Cyberscope

Audit Report

Eagle AI Staking

March 2025

Network : BASE

Address : 0x18f8d9193af3bbe7f79100dafc0aa40421f8036e

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

Overview

The EAIStaking contract has undergone a comprehensive review of its staking contract. This contract involves staking EAI tokens to earn rewards in both EAI and USDC. The core functionality consists of a staking mechanism where users can deposit EAI tokens and receive rewards based on their locked balance during an epoch. Users can stake their tokens, which are then locked for a specified period, and claim rewards once the epoch concludes. Main functionalities of the contract include:

Staking:

The contract allows users to stake EAI tokens, which are initially marked as pending. These tokens remain pending until at least one epoch has passed, after which they may be designated as locked and become eligible to earn rewards. Stakes during new epochs lock pending stakes from past epochs. The owner also has the ability to process pending stakes for all users. In exchange for the staked tokens, tdEAI tokens are minted at a 1:1 ratio. Users can unstake their tokens by burning the minted tdEAI tokens. Unstaking is possible immediately after the deposit.

Unstaking

The owner can distribute rewards to stakers in the form of EAI or USDC by transferring funds to the contract. Rewards are calculated for each staking epoch proportionally to the staked amount. Users can claim rewards once per epoch.

Claiming process

Users can claim rewards individually for past epochs through the claimReward function. Additionally, the contract provides functionality for users to claim retroactively for multiple past epochs.

Review

Explorer	https://basescan.org/address/0x18f8d9193af3bbe7f79100dafc0aa40421f8036e

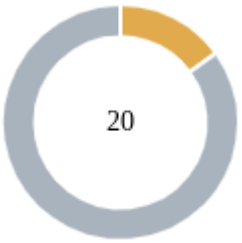
Audit Updates

Initial Audit	26 Mar 2025

Source Files

Filename	SHA256
EAIStaking.sol	6663589b0ae2230c002622d9b9a1757f56c7328ba6a25e69ccdecc0961a47b6e

Findings Breakdown



Critical	0
Medium	3
Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	3	0	0	0
Minor / Informative	17	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	IEI	Inconsistent Epoch Identification	Unresolved
Medium	IPU	Inconsistent Pre-Epoch Unstake	Unresolved
Medium	TIRC	Time Invariant Reward Calculation	Unresolved
Minor / Informative	CO	Code Optimization	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	IUP	Immediate Unstake Period	Unresolved
Minor / Informative	ISA	Inactive Staked Amount	Unresolved
Minor / Informative	ICM	Incompatible Claim Methods	Unresolved
Minor / Informative	MEM	Misleading Error Messages	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	RC	Repetitive Calculations	Unresolved
Minor / Informative	TUU	Time Units Usage	Unresolved
Minor / Informative	TSI	Tokens Sufficiency Insurance	Unresolved

●	L02	State Variables could be Declared Constant	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

IEI - Inconsistent Epoch Identification

Criticality	Medium
Location	EAIStaking.sol#L466
Status	Unresolved

Description

The contract allows for the staking of tokens provided that `isContractActive` is set to `true`. This enables users to stake tokens even when epochs have not yet started (`isEpochStarted` is `false`). The contract accommodates this by directly allocating the deposited funds as locked for the upcoming first epoch. Users who choose to deposit before epochs have started can unstake using the `unstake` method. However, in this scenario, the stored information is attributed to `currentEpochNumber = 0`. As a result, a significant inconsistency arises between the stake and unstake methods during the pre-epoch period. This discrepancy affects the consistency of operations throughout the contract.

```
function unstake(uint256 amount) external nonReentrant
whenNotPaused whenContractActive {
...
uint256 currentEpochNumber = getCurrentEpochNumber();
if (epochs[currentEpochNumber].totalStaked >= remaining) {
epochs[currentEpochNumber].totalStaked -= remaining;
epochs[currentEpochNumber].totalLockstaked -= remaining;
} else {
epochs[currentEpochNumber].totalStaked = 0;
epochs[currentEpochNumber].totalLockstaked = 0;
}
...
}
```

Recommendation

The team is advised to revise the current implementation to ensure that in all cases, stored variables accurately reflect the actual state of the contract. The team should ensure consistency between the stake and unstake methods throughout the contract's lifecycle.

IPU - Inconsistent Pre-Epoch Unstake

Criticality	Medium
Location	EAIStaking.sol#L673
Status	Unresolved

Description

The contract enables the staking of tokens provided `isContractActive` is set to `true`. This allows users to stake tokens even when epochs have not yet started (`isEpochStarted` is `false`). The contract accommodates this by directly allocating the deposited funds as locked for the upcoming first epoch. Users who choose to deposit before epochs have started can unstake using the `unstake` method. However, if users attempt to partially unstake, only the first unstake will succeed. All subsequent attempts will revert due to an underflow error, as described in finding `IEI`. Specifically, the contract attempts to subtract from `epochs[0].totalLockstaked`, which is zero. This will lead to failed unclaims and potential loss of funds.

```
function unstake(uint256 amount) external nonReentrant
whenNotPaused whenContractActive {
...
if (epochs[currentEpochNumber].totalStaked >= remaining) {
epochs[currentEpochNumber].totalStaked -= remaining;
epochs[currentEpochNumber].totalLockstaked -= remaining;
} else {
epochs[currentEpochNumber].totalStaked = 0;
epochs[currentEpochNumber].totalLockstaked = 0;
}
...
}
```

Recommendation

The team is advised to revise the current implementation to ensure that funds are always accessible. In particular, the team should ensure that, in all cases, stored variables accurately reflect the actual state of the contract. An effective measure to prevent this issue is to ensure that epochs have been activated before users stake their tokens.

TIRC - Time Invariant Reward Calculation

Criticality	Medium
Location	EAIStaking.sol#L861
Status	Unresolved

Description

The contract distributes rewards to stakers proportionally to their locked balance during an epoch. However, the calculation does not account for temporal variability within a period. A stake made at the start of an epoch is treated the same as a stake made at the end of an epoch. Given that an epoch may last 30 days, users are incentivized to stake only at the end of the period, receiving the same rewards as if they had staked from the start. This may lead to an inconsistency with the intended behavior of the contract.

```
function claimReward(uint256 epoch, bool isUSDC) external
nonReentrant whenNotPaused whenContractActive {
...
uint256 userStake = epochData.rewardStakeBalance[msg.sender];
...
if (isUSDC) {
require(epochData.usdcRewardStatus[msg.sender]==false, "Already
claimed epoch." );
rewardAmount = (userStake * epochData.usdcRewards) /
epochData.totalLockstaked;
...
}
```

Recommendation

It is advisable to account for variability within an epoch. This would incentivize users to maintain their position throughout the entire period. Alternatively, opting for shorter epochs may be an effective measure.

CO - Code Optimization

Criticality	Minor / Informative
Location	EAIStaking.sol#L763
Status	Unresolved

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 iterates over stakers that are not active and may have a zero balance.

```
function processStakesForNewEpoch() external onlyOwner
whenNotPaused whenContractActive {
...
}
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. It is preferable that the stakers mapping always reflect the state of the contract by removing inactive users. That way the efficiency and performance of the source code will be improved, reducing the cost of execution.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	EAIStaking.sol#L519,531,542,735,763,825,1025,1038,1129,1144
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. Main functionalities such as initiating the reward mechanism and pausing the stake and unstake processes are dependent on the owner. 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 activateContract() external onlyOwner {...}
function setEpoch1date(uint256 dateTimestamp) external
onlyOwner{...}
function startEpoch1() external onlyOwner whenContractActive {...}
function updateEpochTime() external onlyOwner whenNotPaused
whenContractActive {...}
function processStakesForNewEpoch() external onlyOwner
whenNotPaused whenContractActive {...}
function distributeRewards(uint256 amount, bool isUSDC) external
onlyOwner whenNotPaused whenContractActive {...}
function pauseContract() external onlyOwner whenContractActive
{...}
function resumeContract() external onlyOwner {...}
function withdrawEAI(uint256 amount) external onlyOwner {...}
function withdrawUSDC(uint256 amount) external onlyOwner {...}
```

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.

IUP - Immediate Unstake Period

Criticality	Minor / Informative
Location	EAIStaking.sol#L673
Status	Unresolved

Description

The contract implements the unstake method to enable users to withdraw their past deposits made with the stake method. However, the unstaking of a deposit may be executed immediately after a deposit is made, potentially within the same block. Implementing a minimal waiting period between the staking and unstaking of tokens is a common preemptive measure that enhances the security of the contract.

```
function unstake(uint256 amount) external nonReentrant
whenNotPaused whenContractActive {
...
}
```

Recommendation

The team could consider implementing such a time period between the staking and unstaking of tokens to prevent attempts at system manipulation by untimely user operations.

ISA - Inactive Staked Amount

Criticality	Minor / Informative
Location	EAIStaking.sol#L595,763
Status	Unresolved

Description

The contract implements a complex staking mechanism where deposits from past epochs are activated to earn rewards during future epochs. This implies that there will always be an amount of staked tokens that do not earn rewards unless the owner executes the `processStakesForNewEpoch` function, which transfers all pending stakes to active locked stakes.

```
function processStakesForNewEpoch() external onlyOwner
whenNotPaused whenContractActive {
...
}
```

Recommendation

It is advisable for the team to implement a measure that allows users to activate their deposits to be eligible to earn rewards soon after a stake is completed. A waiting period may nevertheless be neccessary to prevent manipulation. This would increase decentralization and user trust in the system.

ICM - Incompatible Claim Methods

Criticality	Minor / Informative
Location	EAIStaking.sol#L861,927
Status	Unresolved

Description

The contract implements a set of claim methods to enable stakers to claim their rewards. Specifically, the contract includes the `claimReward` method and the `claimAll` method. The former is used to claim rewards for a specific past epoch, whereas the latter allows users to iterate over a range of epochs to claim available rewards. However, the `claimAll` method includes user-specific variables that are not tracked or updated in the `claimReward` method. These include the `lastClaimEpoch` and resetting the `rewardStakeBalance` for the user.

```
function claimAll() external nonReentrant whenNotPaused
whenContractActive {
...
ep.rewardStakeBalance[msg.sender] = 0;
user.lastClaimEpoch = epoch;
}
```

Recommendation

It is advisable that the team ensures all claim functionalities align with each other to ensure optimal user experience while enhancing trust in the system

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	EAIStaking.sol#L830
Status	Unresolved

Description

The contract is missing accurate error messages. Specifically, the existing error messages do not accurately reflect the problems, making it difficult to identify and fix issues. As a result, users may struggle to find the root cause of errors. In particular, the `totalStakedAmount` variable records the cumulative staked amount of the contract, not the amount for a specific epoch, as indicated by the error message.

```
require(totalStakedAmount > 0, "No stakes in current epoch");
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	EAIStaking.sol#L735,1025
Status	Unresolved

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.

```
function pauseContract() external onlyOwner whenContractActive {
lastPauseTime = block.timestamp;
_pause();
}
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	EAIStaking.sol#L616,638,644,837,840
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
function stake(uint256 amount) external nonReentrant whenNotPaused
whenContractActive {
...
require(eaiToken.transferFrom(msg.sender, address(this), amount),
"Transfer failed");
...
totalStakedAmount += amount;
...
}
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	EAIStaking.sol#L595
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

```
function stake(uint256 amount) external nonReentrant whenNotPaused
whenContractActive {
...
if(currentEpoch == 0){
userinfo.lastStakeTimestamp = epochStartTime;
userinfo.lastStakeEpoch = 1;
epochs[getCurrentEpochNumber()+1].totalStaked = totalStakedAmount;
epochs[getCurrentEpochNumber()+1].totalLockstaked = totalStakedAmount;
epochs[getCurrentEpochNumber()+1].rewardStakeBalance[msg.sender] = amount;
userinfo.lastRolloverStake = 1;
// For simplicity, all stakes made during the current epoch are recorded as
pending.
userinfo.lockedStake += amount;
}else{
userinfo.lastStakeTimestamp = block.timestamp;
userinfo.lastStakeEpoch = getCurrentEpochNumber();
epochs[getCurrentEpochNumber()].totalStaked = totalStakedAmount;
epochs[getCurrentEpochNumber()].rewardStakeBalance[msg.sender] =
userinfo.lockedStake;
userinfo.pendingStake += amount;
}
...
}
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

TUU - Time Units Usage

Criticality	Minor / Informative
Location	EAIStaking.sol#L455
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 public constant EPOCH_DURATION = 30 * 86400;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	EAIStaking.sol#L825
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
function distributeRewards(uint256 amount, bool isUSDC) external
onlyOwner whenNotPaused whenContractActive {
...
}
```

Recommendation

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

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	EAIStaking.sol#L466
Status	Unresolved

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.

```
uint256 public batchSize = 1000
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	EAIStaking.sol#L533
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
epochStartTime = dateTimestamp
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	EAIStaking.sol#L873,884
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
require(epochData.usdcRewardStatus[msg.sender]==false, "Already
claimed epoch." );


require(epochData.eaiRewardStatus[msg.sender]==false, "Already
claimed epoch." );
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	EAIStaking.sol#L567,737
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 timeElapsed = block.timestamp - epochStartTime -
totalPauseDuration;
uint256 completedEpochs = timeElapsed /EPOCH_DURATION;
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	EAIStaking.sol#L6,88,168,199,306,406
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.20;
pragma solidity ^0.8.0;
pragma solidity 0.8.26;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	EAIStaking.sol#L168,199
Status	Unresolved

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 ^0.8.20;
pragma solidity ^0.8.0;
```

Recommendation

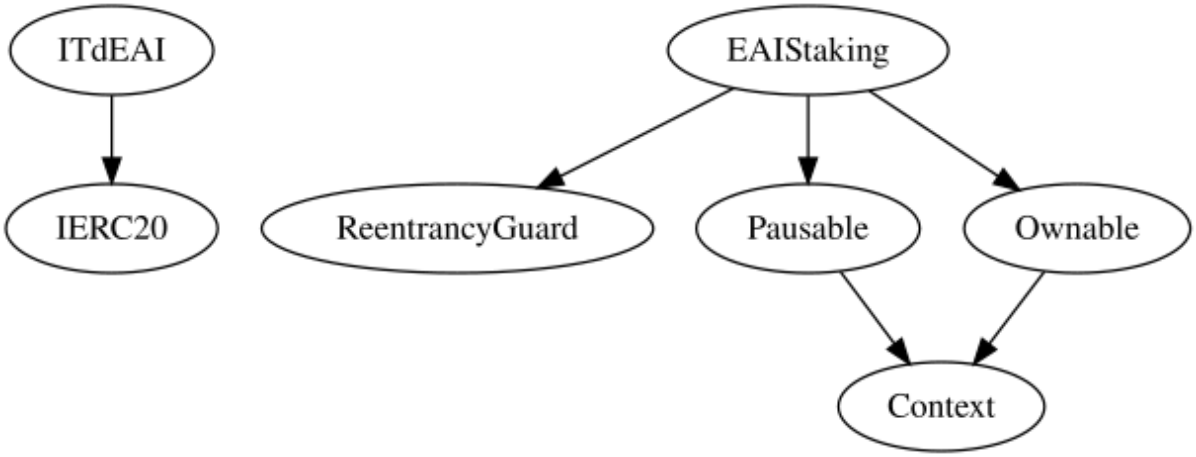
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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.

Functions Analysis

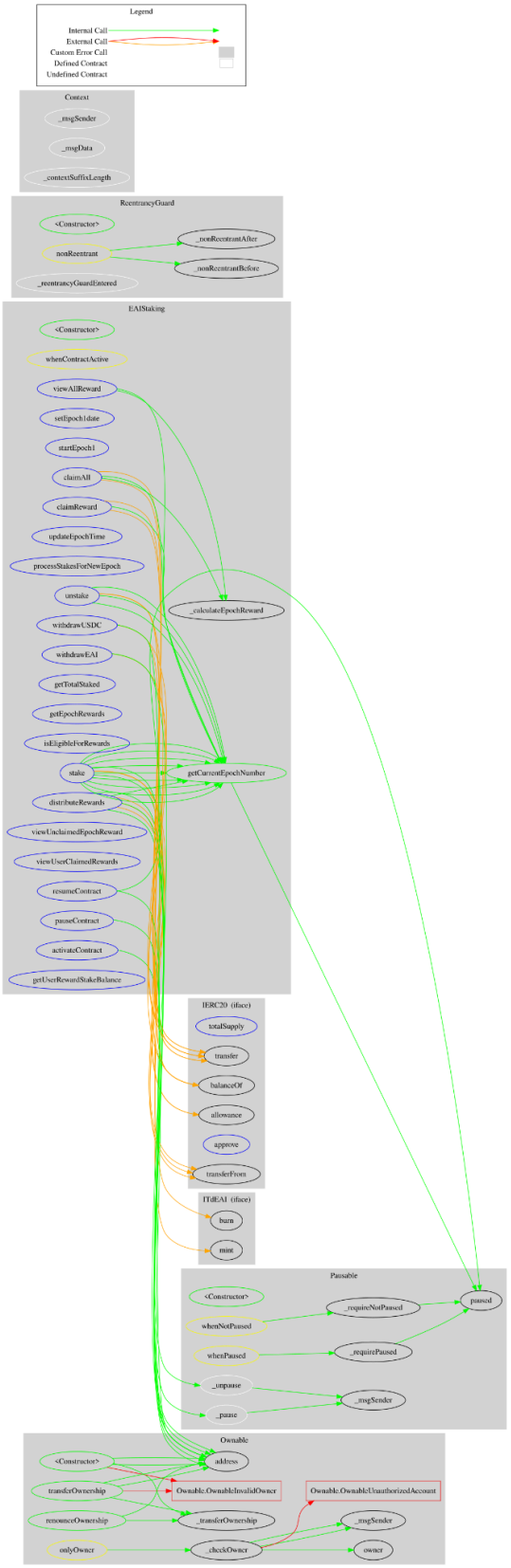
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ITdEAI	Interface	IERC20		
	mint	External	✓	-
	burn	External	✓	-
EAIStaking	Implementation	ReentrancyGuard, Pausable, Ownable		
		Public	✓	Ownable
	activateContract	External	✓	onlyOwner
	setEpoch1date	External	✓	onlyOwner
	startEpoch1	External	✓	onlyOwner whenContractActive
	getCurrentEpochNumber	Public		-
	stake	External	✓	nonReentrant whenNotPaused whenContractActive
	unstake	External	✓	nonReentrant whenNotPaused whenContractActive
	updateEpochTime	External	✓	onlyOwner whenNotPaused whenContractActive

	processStakesForNewEpoch	External	✓	onlyOwner whenNotPaused whenContractActive
	distributeRewards	External	✓	onlyOwner whenNotPaused whenContractActive
	claimReward	External	✓	nonReentrant whenNotPaused whenContractActive
	_calculateEpochReward	Internal		
	claimAll	External	✓	nonReentrant whenNotPaused whenContractActive
	getTotalStaked	External		-
	getEpochRewards	External		-
	isEligibleForRewards	External		-
	pauseContract	External	✓	onlyOwner whenContractActive
	resumeContract	External	✓	onlyOwner
	viewUnclaimedEpochReward	External		-
	viewUserClaimedRewards	External		-
	viewAllReward	External		-
	withdrawEAI	External	✓	onlyOwner
	withdrawUSDC	External	✓	onlyOwner
	getUserRewardStakeBalance	External		-

Inheritance Graphs



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
Context
_msgSender
_msgData
_contextSuffixLength
ReentrancyGuard
<Constructor>
nonReentrant
_reentrancyGuardEntered
_nonReentrantAfter
_nonReentrantBefore
EAIStaking
<Constructor>
whenContractActive
viewAllReward
setEpoch1date
startEpoch1
claimAll
claimReward
updateEpochTime
processStakesForNewEpoch
unstake
withdrawUSDC
withdrawEAI
getTotalStaked
getEpochRewards
isEligibleForRewards
stake
distributeRewards
viewUnclaimedEpochReward
viewUserClaimedRewards
resumeContract
pauseContract
activateContract
getUserRewardStakeBalance
_calculateEpochReward
getCurrentEpochNumber
IERC20 (iface)
totalSupply
transfer
balanceOf
allowance
approve
transferFrom
ITdlEAI (iface)
burn
mint
Pausable
<Constructor>
whenNotPaused
whenPaused
_unpause
_pause
_requireNotPaused
_requirePaused
paused
_msgSender
Ownable
<Constructor>
transferOwnership
renounceOwnership
onlyOwner
address
_transferOwnership
checkOwner
Ownable.OwnableInvalidOwner
Ownable.OwnableUnauthorizedAccount
_msgSender
owner

Summary

Eagle AI contract implements a staking and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements. The audit process identified no critical issues or compiling concerns.

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