Cyberscope

Audit Report

Autonomi

May 2025

Repository	https://github.com/lajosdeme/impossible-futures-contracts
Commit	a9605d97176cccc7fe36179d8306a054ee218c2f
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/lajosdeme/impossible-futures-contracts
Commit	a9605d97176cccc7fe36179d8306a054ee218c2f

Audit Updates

Initial Audit	05 May 2025

Source Files

Filename	SHA256
Phase1Voting.sol	c4ad16551a9866d0a134b1f1f414c73d72250bb92890255039f3f9cf1cb45e1b
interfaces/IPhase1Voting.sol	f62f3363c2763ebae886bc425b99e6c67794587a2ac44dea3783749d65ab71ca
interfaces/IAppRegistry.sol	8da9d7591854b5af3541fbd41419eeb44c5d03bdb01ba3e6ac11595f3260f8d9

Overview

The `Phase1Voting` contract is a mechanism designed to facilitate a voting process where users can cast votes for registered applications using the ANT token. It implements a Logarithmic Market Scoring Rule (LMSR) cost function to determine the cost of votes, with a time-based multiplier to discourage late voting. The contract integrates with an external IAppRegistry to validate app IDs and supports a one-week voting period, with owner-controlled parameters to manage the process. Key functionalities include voting, querying vote data, calculating vote costs and prices, and administrative controls for the owner.

Voting for Applications

The contract allows users to vote for registered applications by calling the `vote` function, specifying an `appId` and the number of votes (`newVotes`). Votes must meet a minimum threshold (`MIN_VOTE_AMOUNT = 1 ether`), and the app ID must be valid per the `IAppRegistry` . The cost of voting is calculated using the `userCost` function, which applies the LMSR cost function and a time-based multiplier. Votes are recorded in the `votesForApp` mapping for the app and the `userVotes` array for the user, and the required ANT tokens are transferred from the user to a designated beneficiary. The function emits a `Voted` event to log the action.

Retrieving Leaderboard Data

The `getLeaderboard` function provides a sorted list of all registered apps and their total votes, enabling users to view the current ranking of applications. It fetches app IDs from IAppRegistry, constructs an array of Vote structs containing each app's ID and vote count, and sorts the array in descending order using a bubble sort algorithm. This functionality allows users to assess which apps are leading in the voting process, supporting transparency and decision-making.

Querying User Vote Information

The contract offers multiple functions to query user-specific vote data. The `getUserVotes` function returns an array of Vote structs detailing all votes cast by a user, including the app IDs and vote amounts. The `getUserVotesLength` function returns the number of vote entries for a user, useful for iterating over their votes. The `getUserTotalVotes` function calculates the sum of all votes cast by a user by iterating over their userVotes array. These functions provide users with insights into their voting activity and contributions.

Calculating Vote Costs

The `userCost` function computes the cost in ANT tokens for casting a specified number of votes for an app, using the LMSR cost function implemented in the `c` function. It calculates the difference between the cost function before and after adding the new votes (qNew - qOld), scales it by a constant K, and applies a timeMultiplier. The `timeMultiplier` remains 1.0 until 90% of the voting period, then increases quadratically to discourage late voting. This functionality ensures dynamic pricing that reflects vote demand and timing.

The `instantaneousMarketPrice` function calculates the current price for voting on a specific app, based on the LMSR cost function. It computes the price as:

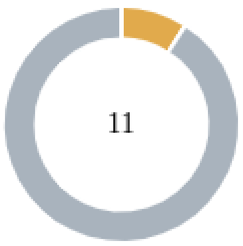
`K x exp(q_i/b - C(q)/b)`, where

(q_i) is the app's vote count, (C(q)) is the total cost function, and b) is the liquidity parameter. This function allows users to assess the cost of voting for an app at any point, providing transparency into the market dynamics of the voting process.

Owner Functionalities

The contract includes administrative functions restricted to the owner, defined via the Ownable inheritance. The `setVotingActive` function toggles the `isVotingActive` flag, enabling or disabling the voting process. The `setAntBeneficiary` function updates the address receiving ANT tokens from votes, ensuring flexibility in fund allocation. The `setB` function adjusts the liquidity parameter b, which influences the sensitivity of vote costs in the LMSR cost function. These functions allow the owner to manage the voting process, configure economic parameters, and ensure operational control, with events emitted for transparency.

Findings Breakdown



●	Critical	0
●	Medium	1
●	Minor / Informative	10

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	10	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🟠	UVP	Unbounded Voting Period	Unresolved
⚪	ACI	Accidental Console Import	Unresolved
⚪	IDI	Immutable Declaration Improvement	Unresolved
⚪	ILS	Inefficient Leaderboard Sorting	Unresolved
⚪	MVC	Missing Validation Checks	Unresolved
⚪	MU	Modifiers Usage	Unresolved
⚪	PTAI	Potential Transfer Amount Inconsistency	Unresolved
⚪	RAA	Redundant Array Access	Unresolved
⚪	UBM	Unrestricted B Modification	Unresolved
⚪	UTPD	Unverified Third Party Dependencies	Unresolved
⚪	L04	Conformance to Solidity Naming Conventions	Unresolved

UVP - Unbounded Voting Period

Criticality	Medium
Location	Phase1Voting.sol#L41,82
Status	Unresolved

Description

The contract is missing an end check for the voting period, allowing votes to be cast beyond the intended duration of one week (`VOTING_DURATION = 604,800 seconds`). The `onlyVotingActive` modifier enforces that `isVotingActive` is `true` but does not verify whether the current `block.timestamp` is within the voting window defined by `VOTING_START_TIME` and `VOTING_DURATION`. As a result, if the owner fails to call `setVotingActive(false)` after the voting period ends, users can continue to call the vote function, potentially skewing the voting results. Additionally, the `timeMultiplier` function, which increases voting costs quadratically after 90% of the period, could lead to exorbitantly high costs for late votes, causing financial loss for users who assume voting is still valid. This oversight undermines the integrity of the voting process and could lead to governance disputes or manipulation of outcomes.

```
    modifier onlyVotingActive() {
        if (!isVotingActive) {
            revert VotingNotActive();
        }
        _;
    }

  function vote(bytes32 appId, uint256 newVotes) external
onlyVotingActive {
        // verify that the app ID is valid
        if (!APP_REGISTRY.isRegisteredApp(appId)) {
            revert InvalidAppId();
        }
        ...
        emit Voted(msg.sender, appId, newVotes);
    }
```

Recommendation

It is recommended to implement a timestamp-based check in the `onlyVotingActive` modifier or the vote function to ensure that `block.timestamp` does not exceed `VOTING_START_TIME + VOTING_DURATION`. This would enforce the intended voting period and prevent votes from being cast after the designated end time. Additionally, consider automating the deactivation of voting by integrating this check, reducing reliance on the owner’s manual intervention.

ACI - Accidental Console Import

Criticality	Minor / Informative
Location	Phase1Voting.sol#L4
Status	Unresolved

Description

The contract imports the console from the `forge-std` lib. This is typically used for debugging purposes during development and testing but should be removed in production to avoid unnecessary dependencies and potential security risks.

```
import "forge-std/src/console.sol";
```

Recommendation

It is recommended to remove imports used for testing when trying to deploy the contract on an actual network.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Phase1Voting.sol#L76
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
maxMultiplier
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

ILS - Inefficient Leaderboard Sorting

Criticality	Minor / Informative
Location	Phase1Voting.sol#L114
Status	Unresolved

Description

The contract is inefficient in its `getLeaderboard` function, which employs a bubble sort algorithm to order apps by their vote counts. Despite being a `view` function, this `O(n^2)` complexity algorithm requires multiple iterations over the array of apps retrieved from `APP_REGISTRY.getAppIds()`, leading to significant gas consumption when the number of apps is large. Even though view functions do not incur transaction costs, the high computational overhead can still result in expensive gas estimates, potentially approaching or exceeding the block gas limit for large app counts. This could cause the function to revert, making the leaderboard inaccessible on-chain and degrading the user experience. The inefficiency undermines the contract's performance and scalability, particularly in scenarios with many registered apps.

```
 function getLeaderboard() external view returns (Vote[] memory) {
        bytes32[] memory apps = APP_REGISTRY.getAppIds();
        Vote[] memory leaderboard = new Vote[](apps.length);

        for (uint256 i = 0; i < apps.length; i++) {
            leaderboard[i] = Vote({appId: apps[i], votes:
votesForApp[apps[i]]});
        }

        // Sort the array using bubble sort
        for (uint256 i = 0; i < leaderboard.length - 1; i++) {
            for (uint256 j = 0; j < leaderboard.length - i - 1; j++) {
                if (leaderboard[j].votes < leaderboard[j + 1].votes) {
                    // Swap positions
                    Vote memory temp = leaderboard[j];
                    leaderboard[j] = leaderboard[j + 1];
                    leaderboard[j + 1] = temp;
                }
            }
        }
        return leaderboard;
    }
```

Recommendation

It is recommended to enhance the efficiency of the `getLeaderboard` function by implementing a more gas-efficient on-chain sorting algorithm, such as quicksort or mergesort, if feasible within gas constraints. Alternatively, consider maintaining a dynamically updated, sorted data structure to track app rankings as votes are cast, eliminating the need for sorting during queries. Additionally, consider document a maximum supported number of apps to ensure the function remains executable within block gas limits.

MVC - Missing Validation Checks

Criticality	Minor / Informative
Location	Phase1Voting.sol#L48
Status	Unresolved

Description

The contract is missing essential validation checks on the constructor parameters `_maxMultiplier` and `_votingStartTime`. Specifically, there is no check to ensure that `_maxMultiplier` is strictly greater than the internal `SCALE` constant, which represents the fixed-point base value of `1e18`. Failing to enforce this can lead to incorrect calculations where the multiplier is expected to scale values above the base unit. Additionally, the contract does not verify that `_votingStartTime` is set to a future timestamp relative to the current block timestamp at deployment. Without this check, the voting process could start immediately or even in the past, undermining the expected timing logic and potentially allowing actions to occur before participants are ready. These oversights may impact the correctness and predictability of the contract's voting and multiplier mechanics.

```
constructor(
        uint256 _b,
        uint256 k,
        IAppRegistry appRegistry,
        IERC20 ant,
        address _antBeneficiary,
        uint256 _votingStartTime,
        uint256 _maxMultiplier
    ) Ownable(msg.sender) {
        ...
        VOTING_START_TIME = _votingStartTime;
        maxMultiplier = _maxMultiplier;
    }
```

Recommendation

It is recommended to add validation that explicitly enforces `_maxMultiplier` to be strictly greater than the `SCALE` constant (1e18) to ensure meaningful scaling behaviour. Furthermore, it is recommended to validate that `` `_votingStartTime `` is set to a future time relative to the current block timestamp during deployment, to guarantee proper scheduling and participation opportunities in the voting process.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	Phase1Voting.sol#L84,169
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
if (!APP_REGISTRY.isRegisteredApp(appId)) {
    revert InvalidAppId();
}
...
if (!APP_REGISTRY.isRegisteredApp(targetAppId)) {
    revert InvalidAppId();
}
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Phase1Voting.sol#L100
Status	Unresolved

Description

The `safeTransferFrom` function is 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

```
ANT.safeTransferFrom(msg.sender, antBeneficiary, _userCost);
```

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
```

RAA - Redundant Array Access

Criticality	Minor / Informative
Location	Phase1Voting.sol#L114
Status	Unresolved

Description

The contract's `c` function repeatedly fetches the array of application IDs from `APP_REGISTRY.getAppIds()` and accesses its length multiple times across two separate loops used to compute the Logarithmic Market Scoring Rule (LMSR) cost. This function, integral to calculating vote costs and market prices, iterates over the app IDs to sum exponential terms, relying on the array length to control both loops. Each redundant call to `APP_REGISTRY.getAppIds()` and length access incurs unnecessary gas costs, particularly when the number of registered apps is large. These repeated operations increase the computational overhead of the `c` function, which is invoked by userCost and instantaneousMarketPrice, affecting the gas required for voting and price queries. As a result, the contract is inefficient, leading to higher gas costs and reduced performance for users.

```
    function c(bytes32[] memory appIds, bytes32 selectedAppId, uint256
newVotes) public view returns (uint256) {
        ...;
        for (uint256 i = 0; i < appIdsLength; i++) {
            uint256 qi = votesForApp[appIds[i]];
            if (appIds[i] == selectedAppId) {
                qi += newVotes;
            }
            UD60x18 qiFixed = UD60x18.wrap(qi);
            UD60x18 qiOverB = qiFixed.div(bFixed);
            if (qiOverB.gt(maxVal)) {
                maxVal = qiOverB;
            }
        }

        ...
        for (uint256 i = 0; i < appIdsLength; i++) {
            // Get votes for this app (already scaled by 1e18)
            uint256 qi = votesForApp[appIds[i]];
            if (appIds[i] == selectedAppId) {
                qi += newVotes;
            }
        ...
        }
```

Recommendation

It is recommended to optimize the c function by caching the array of application IDs and its length in a single operation before the loops, thereby eliminating redundant accesses and external calls. Storing the result of `APP_REGISTRY.getAppIds()` in a local variable and reusing its length throughout the function will reduce gas consumption. This optimization will enhance the efficiency of vote cost calculations and price queries, improving the user experience.

UBM - Unrestricted B Modification

Criticality	Minor / Informative
Location	Phase1Voting.sol#L323
Status	Unresolved

Description

The contract is at risk due to the unrestricted ability of the owner to modify the liquidity parameter `b` via the `setB` function at any time during the voting process. The `setB` function allows the owner to update `b`, a crusial parameter in the Logarithmic Market Scoring Rule (LMSR) cost function, without restrictions on when the change can occur or its impact on ongoing votes. The value of `b` significantly influences vote pricing: a smaller `b` causes rapid price changes with fewer votes, while a larger `b` makes prices more stable but harder to adjust. Modifying `b` mid-voting could alter the cost dynamics for existing and future votes, potentially affecting the fairness of the voting outcome and user expectations.

For instance, reducing b could make votes cheaper for apps with low vote counts, while increasing b could make additional votes prohibitively expensive, impacting smaller markets differently than larger ones. This lack of temporal or contextual constraints on b changes introduces unpredictability and potential for manipulation, undermining the integrity of the voting mechanism.

```
    function setB(uint256 _b) external onlyOwner {
        if (_b == 0) {
            revert BCantBeZero();
        }

        b = _b;

        emit SetB(_b);
    }
```

Recommendation

It is recommended to restrict modifications to the `b` parameter to ensure stability and fairness during active voting periods. Consider implementing a mechanism that prevents `b` from being changed while voting is active (`isVotingActive = true`) or after `VOTING_START_TIME` begins, preserving consistent pricing for ongoing votes. Alternatively, explore a governance model where b changes require a delay or approval process, allowing users to anticipate adjustments. To address varying market sizes, consider defining multiple b values tailored to different app categories or vote types at contract deployment, rather than modifying a single b dynamically. Comprehensive testing should evaluate the impact of different b values on vote pricing for small and large markets, and clear documentation should guide the owner on selecting appropriate b values to align with market dynamics and expected trading activity.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	Phase1Voting.sol#L72,84,115
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
APP_REGISTRY = appRegistry;
...
if (!APP_REGISTRY.isRegisteredApp(appId)) {
    revert InvalidAppId();
}
...
bytes32[] memory apps = APP_REGISTRY.getAppIds();
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Phase1Voting.sol#L20,22,24,33,304,313,323
Status	Unresolved

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.

```
uint256 public immutable K
IERC20 public immutable ANT
IAppRegistry public immutable APP_REGISTRY
uint256 immutable VOTING_START_TIME
ool _isVotingActive)
ddress _antBeneficiary)
int256 _b)
```

Recommendation

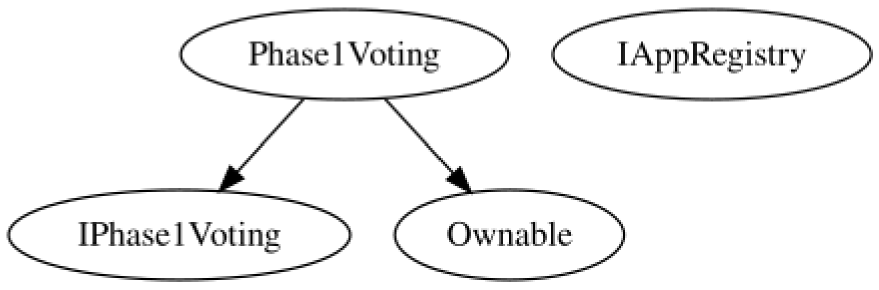
By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Functions Analysis

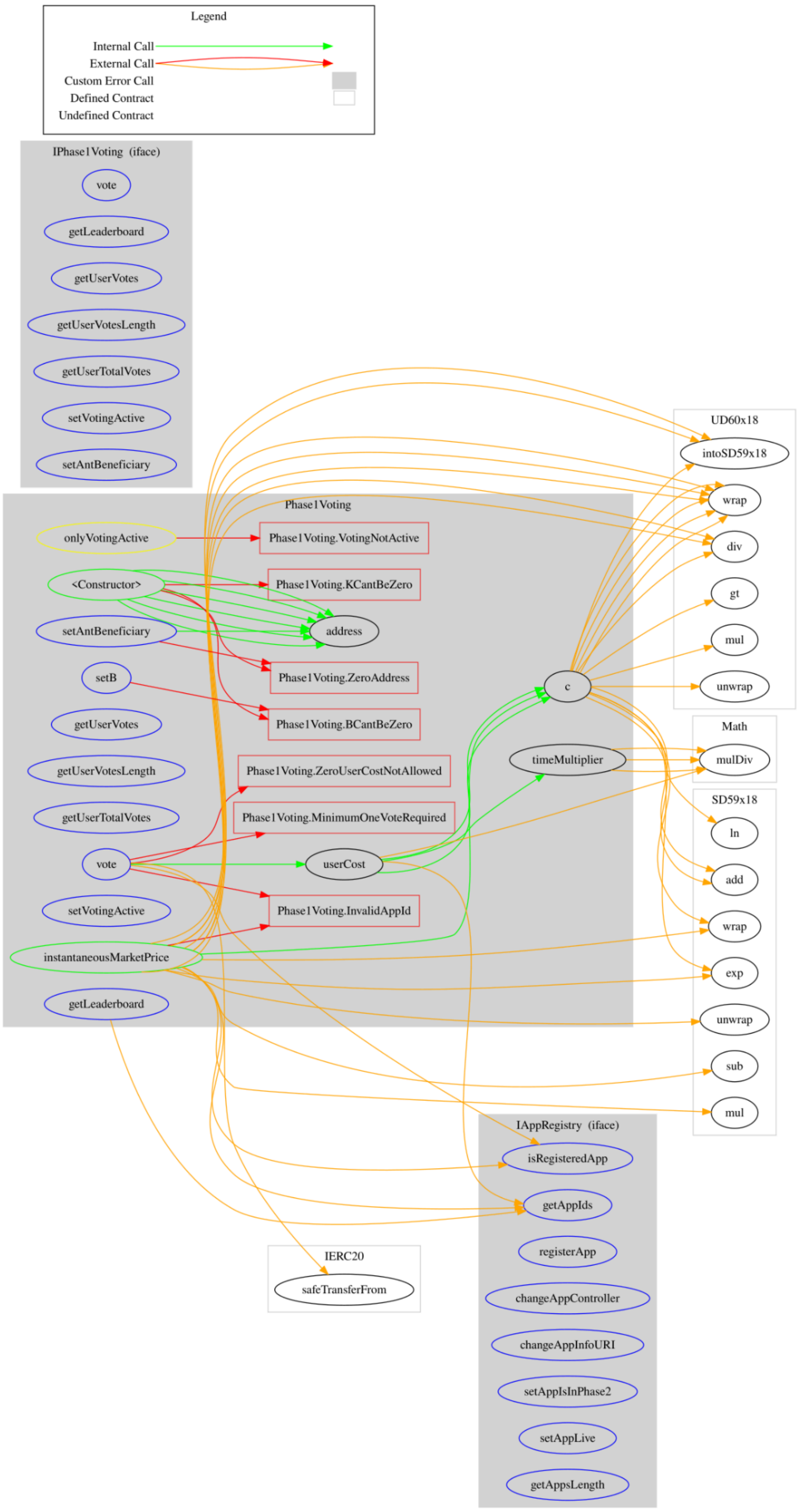
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Phase1Voting	Implementation	IPhase1Voting, Ownable		
		Public	✓	Ownable
	vote	External	✓	onlyVotingActive
	getLeaderboard	External		-
	getUserVotes	External		-
	getUserVotesLength	External		-
	getUserTotalVotes	External		-
	instantaneousMarketPrice	Public		-
	userCost	Public		-
	timeMultiplier	Public		-
	c	Public		-
	setVotingActive	External	✓	onlyOwner
	setAntBeneficiary	External	✓	onlyOwner
	setB	External	✓	onlyOwner
IPhase1Voting	Interface			
	vote	External	✓	-
	getLeaderboard	External		-
	getUserVotes	External		-

	getUserVotesLength	External		-
	getUserTotalVotes	External		-
	setVotingActive	External	✓	-
	setAntBeneficiary	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
IPhase1Voting (iface)
vote
getLeaderboard
getUserVotes
getUserVotesLength
getUserTotalVotes
setVotingActive
setAntBeneficiary
Phase1Voting
onlyVotingActive
Phase1Voting.VotingNotActive
<Constructor>
Phase1Voting.KCantBeZero
setAntBeneficiary
address
setB
Phase1Voting.ZeroAddress
getUserVotes
Phase1Voting.BCantBeZero
getUserVotesLength
Phase1Voting.ZeroUserCostNotAllowed
getUserTotalVotes
Phase1Voting.MinimumOneVoteRequired
vote
userCost
setVotingActive
Phase1Voting.InvalidAppId
instantaneousMarketPrice
getLeaderboard
c
timeMultiplier
UD60x18
intoSD59x18
wrap
div
gt
mul
unwrap
Math
mulDiv
SD59x18
ln
add
wrap
exp
unwrap
sub
mul
IAppRegistry (iface)
isRegisteredApp
getAppIds
registerApp
changeAppController
changeAppInfoURI
setAppIsInPhase2
setAppLive
getAppsLength
IERC20
safeTransferFrom

Summary

The Autonomi `Phase1Voting` contract implements a voting mechanism using a Logarithmic Market Scoring Rule (LMSR) to facilitate user voting for registered applications with ANT tokens, incorporating a time-based cost multiplier to discourage late votes. This audit investigates security issues, business logic concerns, and potential improvements, like optimizing gas usage, to ensure the voting process's robustness, fairness, and cost-effectiveness.

Disclaimer

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