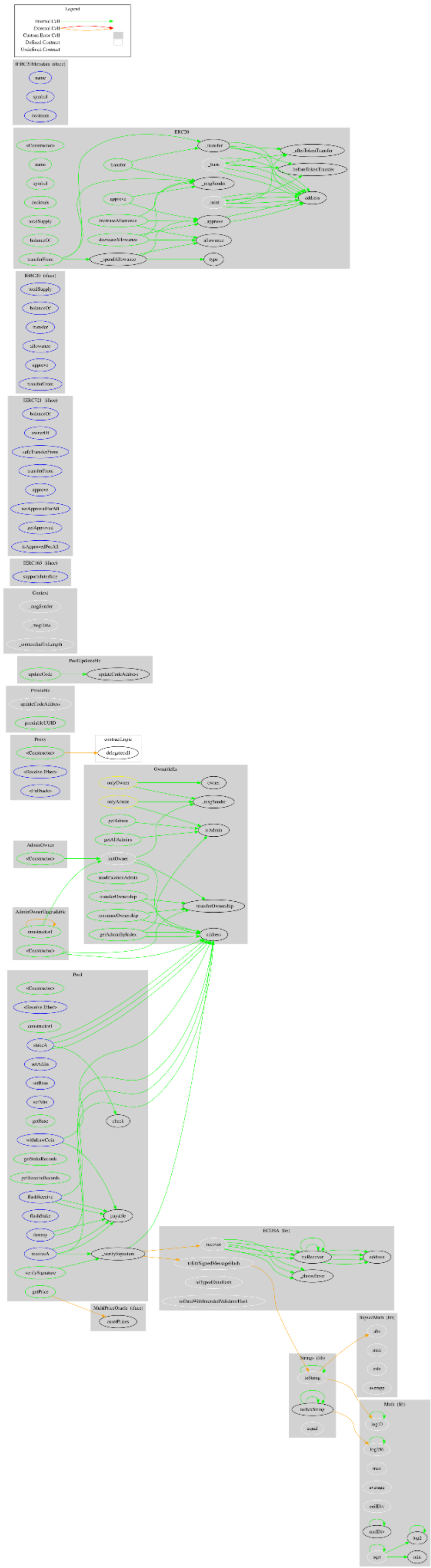
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	

	_spendAllowance	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
IERC20Metadata	Interface	IERC20		
	name	External		-
	symbol	External		-
	decimals	External		-

Inheritance Graph



Flow Graph



Summary

AIA Power contract implements a staking mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io