Zellic

Prepared for
Jeff Z
Opinion Labs

Prepared by
Qingying Jie
Jisub Kim
Zellic

October 31, 2024

Prediction Market
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Opinion Labs from October 8th to October 11th, 2024. During this engagement, Zellic reviewed Prediction Market's code for security vulnerabilities, design issues, and general weaknesses in security posture.

This audit was a diff audit between following commits:

1. Git commit `bc355791` ↗ in the main branch of the ctf-exchange/* repository, and Git commit 2745c301 ↗ in the main branch of the Polymarket/ctf-exchange repository.
2. Git commit `bc355791` ↗ in the main branch of the conditional-tokens-contracts/* repository, and Git commit eeefca66 ↗ in the master branch of the gnosis/conditional-tokens-contracts repository.
3. Git commit `bc355791` ↗ in the main branch of the optimistic-oracle/contracts/* folder, and Git commit 6e9ab5fe ↗ in the master branch of the UMAprotocol/protocol repository.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker drain user funds?
- Could a malicious user disrupt the voting system?
- Are the libraries implemented correctly?
- Could an on-chain attacker conduct a successful a flash-loan attack?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Forked projects
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

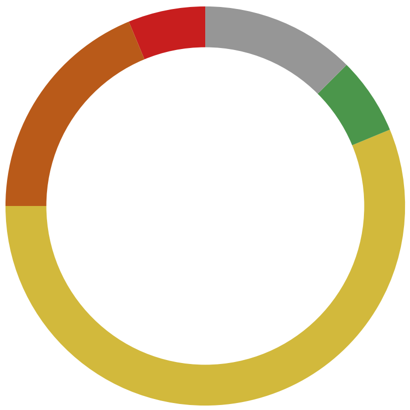
1.4. Results

During our assessment on the scoped Prediction Market contracts, we discovered 16 findings. One critical issue was found. Three were of high impact, nine were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Opinion Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	3
🟨 Medium	9
🟩 Low	1
⬜ Informational	2



2. Introduction

2.1. About Prediction Market

Opinion Labs contributed the following description of Prediction Market:

> Opinion Labs democratizes access to global trading by eliminating cross-market frictions and providing a seamlessly integrated platform with unified liquidity. We are building the world's opinion protocol and creating a new trading paradigm. Opinion Labs enables anyone to create any predictions using any token, in a decentralized, permissionless way. We empower users to predict, trade, and verify truth like never before.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Prediction Market Contracts

Type	Solidity
Platform	EVM-compatible
Target	prediction-market-smart-contract-v2
Repository	https://github.com/OpinionLabs/prediction-market-smart-contract-v2 ↗
Version	`bc355791ccffde709a4662f8ecc2ff8044cce669`

Programs	`conditional-tokens-contracts/*`
	`optimistic-oracle/contracts/*`
	`ctf-exchange/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-days. The assessment was conducted by two consultants over the course of four calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io

Jisub Kim
Engineer
jisub@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 8, 2024	Kick-off call
October 8, 2024	Start of primary review period
October 11, 2024	End of primary review period

3. Detailed Findings

3.1. Lack of access-control modifier on setting voter

Target	VotingV2		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The VotingV2 contract contains two critical functions — `votersPickedSuccessCallback` and `votersPickedFailureCallback` — that are responsible for setting the voters for a specific trial and handling failures in voter selection.

```
function votersPickedSuccessCallback(
    uint256 requestId, address[] calldata voters
) external override {
    // emit event - VotingActionRequired(requestId, voters);
    // update the jury of the specific "trial" corresponding to the requestId
    _setVoter(voterRequest[requestId], voters);
}

function votersPickedFailureCallback(uint256 requestId) external override {
    // set the requestId to failed
    uint32 roundId = voterRequest[requestId];
    if (roundId > 0) {
        delete voterRequest[requestId];
        delete roundRequest[roundId];

        emit VoterRequestFailed(roundId, requestId);
    }
}
```

However, these functions lack proper access-control modifiers, such as `onlyAuthorized` or `onlyVoterRegistrar`, which means they can be called by any external account. This allows any user to manipulate the voter-selection process by arbitrarily setting or deleting voters, potentially compromising the integrity of the voting system.

Impact

An arbitrary user can call and manipulate the voter selection.

Recommendations

Consider adding appropriate access-control modifiers.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `72635a9d` ↗.

3.2. Typographical error in the remove function causes inconsistent state

Target	DynamicArray		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In the `remove()` function of the DynamicArray library, there is a typo where an assignment operator (=) was intended, but an equality check operator (==) is used instead. This mistake leads to a no-operation (no-op), and the state variable `self.contains` is not updated as intended — specifically, this code:

```
function remove(Data storage self, uint256 _id) internal {
    // [..]
    self.contains[self.items[index].data] == 0; // here
}
```

Impact

This typo prevents the `remove()` function from updating the `contains` mapping, which tracks whether an item is present in the array. As a result, the item may still be considered present even after it has been removed from the array, leading to inconsistent state and potential errors in any logic that relies on the `contains` mapping.

Recommendations

Consider modifying this to an assignment operator properly.

```
function remove(Data storage self, uint256 _id) internal {
    // [..]
    self.contains[self.items[index].data] == 0;
    self.contains[self.items[index].data] = 0;
}
```

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `72635a9d` ↗.

3.3. Incorrect `rewardRate` calculation

Target	Staker		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The owner of the Staker contract can call the `notifyRewardAmount` function to set the amount of voting tokens rewarded to all staked voters per second (i.e., the `rewardRate`).

The variable `rewardDuration` defines the duration of the reward distribution, and the `periodFinish` is the timestamp when the reward distribution ends. If the owner calls the `notifyRewardAmount` function after the reward distribution has ended, the `rewardRate` is correctly calculated as `_amount / rewardsDuration`.

However, if the function is called during an ongoing reward-distribution period, the total reward becomes the sum of the newly added amount and the remaining amount from the previous period. The issue arises when the `rewardRate` is incorrectly calculated by using `periodFinish` as the denominator instead of `rewardsDuration`.

```
function notifyRewardAmount(uint256 _amount) external onlyOwner {
    _updateReward(address(0));
    if (block.timestamp >= periodFinish) {
        rewardRate = _amount / rewardsDuration;
    } else {
        uint256 remaining = periodFinish - block.timestamp;
        uint256 leftover = remaining * rewardRate;
        rewardRate = (_amount + leftover) / periodFinish;
    }

    lastUpdateTime = uint64(block.timestamp);
    periodFinish = block.timestamp + rewardsDuration;
    // [...]
}
```

Impact

The reward should be distributed over the `rewardDuration`, but it is instead distributed incorrectly based on the timestamp. This results in a much lower `rewardRate` than expected.

Recommendations

Consider changing the equation `(_amount + leftover) / periodFinish` to `(_amount + leftover) / rewardDuration`.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `3117767d` ↗.

3.4. Lack of comprehensive test suite

Target	All scoped contracts		
Category	Code Maturity	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

During this audit, we observed a number of findings that affect the core logic of the codebase. Some of these findings could result in the failure of a working production environment, even if no malicious attack is assumed.

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The project has very limited test coverage, especially for the code that has been modified or added compared to the forked projects. It is important to test the invariants required for ensuring security and also verify mathematical properties as specified in the documents.

Impact

This code has not been exhaustively tested, increasing the likelihood of potential bugs.

Recommendations

We recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Good test coverage has multiple effects:

- It finds bugs and design flaws early (pre-audit or pre-release).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

Remediation

Opinion Labs introduced additional testing between commits `15111134` ↗ and `c993ef9a` ↗.

3.5. The setVrfCoordinator does not set the state variable coordinator

Target	VoterRegistrar		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

There are two state variables, `coordinator` and `vrfCoordinator`, in the contract VoterRegistrar. They should point to the same VRF coordinator. The difference is that the `coordinator` is of type `IVRFCoordinatorV2Plus`, and the `vrfCoordinator` is of type `address`.

```
IVRFCoordinatorV2Plus coordinator;
address public vrfCoordinator;
```

The admin can set the value of `vrfCoordinator` through the `setVrfCoordinator` function. However, there is no function to set the variable `coordinator`.

```
function setVrfCoordinator(address _vrfCoordinator) external onlyAdmin {
    vrfCoordinator = _vrfCoordinator;
}
```

Additionally, the VoterRegistrar contract does not initialize the variable `vrfCoordinator` in the constructor.

```
constructor(
    uint256 _subscriptionId,
    address _vrfCoordinator,
    bytes32 _keyHash
) VRFConsumerBaseV2Plus(_vrfCoordinator) {
    coordinator = IVRFCoordinatorV2Plus(_vrfCoordinator);
    subscriptionId = _subscriptionId;
    keyHash = _keyHash;
    _grantRole(ADMIN_ROLE, msg.sender);
}
```

Impact

The state variable `coordinator` is used in the `_requestRandomVoters` function to call the VRF coordinator's `requestRandomWords` function, while the state variable `vrfCoordinator` is unused.

The `coordinator` cannot be updated. Meanwhile, updating `vrfCoordinator` will cause it to be inconsistent with the `coordinator`, leading to confusion.

Recommendations

Since the contract VoterRegistrar inherits from the contract VRFConsumerBaseV2Plus, consider using the variable `s_vrfCoordinator` and function `setCoordinator` defined in the contract VRFConsumerBaseV2Plus, or add code to set `coordinator` to the `setVrfCoordinator` function.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit 5bb84492 ↗.

3.6. Snapshots are updated after the balance changes

Target	ERC20Snapshot		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The ERC20Snapshot contract extends an ERC-20 token with a snapshot mechanism. Users can retrieve the balances and total supply at the time a snapshot is created.

If a balance is not updated, the balance when a snapshot is created is the same as the current balance. But if mint, burn, or transfer operations are to be performed right after a snapshot, the balance before the operations must be recorded, which reflects the balance at the time of the snapshot.

The `_update` function in the ERC20 contract contains the logic for updating the account balances and total supply. The ERC20Snapshot contract overrides the `_update` function and appends the logic for updating the snapshot after the balance change.

```
function _update(address from, address to, uint256 amount)
    internal virtual override {
    super._update(from, to, amount);

    if (from == address(0)) {
        // mint
        _updateAccountSnapshot(to);
        _updateTotalSupplySnapshot();
    } else if (to == address(0)) {
        // burn
        _updateAccountSnapshot(from);
        _updateTotalSupplySnapshot();
    } else {
        // transfer
        _updateAccountSnapshot(from);
        _updateAccountSnapshot(to);
    }
}
```

Impact

Since the snapshot is updated after the balance modification, the function that is used to get the balance or total supply at a specified snapshot time actually returns the value after the balance changes at a certain time.

Recommendations

Consider moving the code for updating the snapshots to before calling `super._update`.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `6c95d53b` ↗.

3.7. The `effectiveStake` used to calculate the slash amount is not capped

Target	VotingV2		
Category	Business Logic	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

Voters selected in a round can commit a vote for a price request. The number of votes a voter can cast depends on the amount of staked voting tokens. Additionally, to prevent a single voter from having too much influence, the contract limits the maximum number of tokens in a single vote to `maxTokenPerVote`.

```
function revealVote(
    bytes32 identifier,
    uint256 time,
    int256 price,
    bytes memory ancillaryData,
    int256 salt
) public override nonReentrant {
    // [...]

    // This is capped at maxTokenPerVote to prevent a voter from having too
    much influence.
    uint128 effectiveStake = uint128(
        Math.min(
            voterStakes[voter].stake -
                voterStakes[voter].pendingStakes[currentRoundId],
            maxTokenPerVote
        )
    );

    voteInstance.results.addVote(price, effectiveStake); // Add vote to the
    results.

    // [...]
}
```

In the function `_updateAccountSlashingTrackers`, the slashed amount is calculated based on the voter's `effectiveStake`. But the `effectiveStake` is not capped here.

```
function _updateAccountSlashingTrackers(
    address voter,
    uint64 maxTraversals
) internal {
    // [...]

    uint256 effectiveStake = voterStake.stake -
        voterStake.pendingStakes[trackers.lastVotingRound];

    // effectiveStake should be capped per user by

    // [...]

    else if (participation == VoteParticipation.DidNotVote)
        // The voter did not reveal or did not commit.  Slash at noVote rate.
        slash = -int256(
            Math.ceilDiv(
                effectiveStake * trackers.noVoteSlashPerToken,
                1e18
            )
        );

    // [...]
}
```

Impact

If the `effectiveStake` is larger than the `maxTokenPerVote`, the voter may receive more voting tokens or may have more voting tokens slashed. Meanwhile, since the `totalVotes` and `totalStaked` are calculated using the capped effective stake, uncapped effective stake in the function `_updateAccountSlashingTrackers` may also lead to inconsistencies in internal accounting.

Recommendations

Consider updating the calculation of the `effectiveStake` in the function `_updateAccountSlashingTrackers` to the following code:

```
uint256 effectiveStake = Math.min(
    voterStake.stake - voterStake.pendingStakes[trackers.lastVotingRound],
    maxTokenPerVote
);
```

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `a20527a0` ↗.

3.8. Voters can register or deregister during the voter-selection period

Target	VoterRegistrar		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The owner of the VotingV2 contract can request to select voters for each round. This will send requests to the Chainlink VRF coordinator, and then the coordinator will set the voters with the random words through the `fulfillRandomWords` function.

The selection of voters mainly depends on the random words and the total number of voters. During the request-pending period, voters can still register and deregister, and voter registration and deregistration will change the total number of voters. This means that voter registration and deregistration actions can slightly alter the result of the voter selection.

```
function fulfillRandomWords(
    uint256 requestId,
    uint256[] calldata _randomWords
) internal override {
    // [...]
    for (uint256 i = 0; i < _randomWords.length; i++) {
        (uint256 randomVoterSerialNum, address voter) = _pickVoter(
            _randomWords[i],
            0
        );
    // [...]
}

function _pickVoter(
    uint256 randomWord,
    uint256 retryCount
) internal view returns (uint256, address) {
    return voters.get((randomWord + retryCount) % voters.length());
}
```

Impact

While the transaction for `fulfillRandomWords` is pending, voters can know the selection result. They can front-run the transaction to alter the result by registering or deregistering.

Note that the transaction will be reverted when the `voters.length()` is zero. Since the transaction is reverted, the function `votersPickedFailureCallback` will not be called to clear the request record. If the request record is not cleared, a new request cannot be initiated, and the coordinator will not fulfill the same request again.

```
function fulfillRandomWords(
    uint256 requestId,
    uint256[] calldata _randomWords
) internal override {
    // [...]

        if (randomVoterSerialNum == 0) {
            // send callback
            VoterRegistrarCallbackRecipientInterface(
                requests[uint256(requestId)].callbackRecipentAddress
            ).votersPickedFailureCallback(uint256(requestId));

            emit RequestFullfilled(
                uint256(requestId),
                randomVoterAddresses,
                false
            );
            return;
        }

    // [...]
}
```

Recommendations

Consider prohibiting registration or deregistration of voters while random voter requests are pending.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit a2b3364b ↗.

3.9. The owner may use stakers' staked tokens as rewards

Target	Staker		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The `notifyRewardAmount` function can allocate a specified amount of voting tokens already in the contract as a reward for the staked voters. And the voters can withdraw rewards through the `withdrawRewards` function.

Apart from the voting tokens used as rewards, the contract also holds voting tokens staked by voters. Since the total reward `rewardRate * rewardsDuration` is compared to the voting token balance of the contract, it is possible for the owner to notify staked voting tokens as rewards.

```
function notifyRewardAmount(uint256 _amount) external onlyOwner {
    _updateReward(address(0));

    // [...]

    require(rewardRate * rewardsDuration
    <= votingToken.balanceOf(address(this)), "Reward amount > balance");

    emit RewardAdded(_amount);
}
```

Impact

If a portion of the staked voting tokens is mistakenly notified as rewards and voters have withdrawn some of these rewards, the voters might be unable to unstake due to an insufficient amount of voting tokens in the contract.

Recommendations

Consider comparing the total reward `rewardRate * rewardsDuration` to `votingToken.balanceOf(address(this)) - cumulativeStake`.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit 15111134 ↗.

3.10. Picking a voter may fail

Target	VoterRegistrar, DynamicArray		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `_pickVoter` uses `(randomWord + retryCount) % voters.length()` to retrieve a specified voter.

```
function _pickVoter(
    uint256 randomWord,
    uint256 retryCount
) internal view returns (uint256, address) {
    return voters.get((randomWord + retryCount) % voters.length());
}
```

However, the function `get` in the library DynamicArray requires an ID greater than zero, while the range of `(randomWord + retryCount) % voters.length()` is [0, `voters.length() - 1`].

```
function get(
    Data storage self,
    uint256 _id
) internal view returns (uint256 id, address data) {
    require(_id > 0 && _id <= self.count, "Invalid ID");
    uint256 index = self.idToIndex[_id];
    Item memory item = self.items[index];
    return (item.id, item.data);
}
```

Impact

The transaction may be reverted due to an invalid ID error.

Recommendations

Consider using the index to retrieve the voter.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `72635a9d` ↗.

3.11. The re-picked `randomVoterSerialNum` may still be duplicated

Target	VoterRegistrar		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

In the contract VoterRegistrar, the function `fulfillRandomWords` will randomly select voters one by one based on the Chainlink VRF result. If it finds a duplicate, it will reroll once. However, the result after the reroll may still be a duplicate.

```
function fulfillRandomWords(
    uint256 requestId,
    uint256[] calldata _randomWords
) internal override {
    // [...]

    for (uint256 i = 0; i < _randomWords.length; i++) {
        (uint256 randomVoterSerialNum, address voter) = _pickVoter(
            _randomWords[i],
            0
        );

        // reroll if found duplicate
        if (
            randomVoterSerialNumMap[uint256(requestId)][
                randomVoterSerialNum
            ]
        ) {
            (randomVoterSerialNum, voter) = _pickVoter(_randomWords[i], 1);
        }

    // [...]
}

function _pickVoter(
    uint256 randomWord,
    uint256 retryCount
) internal view returns (uint256, address) {
    return voters.get((randomWord + retryCount) % voters.length());
}
```

For example, there are four voters, and we want to pick three voters. Assume the `_randomWords` is [2, 1, 1], and the `voters.get` function uses the index to retrieve the voter in the array. Then, `voters.get(2)` and `voters.get(1)` will become the first and the second voter, without duplication. Next, the `fulfillRandomWords` function will pick `voters.get(1)`. Since it finds the picked voter duplicate, it will then pick `voters.get(2)`, but this is still a duplicate.

Impact

The number of voters selected in a round may be smaller than expected.

Recommendations

Consider rerolling and increasing the `retryCount` until there is no duplication.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `d1e1d26e` ↗.

3.12. Typographical error in `setThresholds` function causes no-op

Target	VotingV2		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

In the VotingV2 contract, the `setThresholds()` function is designed to allow the contract owner to update various threshold values, including `MaxTokenPerVote`. However, due to a coding mistake, the function does not correctly update the `MaxTokenPerVote` variable. Instead, it performs a no-op by assigning the parameter `newMaxTokenPerVote` to itself, leaving the state variable `MaxTokenPerVote` unchanged.

```
function setThresholds(
    // [..]
    uint128 newMaxTokenPerVote
) public override onlyOwner {
    // [..]
    newMaxTokenPerVote = newMaxTokenPerVote; // This is a no-op and does
    nothing
}
```

Impact

The `maxTokenPerVote` value cannot be updated and remains zero. Since the `maxTokenPerVote` value is zero, the voter cannot unstake because the `staked` value will always be greater than or equal to zero.

```
function _afterRequestUnstake(
    address voter,
    uint256 staked
) internal override {
    require(!voterInfo[getCurrentRoundId()].voters[voter], "selected");
    if (staked < maxTokenPerVote) {
        voterRegistrar.unregisterVoter(voter);
    }
}
```

Recommendations

Consider correcting the assignment in the `setThresholds()` function to properly update the `Max-TokenPerVote` state variable with the new value provided by the function parameter.

```
function setThresholds(
    // [..]
    uint128 newMaxTokenPerVote
) public override onlyOwner {
    // [..]
    newMaxTokenPerVote = newMaxTokenPerVote;
    maxTokenPerVote = newMaxTokenPerVote;
}
```

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit 72635a9d ↗.

3.13. Incorrect handling of ID and count

Target	DynamicArray		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The DynamicArray library uses an `idToIndex` mapping to retrieve items by their IDs. However, the item ID is based on a counter that decreases when an item is removed, potentially leading to duplicate IDs. Since `self.count` decreases upon removal, an ID may become greater than the current count. Consequently, the check `require(_id > 0 && _id <= self.count, "Invalid ID")` in the `get()` and `remove()` functions may not accurately validate the IDs.

Here is proof-of-concept code of this issue:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import "../contracts/libraries/DynamicArray.sol";

contract DynamicArrayTest is Test {
    using DynamicArray for DynamicArray.Data;
    DynamicArray.Data public arrayData;

    function testReAdd() public {
        require(arrayData.add(address(0x1337)) != 0);
        arrayData.remove(1);

        // Add failed
        require(arrayData.add(address(0x1337)) == 0);
    }

    function testDuplicateIds() public {
        arrayData.add(address(0x11));
        arrayData.add(address(0x22));

        // remove the first item, so the self.count back to 1
        arrayData.remove(1);

        require(arrayData.idToIndex[2] == 0);
```

```
            // This will overwrite the idToIndex[2] to 1
            arrayData.add(address(0x33));
            require(arrayData.idToIndex[2] == 1);
        }
    }
```

Impact

Duplicate IDs can cause incorrect retrieval or removal of items, leading to data inconsistencies within the contract.

Recommendations

Consider using the index to retrieve the item.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit 72635a9d ↗.

3.14. The `settlementResolution` may be incorrectly set when no one disputes

Target	OptimisticOracleV3		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `settlementResolution` records the resolution of the assertion. If the assertion was disputed and configured to discard the oracle, the resolution should be false.

```
struct Assertion {
    // [...]
    bool settlementResolution; // Resolution of the assertion (false till
    resolved).
    // [...]
}
```

During the settlement of an assertion, if no one disputes, the `settlementResolution` will be set to true. However, if the `assertionPolicy.discardOracle` is true, it will be treated as a dispute. And in this case, the `settlementResolution` should be set to false.

```
function settleAssertion(bytes32 assertionId) public nonReentrant {
    Assertion storage assertion = assertions[assertionId];
    // [...]
    EscalationManagerInterface.AssertionPolicy memory assertionPolicy
    = _getAssertionPolicy(assertionId);
    if (assertion.disputer == address(0)) {
        // No dispute, settle with the asserter
        require(assertion.expirationTime <= getCurrentTime(), "Assertion not
    expired"); // Revert if not expired.
       assertion.settlementResolution = true;
        assertion.currency.safeTransfer(assertion.asserter, assertion.bond);
        if (!assertionPolicy.discardOracle) {
            _callbackOnAssertionResolve(assertionId, true);
            emit AssertionSettled(assertionId, assertion.asserter, false,
    true, msg.sender);
        } else {
           // discard as "dispute"
           _callbackOnAssertionResolve(assertionId, false);
```

```
                emit AssertionSettled(assertionId, assertion.asserter, true,
        false, msg.sender);
            }
        }
        // [...]
}
```

Impact

This could potentially affect off-chain programs and cause confusion.

Recommendations

Consider setting the value of `assertion.settlementResolution` according to the value of `assertionPolicy.discardOracle`.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `45b0aaf7` ↗.

3.15. Centralized risk

Target	VotingV2		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

There is a privileged owner for the contract:

1. The owner has significant control over the voting process and can manipulate voter selection via `emergencyJuryPicking()`.
2. The owner can prevent voters from being selected by not calling `requestVoterSet()`, effectively halting the voting process.
3. The owner can change the slashing library through `setSlashingLibrary()`.
4. The owner can change the maximum number of rounds to roll via `setMaxRolls()`.

Impact

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system.

Recommendations

We recommend clearly documenting this centralized design to inform users about the owner's control over the contract. This transparency enables users to make informed decisions about their participation in the project.

Additionally, outlining the specific circumstances under which the owner may exercise these powers can build trust and enhance transparency, and we also suggest implementing minimum and maximum values for each setter function that restrict governance changes to a safe range.

Remediation

This issue has been acknowledged by Opinion Labs.

Opinion Labs introduced changes to address this issue in commit `83ae9839` ↗. However, we believe that these changes do not fully mitigate the risk, as the admin can still assign the operator role to themselves.

To address this, potential solutions could include using a multi-signature wallet, restricting admin functions, limiting the protocol parameters, or redesigning the architecture. However, as this finding is evaluated as informational, centralized control may be considered a deliberate design choice rather than a critical issue.

Opinion Labs states that they will use a multi-signature wallet for admin.

3.16. Unused variables and functions

Target	VoterRegistrar, Staker		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is an unused variable `maxRetries` in the contract VoterRegistrar.

```
uint16 public maxRetries = 10;
```

There are two unused functions, `setDelegate` and `setDelegator`, in the contract Staker, whose codes are commented out.

```
function setDelegate(address delegate) external {
    // voterStakes[msg.sender].delegate = delegate;
    // emit DelegateSet(msg.sender, delegate);
}

function setDelegator(address delegator) external {
    // delegateToStaker[msg.sender] = delegator;
    // emit DelegatorSet(msg.sender, delegator);
}
```

Impact

This may compromise code readability and could consume more gas fees.

Recommendations

Consider removing the unused functions and removing or integrating the unused variables into the contract logic to ensure consistency and avoid potential confusion.

Remediation

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit `d1e1d26e` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Ambiguous functional behavior

In the DynamicArray contract, the `contains` function will return true when `_data` is not contained, which seems counterintuitive.

```
function contains(
    Data storage self,
    address _data
) public view returns (bool) {
    return self.contains[_data] == 0;
}
```

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit 72635a9d ↗.

4.2. Typographical errors

Upon reviewing the codebase, several minor typographical errors were identified. They do not affect code functionality but can lead to confusion and potential bugs.

In the contract VoterRegistrar,

- The event `RequestFullfilled` should be `RequestFulfilled`.
- The variable `callbackRecipentAddress` should be `callbackRecipientAddress`.
- The function parameter `callbackRecipent` should be `callbackRecipient`.

This issue has been acknowledged by Opinion Labs, and a fix was implemented in commit e0c38969 ↗.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: ConditionalTokenSettler.sol

Function: `assertPredictionMarket(byte[32] questionId, uint256 outcome)`

This function is called by the operator to assert the outcome of a prediction market's question.

Inputs

- `questionId`
  - **Control**: Fully controlled by the caller with an `OPERATOR_ROLE` role.
  - **Constraints**: None.
  - **Impact**: Identifies the prediction market question for which the outcome is being asserted.
- `outcome`
  - **Control**: Fully controlled by the caller with an `OPERATOR_ROLE` role.
  - **Constraints**: None.
  - **Impact**: Represents the outcome being asserted for the question.

Branches and code coverage

Intended branches

- Create a new assertion.
  - ☐ Test coverage
- Transfer the required bond from the operator to the contract using the default currency and bond amount obtained from the oracle.
  - ☐ Test coverage
- Assert the outcome, passing the encoded condition ID and outcome along with other necessary parameters.
  - ☐ Test coverage
- Update the assertion records with the new `assertionId`.
  - ☐ Test coverage

Negative behavior

- Revert if the question does not exist.

- ☐ Negative test
- Revert if `outcome` is not `1` or `2`.
  - ☐ Negative test
- Revert if a previous assertion exists for the `questionId` and it is not finalized.
  - ☐ Negative test

5.2. Module: VoterRegistrar.sol

Function: `getRequest(uint256 _requestId)`

This function allows anyone to retrieve the details of the request, including whether it has been fulfilled and the list of voters were selected.

Inputs

- `_requestId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: An ID to retrieve request information.

Branches and code coverage

Intended branches

- Return the `fulfilled` status and `randomVoters` array for the given `_requestId`.
  - ☐ Test coverage

Negative behavior

- Revert if the `_requestId` does not correspond to an existing request.
  - ☐ Negative test

Function: `requestRandomVoters(uint32 totalRandomVoters, uint32 callbackGasLimit, address callbackRecipent)`

This function allows a requester to initiate a randomness request to select a specified number of random voters. It interacts with the Chainlink VRF to obtain randomness.

Inputs

- `totalRandomVoters`
  - **Control**: Fully controlled by the caller with a `REQUESTER_ROLE` role.
  - **Constraints**: None.

    - **Impact**: The number of voters that will be selected.
- `callbackGasLimit`
    - **Control**: Fully controlled by the caller with a `REQUESTER_ROLE` role.
    - **Constraints**: None.
    - **Impact**: The gas limit for the VRF callback function.
- `callbackRecipent`
    - **Control**: Fully controlled by the caller with a `REQUESTER_ROLE` role.
    - **Constraints**: None.
    - **Impact**: The address that will receive the callback once random voters are selected.

Branches and code coverage

Intended branches

- Initiate the VRF request to select a specified number of random voters and return the `requestId` of the VRF request.
    - ☐ Test coverage

Negative behavior

- Revert if the caller does not have the `REQUESTER_ROLE`.
    - ☐ Negative test
- Revert if `totalRandomVoters` is greater than the number of registered voters.
    - ☐ Negative test

5.3. Module: VotingV2.sol

Function: `commitVote(byte[32] identifier, uint256 time, bytes ancillaryData, byte[32] hash)`

This function allows a selected voter to commit a vote for a specific price request during the commit phase.

Inputs

- `identifier`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The price request's identifier that the voter is committing a vote for.
- `time`
    - **Control**: Arbitrary.
    - **Constraints**: None.

    - **Impact**: The Unix timestamp of the price being voted on.
- `ancillaryData`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Arbitrary data appended to a price request to give the voters more info from the caller.
- `hash`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The `keccak256` hash of the price, salt, voter address, time, `ancillaryData`, current `roundId`, and identifier.

Branches and code coverage

Intended branches

- Commit the vote hash for the voter if all conditions are met.
    - ☐ Test coverage

Negative behavior

- Revert if `hash` is zero.
    - ☐ Negative test
- Revert if the current phase is not the commit phase.
    - ☐ Negative test
- Revert if the voter is not selected for the current round.
    - ☐ Negative test
- Revert if the price request is not active.
    - ☐ Negative test

Function: `emergencyJuryPicking(uint32 roundId, address[] jury)`

This function allows the owner to manually set the voter set for a specific round in case of an emergency or if the automatic voter selection fails.

Inputs

- `roundId`
    - **Control**: Fully controlled by the owner.
    - **Constraints**: None.
    - **Impact**: The voting round for which the voter set is being manually set.
- `jury`
    - **Control**: Fully controlled by the owner.

- **Constraints**: None.
- **Impact**: The list of voters to be set for the specified round.

Branches and code coverage

Intended branches

- Set the voter set for the specified round using `_setVoter`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☐ Negative test

Function: `requestVoterSet(uint32 totalRandomVoters, uint32 callbackGasLimit)`

This function allows the owner to request a set of random voters for the current voting round from VoterRegistrar.

Inputs

- `totalRandomVoters`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The number of random voters to request.
- `callbackGasLimit`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The gas limit for the callback function in VoterRegistrar.

Branches and code coverage

Intended branches

- Request random voters if not already requested for the current round.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the `owner`.
  - ☐ Negative test
- Revert if a voter set has already been requested for the current round.

- ☐ Negative test

Function: `votersPickedSuccessCallback(uint256 requestId, address[] voters)`

This function is a callback invoked by VoterRegistrar when random voters have been successfully selected.

Inputs

- `requestId`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: the voter request for which voters have been selected.
- `voters`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The list of randomly selected voters for the specified request.

Branches and code coverage

Intended branches

- Updates the voter set for the associated round using `_setVoter`.
  - ☐ Test coverage

Function call analysis

- `_setVoter(voterRequest[requestId], voters)`
  - **What is controllable?** `requestId` and `voters`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the `voterInfo` for the specified round with the selected voters — no return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Base Mainnet.

During our assessment on the scoped Prediction Market contracts, we discovered 16 findings. One critical issue was found. Three were of high impact, nine were of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.