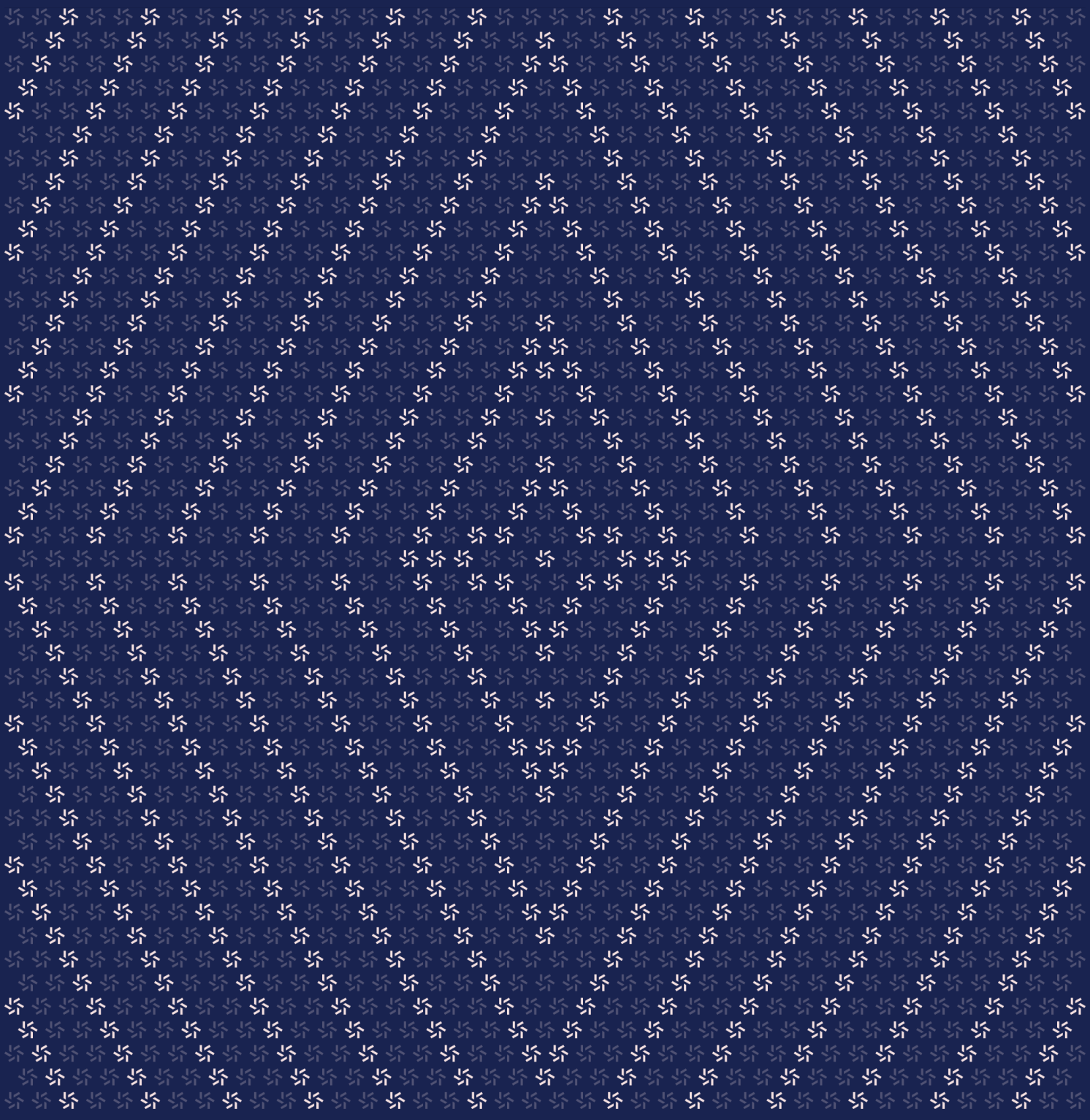
Zellic

Prepared for
Kevin Choi
Spectral Finance

Prepared by
Vlad Toie
Yuhang Wu
Zellic

November 24, 2023

Spectral Modelers
Smart Contract Security Assessment



Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Spectral Finance from November 6th to November 27th, 2023. During this engagement, Zellic reviewed Spectral Modelers's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. 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:

- Are there any bugs in the upgradability section of the contracts?
- Could any modeler or validator lose funds due to a bug in the staking or reward distribution logic?
- Could any modeler abuse the system to gain an unfair advantage?
- Does the competition logic correctly enforce the rules of the competition?
- Could anyone break the fairness of the competition?

1.2. Non-goals and Limitations

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

- Web2-related elements
- Front-end components
- Infrastructure relating to the project
- Key custody

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

During this assessment, the lack of testing prevented us from dynamically analyzing the protocol and ensuring that all its components work as they should. As can be seen in section (3. ↗), most of the issues that have been identified stem from a lack of testing that would cover those particular cases. Moreover, as there was no up-to-date testing environment, we were unable to thoroughly test the protocol's upgradeability. The absence of this testing, combined with the other issues outlined in this report, suggests that the protocol is not yet ready for production, and requires additional testing and verification.

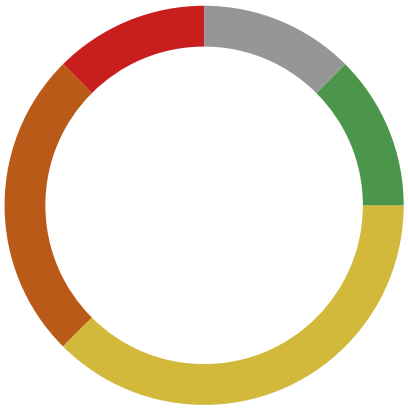
1.3. Results

During our assessment on the scoped Spectral Modelers contracts, we discovered eight findings. One critical issue was found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Spectral Finance's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	2
Medium	3
Low	1
Informational	1



2. Introduction

2.1. About Spectral Modelers

Spectral Modelers is a trustless solver network leveraging zero-knowledge machine learning to guarantee integrity and quality of machine-learning models that solve real-world problems.

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:

Spectral Modelers Contracts

Repository	https://github.com/Spectral-Finance/sc-poc ↗
Version	sc-poc: `41f64855bf0e8f446d5d37aabe5e1f56f9f6b6a6`
Programs	• CompetitionFactory.sol • Modeler.sol • ServiceAgreement.sol • Competition.sol
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Vlad Toie
Engineer
vlad@zellic.io ↗

Yuhang Wu
Engineer
yuhang@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 6, 2023	Kick-off call
November 6, 2023	Start of primary review period
November 27, 2023	End of primary review period

3. Detailed Findings

3.1. Anyone can become a validator

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

Description

The `registerValidator` function facilitates the registration of validators, a privileged role within the Modeler contract.

```
function registerValidator() public whenNotPaused {
    // commented the library call for ease of reading
    //ModelerLibrary.registerValidator(
    // validatorAddresses,
    // maxValidators,
    // validatorToken,
    // validators[msg.sender],
    // validatorStakeAmount,
    // isValidatorRegistered
    //);

    require(validatorAddresses.length < MAX_VALIDATORS, "Max validators
    reached");
    require(isValRegistered[msg.sender] == false || (validator.optOutTime >
    0 && validator.optOutTime < block.timestamp - ONE_DAY) , "Validator
    already registered");
    validatorToken.safeTransferFrom(msg.sender, address(this),
    validatorStakeAmount);
    isValRegistered[msg.sender] = true;
    validator.currentStake = validatorStakeAmount;
    validator.optOutTime = ~uint256(0);
    addValidatorRewardList(msg.sender, validatorAddresses,
    isValRegistered);
}
```

The function checks whether the `validatorAddresses.length < MAX_VALIDATORS` and assures that the required amount of validator tokens is paid for by the to-be validator (i.e., `msg.sender`). It then sets the value of `isValRegistered` (a storage reference of `isValidatorRegistered`) to true, effectively giving the validator role to the `msg.sender`.

Technically all the checks so far are performing their intended role. The `MAX_VALIDATORS` would

be set to 1 in the current version of the codebase, as stated by the Spectral team, which means that only the first to ever call the `registerValidator` will become validator.

Upon closer examination of the `addValidatorRewardList` function (last line in the `registerValidator` definition), we observe the following issue:

```
function addValidatorRewardList(address _validator, address[]
    storage validatorAddresses, mapping(address => bool)
    storage isValRegistered) internal {
    if (isValRegistered[_validator]) {
        return;
    }
    validatorAddresses.push(_validator);
}
```

Note that before calling `addValidatorRewardList`, the storage was previously updated `isValRegistered[msg.sender] = true;`. Thus, the `if (isValRegistered[_validator])` will always hold true, and therefore the `validatorAddresses.push(_validator);` will never be called. This, in turn, means that the previous `require(validatorAddresses.length < MAX_VALIDATORS, "Max validators reached");` check will always be redundant, allowing for an unlimited amount of users to call `registerValidator` with the only virtual constraint of affording the staking required to do so.

Impact

Anyone can call `registerValidator` and become a validator, the highest ranking role in the Modeler contract. The capabilities of a validator include the removal of other validators or modelers at will, changing the performance rating of modelers, overwriting existing challenges, and so forth.

Recommendations

We recommend moving the `addValidatorRewardList` before the `isValRegistered` storage change. For additional assurance, we highly recommend drastically limiting the access to this function, by means of setting an admin that would oversee the registration of validators.

We consider that this particular issue could have been readily identified with a basic testing suite and therefore would encourage the team to set up a comprehensive testing suite to help mitigate against similar issues in the future.

Remediation

The Spectral team has mitigated this issue in commit 2b6f0b9e ↗ by performing a `push` operation into the `validatorAddresses` straight away, rather than checking whether the value is already registered as a validator or not. This solution does address the issue, on the assumption that the

`require(validatorAddresses.length < MAX_VALIDATORS, “Max validators reached”);` holds. No additional access control modifiers have been added to this function.

3.2. Storage not set properly

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

Description

A ZKMLChallenge is one type of challenge that the Spectral protocol supports. The modelers are required to reply to a challenge, by essentially answering with their model's response. The `respondToZKMLChallenge` function thus allows the modeler to respond to a ZKMLChallenge by providing the IPFS response where their model's response has been uploaded.

```
function respondToZKMLChallenge(address _validator,
    string calldata _ipfsResponse) external {
    require(modelers[msg.sender].modelerSubmissionBlock != 0, "M not
    registered");
    require(isValidatorRegistered[_validator], "V not registered");
    DataTypes.ModelerChallenge memory zc = ZKMLChallenges[msg.sender];
    require(bytes(zc.ipfsChallenge).length > 0, "Challenge not found");
    zc.ipfsResponse = _ipfsResponse; // @audit-issue this is memory!!!
    storage won't update properly!!!!
    emit ZKMLChallengeResponse(_validator, msg.sender, _ipfsResponse);
}
```

The function assures that the `msg.sender` is a registered modeler and would then store the user's response. The issue arises, however, due to the fact that the `ZKMLCHallenges[msg.sender]` object is referenced by memory instead of storage, so any update to it will not be persistent after the function call ends.

Impact

The user's `ipfsResponse` for the `ZKML` challenges will never actually be stored.

Recommendations

Change the type of `zc` to `storage` instead of `memory` to ensure that the changes are persistent.

```
function respondToZKMLChallenge(address _validator,
    string calldata _ipfsResponse) external {
    // ...
    DataTypes.ModelerChallenge memory zc = ZKMLChallenges[msg.sender];
    DataTypes.ModelerChallenge storage zc = ZKMLChallenges[msg.sender];
    // ...
}
```

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit dd51d097 ↗.

3.3. Out-of-bounds access

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

Description

The Competition contract offers a way for the protocol to define performance metrics for the modelers based on their accuracy and results in previous contests. One of those performance metrics is reflected in the "top N Modelers" leaderboard, where the top performing modelers are displayed.

The `replaceOptOutTopNModeler` function allows the removal of a modeler from this leaderboard, under specific circumstances decided by the validators. The function is defined below:

```
function replaceOptOutTopNModeler(uint256 _oldModelerIndex,
    address _newModeler, uint256 _medianPerformanceResults)
    external onlyModelerContract {
    require(_oldModelerIndex < topNModelers.length, "ERR4");
    address _oldModeler = topNModelers[_oldModelerIndex];
    uint256 newIndex = findModelerUpperBound(_medianPerformanceResults);
    for (uint256 i = topNModelers.length; i > newIndex; i--) {
        topNModelers[i] = topNModelers[i - 1];
    }
    topNModelers[newIndex] = _newModeler;
    modelerToMedian[_newModeler] = _medianPerformanceResults;
    delete modelerToMedian[_oldModeler];
    emit TopNModelersUpdated(topNModelers);
}
```

As can be seen, the for loop tries accessing the `topNModelers[topNModelers.length]`, which will revert, as Solidity is a zero-indexed programming language. On top of that, the modeler's index will be overwritten and not removed altogether.

Impact

Even though the code will compile, it will always revert because out-of-bounds accesses are handled by Solidity as exceptions.

Recommendations

We recommend assuring that the start of the for loop is at `topNModelers.length - 1`.

```
function replaceOptOutTopNModeler(uint256 _oldModelerIndex,
    address _newModeler, uint256 _medianPerformanceResults)
    external onlyModelerContract {
    require(_oldModelerIndex < topNModelers.length, "ERR4");
    address _oldModeler = topNModelers[_oldModelerIndex];
    uint256 newIndex = findModelerUpperBound(_medianPerformanceResults);
    for (uint256 i = topNModelers.length - 1; i > newIndex; i--) {
        topNModelers[i] = topNModelers[i - 1];
    }
    topNModelers[newIndex] = _newModeler;
    modelerToMedian[_newModeler] = _medianPerformanceResults;
    delete modelerToMedian[_oldModeler];
    emit TopNModelersUpdated(topNModelers);
}
```

Moreover, we recommend covering for this particular function in the testing suite and assuring that no such similar issues are to occur in the future development of the protocol.

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit `17d3d47a` ↗.

3.4. Inaccurate handling of identical scores in `findModelerUpperBound`

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

Description

The `findModelerUpperBound` function in the CompetitionLibrary contract is intended to find the upper bound index for a given `element` in a sorted array using binary search. However, the current implementation does not correctly handle scenarios where multiple identical scores exist in the `topNModelers`. Specifically, the check `modelerToMedian[topNModelers[low - 1]] == element` only considers the immediate preceding score and does not account for multiple identical scores.

```
function findModelerUpperBound(address[] storage topNModelers,
    mapping(address => uint256) storage modelerToMedian, uint256 element)
    external view returns (uint256) {
        if (topNModelers.length == 0) {
            return 0;
        }
        uint256 low = 0;
        uint256 high = topNModelers.length;
        while (low < high) {
            uint256 mid = (low + high) / 2;
            uint256 midValue = modelerToMedian[topNModelers[mid]];
            if (element > midValue) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }
        if (low > 0 && modelerToMedian[topNModelers[low - 1]] == element) {
            return low - 1;
        } else {
            return low;
        }
    }
```

Impact

This issue can result in inaccurate positioning of modelers within the `topNModelers` list, especially in scenarios with tied scores. It could lead to unfair ranking and potentially affect the outcome of competitions or distributions based on these rankings.

Recommendations

A more robust implementation should be developed to handle identical scores appropriately. The binary search logic should be modified to find the first index that is strictly less than the `element` and correctly position modelers with identical scores.

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit `fbe68d35` ↗.

3.5. Storage gaps improperly defined

Target	Codebase		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Multiple contracts across the codebase are theoretically upgradable. All upgradable contracts require defining the gap array that allows allocating the storage safely in the future upgrades of the contracts, therefore eliminating the risk of storage collisions.

For example, the Competition contract defines the following storage persistent variables.

```
contract Modeler is IModeler, Initializable, PausableUpgradeable  {
    using SafeERC20 for IERC20;
    uint256 private maxValidators;
    IERC20 private validatorToken;
    uint256 private validatorStakeAmount;

    mapping(address => DataTypes.ValidatorData) public validators;
    mapping(address => DataTypes.ModelerData) public modelers;

    mapping(address => bool) public isValidatorRegistered;

    mapping(address => DataTypes.ModelerChallenge) public modelerChallenges;

    mapping(address => DataTypes.ModelerChallenge) public ZKMLChallenges;
    mapping(address => mapping (address => uint256))
    public futureRandSlots; // validator => modeler => randSlot
    mapping (uint256 => DataTypes.DrandProof) public drands; // blockNumber
    => DrandProof

    address[] public validatorAddresses;
    address[] public modelerAddresses;

    string public ipfsTestingDataset;

    address private competitionContract;
```

Despite all of the variables occupying more than one storage slot, the `gap` is set to 50, the default allocation as per the standard.

Instead, the storage gaps should be set up depending on how many actual slots are occupied. the Modeler contract uses 13 slots, so the gap would need to be defined as `gap[50 - 13] = gap[37]`.

Impact

Using the storage gap in an incorrect way may lead to storage collisions down the line, which could severely hamper the security of the protocol as a whole.

Recommendations

We recommend properly defining the storage gaps to match the actual slots currently in use for each contract. The `forge inspect` can be used to see the amount of storage that is currently in use. More can be read about using `forge` here ↗. Moreover, to test the validity of storage gaps in case of an upgrade, read this post ↗.

Remediation

This issue has been acknowledged by Spectral Finance.

3.6. Incorrect removal logic in `removeValidatorRewardList`

Target	ModelerLibrary		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `removeValidatorRewardList` function is designed to remove a validator from the `validatorAddresses` array. However, the current implementation has a flaw. If the validator to be removed is not the last one in the array, the function incorrectly removes the last validator instead of the intended one.

```
function removeValidatorRewardList(address _validator, address[]
    storage validatorAddresses) internal {
    uint256 index = validatorAddresses.length + 1;
    for (uint256 i = 0; i < validatorAddresses.length; i++) {
        if (validatorAddresses[i] == _validator) {
            index = i;
            break;
        }
    }
    require(index < validatorAddresses.length, "Address not found");
    validatorAddresses[index]
    = validatorAddresses[validatorAddresses.length - 1];
    validatorAddresses.pop();
}
```

Impact

This issue can lead to unintended consequences, including the accidental removal of incorrect validators.

Recommendations

We recommend rewriting the removal logic to ensure that the correct validator is removed. The logic should identify the correct index of the validator to be removed and then proceed to replace it with the last validator before popping the last element.

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit `3f88388f` ↗.

3.7. Properties `ipfsChallenge` and `ipfsResponse` should be updated in `updateModel`

Target	Modeler		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `updateModel` function in the provided contract is intended to update the model commitment of a registered modeler. However, this function does not adequately handle the `ipfsChallenge` and `ipfsResponse` properties associated with the modeler. It only resets `ipfsGraded`, `performanceResult`, and `medianPerformanceResults` to default values. This incomplete handling could lead to inconsistencies in tracking and validating the modeler's challenges and responses over the blockchain.

```
function updateModel(string calldata _modelCommitment) external {
    // ...
    modelerChallenges[msg.sender].ipfsGraded = "0";
    modelerChallenges[msg.sender].performanceResult = 0;
    modelers[msg.sender].medianPerformanceResults = 0;
    // ...
}
```

Impact

The lack of proper management for `ipfsChallenge` and `ipfsResponse` can lead to potential mismatches or outdated information being stored on the blockchain. This might not significantly impact the system's functionality but could cause data integrity issues over time.

Recommendations

It is recommended to include the proper handling of `ipfsChallenge` and `ipfsResponse` within the `updateModel` function.

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit `fbe68d35` ↗.

3.8. Code does not compile

Target	BeaconProxyDeployer		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The BeaconProxyDeployer contract handles deploying new UpgradeableBeacon contracts for the modeler and competition implementations.

```
function _createCompetitionBeacon(address logic) internal {
    require(address(competitionBeacon) == address(0), "Competition Beacon is
    already set");
    competitionBeacon = new UpgradeableBeacon(
        logic,
    );
    emit CompetitionBeaconDeployed(competitionBeacon);
}

function _createModelerBeacon(address logic) internal {
    require(address(modelerBeacon) == address(0), "Competition Beacon is
    already set");
    modelerBeacon = new UpgradeableBeacon(
        logic, // implementation
        );
    emit ModelerBeaconDeployed(modelerBeacon);
}
```

The contract uses an older version of OpenZeppelin's UpgradeableBeacon and therefore misses an important upgrade made available in 0.5.0 that sets the initial owner in the constructor of the `UpgradeableBeacon`. Therefore, the `new UpgradeableBeacon` command misses the additional parameter for deploying the contract successfully in the latest version.

Impact

The contract cannot be deployed if the latest UpgradeableBeacon version is used.

Recommendations

We recommend using the latest stable and audited version of the UpgradeableBeacon contract, V5.0. That would imply adding the additional parameter for the `initialOwner`. More can be read about the latest version of the UpgradeableBeacon here ↗.

Remediation

This issue has been acknowledged by Spectral Finance.

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. Return of staked tokens

The function `kickModeler` merely kicks out the modeler but does not return the staked tokens to the modeler. If there are malicious validators, they could use this method to make many modelers lose funds. A similar function, `optOutModeler`, does have the operation of returning the staked tokens to the modeler.

```
function kickModeler(address modeler)
    external onlyValidator whenNotPaused {
        modelers[modeler].optOutTime = block.timestamp;
        emit ModelerKicked(msg.sender, modeler);
    }
```

The second issue is that `validator.currentStake` is directly assigned the value of `validatorStakeAmount` rather than `validator.currentStake += validatorStakeAmount`.

```
function registerValidator(address[] storage validatorAddresses,
    uint256 MAX_VALIDATORS, IERC20 validatorToken,
    DataTypes.ValidatorData storage validator,
    uint256 validatorStakeAmount, mapping(address => bool)
    storage isValRegistered) external {
        require(validatorAddresses.length < MAX_VALIDATORS, "Max validators
    reached");
        require(isValRegistered[msg.sender] == false ||
    (validator.optOutTime > 0 && validator.optOutTime < block.timestamp
    - ONE_DAY) , "Validator already registered");
        validatorToken.safeTransferFrom(msg.sender, address(this),
    validatorStakeAmount);
        isValRegistered[msg.sender] = true;
        validator.currentStake = validatorStakeAmount;
        validator.optOutTime = ~uint256(0);
        addValidatorRewardList(msg.sender, validatorAddresses,
    isValRegistered);
        emit ValidatorRegistered(msg.sender);
    }
```

In some cases, for example, if `validator.optOutTime` > 0 and the validator's staked tokens are not returned (in the case that the validator be kicked out with `emergencyOptOutValidator`), then these staked tokens could be permanently lost.

4.2. Insufficient testing

In the current project, a significant portion of the issues, including bugs, security vulnerabilities, and functional inconsistencies can be traced back to inadequate testing practices. We addressed such issues in the Threat Model section (5. ↗). Comprehensive testing is essential to ensure the correctness and security of the system. Either code coverage or negative testing is a good way to trigger bugs directly (e.g., the out-of-bound issue).

Moreover, the upgradeability of the protocol is not thoroughly tested. The current test suite does not extensively cover the upgradeability of the protocol, a component which is essential for the protocol to be production-ready. As the upgradeability of the protocol is a critical component, requiring a relatively high level of confidence, we recommend that the protocol be thoroughly tested in this regard.

4.3. Missing zero-address check in `setAdmin`

The `setAdmin` function in the contract allows for changing the admin address. However, the current implementation lacks a check to prevent the new admin address from being set to the zero address (0x0).

```
function setAdmin(address _newAdmin) external onlyAdmin {
    admin = _newAdmin;
}
```

Allowing the admin address to be set to the zero address can lead to accidental or intentional locking out of administrative control.

Recommendations

Consider including a check to ensure that `_newAdmin` is not the zero address.

```
function setAdmin(address _newAdmin) external onlyAdmin {
    require(_newAdmin != address(0), "Admin address cannot be zero");
    admin = _newAdmin;
}
```

Remediation

This issue has been acknowledged by Spectral Finance, and a fix was implemented in commit `fbe68d35` ↗.

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: BeaconProxyDeployer.sol

Function: `deployCompetitionBeaconProxy(address logic, bytes payload)`

This deploys a new competition beacon proxy.

Inputs

- `logic`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Must be a valid contract address.
  - **Impact**: Specifies the logic contract to be used by the beacon proxy.
- `payload`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Can be any valid byte array.
  - **Impact**: Used for initializing the beacon proxy.

Branches and code coverage

Intended branches

- Check if a competition beacon is already set and create one if not.
  - ☑ Test coverage

Negative behavior

- Revert if the competition beacon is already set.
  - ☐ Negative test
- Revert on failure to create the proxy instance (code size equal with zero).
  - ☐ Negative test

Function: `deployModelerBeaconProxy(address logic, bytes payload)`

This deploys a new modeler beacon proxy.

Inputs

- `logic`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Must be a valid contract address.
  - **Impact**: Specifies the logic contract to be used by the beacon proxy.
- `payload`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Can be any valid byte array.
  - **Impact**: Used for initializing the beacon proxy.

Branches and code coverage

Intended branches

- Check if a modeler beacon is already set, and create one if not.
  - ☑ Test coverage

Negative behavior

- Revert if the modeler beacon is already set.
  - ☐ Negative test
- Revert on failure to create the proxy instance (code size equal with zero).
  - ☐ Negative test

Function: `upgradeCompetitionBeacon(address newLogic)`

This upgrades the competition beacon to a new logic contract.

Inputs

- `newLogic`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Must be a valid contract address.
  - **Impact**: Determines the new logic contract that the beacon will point to.

Branches and code coverage

Intended branches

- Call to `upgradeTo` on the competition beacon.
  - ☑ Test coverage

Function: `upgradeModelerBeacon(address newLogic)`

This upgrades the modeler beacon to a new logic contract.

Inputs

- `newLogic`
  - **Control**: Determined by the caller of the function.
  - **Constraints**: Must be a valid contract address.
  - **Impact**: Determines the new logic contract that the beacon will point to.

Branches and code coverage

Intended branches

- Call to `upgradeTo` on the modeler beacon.
  - ☑ Test coverage

5.2. Module: CompetitionFactory.sol

Function: `constructor()`

This is the constructor for the CompetitionFactory contract.

Branches and code coverage

Intended branches

- Should disable initializers. Currently not performed.
  - ☐ Test coverage

Function: `deployCompetition(address _admin, string _ipfsCompetition, uint256 _topNParameter)`

This should deploy a new Competition contract.

Inputs

- `_admin`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.
  - **Impact**: Sets the admin variable of the Competition contract.

- `_ipfsCompetition`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be empty.
  - **Impact**: Sets the `ipfsCompetition` variable of the Competition contract.
- `_topNParameter`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero.
  - **Impact**: Sets the `topNParameter` variable of the Competition contract.

Branches and code coverage

Intended branches

- Check that `_ipfsCompetition` has not been used before. Currently not performed.
  - ☐ Test coverage
- Assure that the `initialize` function is called on the Competition contract.
  - ☑ Test coverage
- Append the new Competition contract to the `allCompetitions` array.
  - ☑ Test coverage
- Deploy the new Competition contract.
  - ☑ Test coverage

Negative behavior

- `_admin` cannot be zero address.
  - ☑ Negative test
- `_ipfsCompetition` cannot be empty.
  - ☑ Negative test
- `_topNParameter` cannot be zero.
  - ☑ Negative test
- Caller is the owner.
  - ☑ Negative test

Function: `deployModeler(address _competitionContract, IERC20 _validatorToken, uint256 _validatorStakeAmount, uint256 _maxValidators)`

This should deploy a new Modeler contract.

Inputs

- `_competitionContract`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.

    - **Impact**: Sets the `competitionContract` variable of the Modeler contract.
- `_validatorToken`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be zero address.
    - **Impact**: Sets the `validatorToken` variable of the Modeler contract.
- `_validatorStakeAmount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be zero.
    - **Impact**: Sets the `validatorStakeAmount` variable of the Modeler contract.
- `_maxValidators`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be zero.
    - **Impact**: Sets the `maxValidators` variable of the Modeler contract.

Branches and code coverage

Intended branches

- Assure that the `initialize` function is called on the Modeler contract. Currently not performed.
    - ☐ Test coverage
- Append the new Modeler contract to the `allModelers` array.
    - ☑ Test coverage
- Deploy the new Modeler contract.
    - ☑ Test coverage

Negative behavior

- `_competitionContract` cannot be zero address.
    - ☑ Negative test
- `_validatorToken` cannot be zero address.
    - ☑ Negative test
- `_maxValidators` cannot be zero.
    - ☑ Negative test
- Caller is the owner.
    - ☑ Negative test

Function: `initialize(ICompetition _competitionLogic, IModeler _modelerLogic)`

This initializes the CompetitionFactory contract.

Inputs

- `_competitionLogic`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.
  - **Impact**: Sets the `competitionLogic` variable.
- `_modelerLogic`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.
  - **Impact**: Sets the `modelerLogic` variable.

Branches and code coverage

Intended branches

- Set the `owner` variable. Currently not performed, and the owner will not be maintained over upgrades.
  - ☐ Test coverage
- Should call all underlying initializers.
  - ☑ Test coverage
- Set the `competitionLogic` variable.
  - ☑ Test coverage
- Set the `modelerLogic` variable.
  - ☑ Test coverage
- Set the `version` variable.
  - ☑ Test coverage
- Set the `competitionVersion` variable.
  - ☑ Test coverage
- Set the `modelerVersion` variable.
  - ☑ Test coverage

Negative behavior

- Should not be callable multiple times. Currently not performed.
  - ☑ Negative test

Function: `setNewAdmin(address _newAdmin)`

This facilitates changing the owner of the contract.

Inputs

- `_newAdmin`
  - **Control**: Fully controlled by the caller.

- **Constraints**: Cannot be zero address.
- **Impact**: Sets the owner variable.

Branches and code coverage

Intended branches

- Should be performed in two steps so that the new admin can accept the role. Currently not performed.
  - ☐ Test coverage
- Set the `owner` variable to the `_newAdmin` parameter.
  - ☑ Test coverage

Negative behavior

- Should never be callable by someone other than the owner.
  - ☑ Negative test

Function: `upgradeCompetition(ICompetition newCompetition)`

This facilitates upgrading the logic contract for the Competition contract.

Inputs

- `newCompetition`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.
  - **Impact**: Sets the `competitionLogic` variable.

Branches and code coverage

Intended branches

- Assure that the upgrade procedure did not overwrite storage. Currently not performed specifically.
  - ☐ Test coverage
- Assure that the newCompetition contract is initialized. Currently not explicitly performed.
  - ☐ Test coverage
- Should set the `competitionLogic` variable to the `newCompetition` parameter.
  - ☑ Test coverage
- Increment the `competitionVersion` variable.
  - ☑ Test coverage

Negative behavior

- Should not allow upgrading to a `newCompetition` that has not been deployed by the factory. Currently not performed.
  - ☐ Negative test
- Should not be callable by anyone other than the owner.
  - ☑ Negative test

Function: `upgradeModelerContract(IModeler newModelerContract)`

This facilitates upgrading the logic contract for the Modeler contract.

Inputs

- `newModelerContract`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be zero address.
  - **Impact**: Sets the modelerLogic variable.

Branches and code coverage

Intended branches

- Assure that the upgrade procedure did not overwrite storage. Currently not performed specifically.
  - ☐ Test coverage
- Assure that the newModelerContract contract is initialized. Currently not explicitly performed.
  - ☐ Test coverage
- Should set the `modelerLogic` variable to the `newModelerContract` parameter.
  - ☑ Test coverage
- Increment the `modelerVersion` variable.
  - ☑ Test coverage

Negative behavior

- Should not allow upgrading to a `newModelerContract` that has not been deployed by the factory. Currently not performed.
  - ☐ Negative test
- Should not be callable by anyone other than the owner.
  - ☑ Negative test

5.3. Module: Competition.sol

Function: `emergencyOptOutValidator(address _validator)`

This function allows an admin to emergency opt out for a validator.

Inputs

- `_validator`
  - **Control**: Fully controlled by the admin caller.
  - **Constraints**: Must be a registered validator's address.
  - **Impact**: The specified validator as opted out.

Branches and code coverage

Intended branches

- Check if `_validator` is a registered validator.
  - ☑ Test coverage
- Remove `_validator` from `validatorAddresses`.
  - ☑ Test coverage
- Update `optOutTime` for `_validator`.
  - ☑ Test coverage

Negative behavior

- Caller is a service admin.
  - ☑ Negative test
- `_validator` is not a registered validator.
  - ☑ Negative test
- Address removal in `removeValidatorRewardList` function if `_validator` is not found in `validatorAddresses`.
  - ☐ Negative test

Function: `getModelerToMedian(address _modeler)`

This retrieves the median performance result for a given modeler.

Inputs

- `_modeler`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.

- **Impact**: Determines which modeler's median result is being queried.

Branches and code coverage

Intended branches

- Retrieval of median result for a specific modeler.
  - ☑ Test coverage
- Include function calls.
  - ☐ Test coverage
- End sentences with periods.
  - ☐ Test coverage

Function: `replaceOptOutTopNModeler(uint256 _oldModelerIndex, address _newModeler, uint256 _medianPerformanceResults)`

This function replaces a modeler in the `topNModelers` array with a new modeler based on their performance results.

Inputs

- `_oldModelerIndex`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be less than the length of the `topNModelers` array.
  - **Impact**: Identifies the index of the modeler to be replaced in the `topNModelers` array.
- `_newModeler`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The new modeler to be inserted into the `topNModelers` array.
- `_medianPerformanceResults`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The new index position of the `_newModeler` in the `topNModelers` array.

Branches and code coverage

Intended branches

- Validation of `_oldModelerIndex`.
  - ☐ Test coverage

- Finding the new index for `_newModeler` based on `_medianPerformanceResults`.
  - ☐ Test coverage
- Reordering of the `topNModelers` array and updating the `modelerToMedian` mapping.
  - ☐ Test coverage

Negative behavior

- Revert if `_oldModelerIndex` is out of bounds.
  - ☐ Negative test

Function: `setConsumptionContract(IConsumption _consumptionContract)`

This function sets the consumption contract address in the calling contract.

Inputs

- `_consumptionContract`
  - **Control**: Controlled by the admin.
  - **Constraints**: Address must not be zero.
  - **Impact**: Updates the `consumptionContract` state variable with the provided address.

Branches and code coverage

Intended branches

- Updating the `consumptionContract` state variable.
  - ☑ Test coverage

Negative behavior

- Ensuring caller is an admin.
  - ☑ Negative test
- `_consumptionContract` must not be a zero address.
  - ☐ Negative test

Function: `setModelerNetPerformanceResultAndUpdate(address modeler, uint256 medianPerformanceResults)`

This function updates the performance results of a modeler in the system.

Inputs

- `modeler`

- **Control**: Controlled by the caller.
- **Constraints**: N/A.
- **Impact**: Identifies the modeler whose performance results are being updated.

- `medianPerformanceResults`
  - **Control**: Controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The modeler's position in the top modelers' list, affecting their visibility in the system.

Branches and code coverage

Intended branches

- Check if the `topNModelers` list is empty and add a new modeler if true.
  - ☑ Test coverage
- Insert new modeler if `topNModelers` has space.
  - ☑ Test coverage
- Replace the lowest modeler if the new modeler's performance is better.
  - ☑ Test coverage

Negative behavior

- Revert if the `modeler` is not in `modelerToMedian` mapping.
  - ☐ Test coverage

Function: `signUpToCompetition(string calldata _modelHash)`

This signs up for the competition.

Inputs

- `_modelHash`
  - **Control**: Controlled by the caller.
  - **Constraints**: Must be a non-empty string.
  - **Impact**: Represents the model commitment of the modeler in the competition.

Branches and code coverage

Intended branches

- Register the modeler in the `modelerContract`.

- ☑ Test coverage

Negative behavior

- Function should revert if `_modelHash` is an empty string.
  - ☐ Negative test
- Function should revert if the `modelerContract` address is not set.
  - ☐ Negative test
- Registering a modeler that is already registered.
  - ☐ Negative test
- Registering a `_modelHash` that has already been used.
  - ☐ Negative test

5.4. Module: Modeler.sol

Function: `ZKMLChallengeModeler(address modeler, string _ipfsChallenge, DataTypes.DrandProof _proof)`

This allows ZKML challenging a modeler.

Inputs

- `modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the modeler has a proper `futureRandSlots` mapping.
  - **Impact**: The modeler to be challenged.
- `_ipfsChallenge`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The IPFS of the challenge.
- `_proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the round of the proof matches the `futureRandSlots` mapping.
  - **Impact**: The proof for the specific round.

Branches and code coverage

Intended branches

- Should assure that the proof is valid for the modeler. Currently not performed.
  - ☐ Test coverage

- Update the `ZKMLChallenges` mapping for the modeler by setting the `ipfsChallenge` to the `_ipfsChallenge`.
  - ☑ Test coverage
- Assure that the round of the proof matches the `futureRandSlots` mapping.
  - ☑ Test coverage

Negative behavior

- Should not allow challenging the modeler if they have already been challenged (overriding the current challenge).
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☐ Negative test

Function: `addStakeToModeler(uint256 amount)`

This allows the addition of stake to a modeler.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount that the modeler is staking.

Branches and code coverage

Intended branches

- Assure that the modeler is registered.
  - ☑ Test coverage
- Perform the transfer of the `amount` from the caller to the contract.
  - ☑ Test coverage
- Increase the `currentStake` of the modeler by the `amount`.
  - ☑ Test coverage

Negative behavior

- Should not be callable when the contract is paused.
  - ☑ Negative test

Function: `addStakeToValidator(uint256 amount)`

This adds stake to a validator.

Inputs

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount that the validator is staking.

Branches and code coverage

Intended branches

- Assure that the validator is registered. Currently not performed.
  - ☐ Test coverage
- Perform the transfer of the `amount` from the caller to the contract.
  - ☑ Test coverage
- Increase the `currentStake` of the validator by the `amount`.
  - ☑ Test coverage

Negative behavior

- Should not be callable when the contract is paused.
  - ☑ Negative test

Function: `emergencyOptOutValidator(address _validator)`

This allows a validator to emergency opt out.

Inputs

- `_validator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the validator is registered.
  - **Impact**: The validator to be opted out.

Branches and code coverage

Intended branches

- Assure that the validator has not already opted out. Currently not performed.
  - ☐ Test coverage
- Assure that the validator can no longer perform any actions after opting out. Currently not performed.
  - ☐ Test coverage
- Assure that the validator is registered. Currently not performed.

- ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than a competition contract.
  - ☑ Negative test

Function: `giveChallengeToModeler(string _ipfsChallenge, address _modeler, DataTypes.DrandProof _proof)`

Facilitates challenging a modeler.

Inputs

- `_ipfsChallenge`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The IPFS of the challenge.
- `_modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The modeler to be challenged.
- `_proof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the round of the proof matches the `futureRandSlots` mapping.
  - **Impact**: The proof for the specific round.

Branches and code coverage

Intended branches

- Assure that the `modeler` is registered. Currently not performed.
  - ☐ Test coverage
- Should reset the `ipfsResponse` of the modeler. Currently not performed.
  - ☐ Test coverage
- Map the `ipfsChallenge` between a `validator` and a `modeler`. Currently not performed.
  - ☐ Test coverage
- Assure that the `proof` is valid for the challenged `modeler`. Currently not performed.
  - ☐ Test coverage
- Assumed that the challenge is given for this contract's specific competition. Currently not explicitly checked.

- ☐ Test coverage
- Set the `ipfsChallenge` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage

Negative behavior

- Should not be able to challenge a modeler that has opted out.
  - ☐ Negative test
- Should not be able to challenge a modeler that has already been challenged.
  - ☐ Negative test
- Should not allow a proof that has already been submitted. Currently not checked.
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☑ Negative test

Function: `kickModeler(address modeler)`

This allows a validator to kick a modeler.

Inputs

- `modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the modeler is registered.
  - **Impact**: The modeler to be kicked.

Branches and code coverage

Intended branches

- Assure that the modeler is registered. Currently not performed.
  - ☐ Test coverage
- Currently assume that the modeler's stake is transferred back to them. Currently not performed.
  - ☐ Test coverage
- Assure that the modeler has not already opted out. Currently not performed.
  - ☐ Test coverage
- Assure that the rest of the Modeler's data is reset. Currently not performed.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than a validator.
  - ☑ Negative test

- Should not allow calling this while the contract is paused.
  - ☑ Negative test

Function: `optInModeler()`

This allows a modeler to opt in.

Branches and code coverage

Intended branches

- Assure that the modeler is registered.
  - ☑ Test coverage
- Assure that the modeler has enough stake.
  - ☑ Test coverage
- Update the `optOutTime` of the modeler to the max `uint256`.
  - ☑ Test coverage

Negative behavior

- Should not allow a modeler to opt in if they have been kicked. Currently not checked.
  - ☐ Negative test
- Should not allow modeler to opt in if they have already opted in. Currently not checked.
  - ☐ Negative test

Function: `optInValidator()`

This allows a validator to opt in.

Branches and code coverage

- ☑ Set the `optOutTime` of the validator to the max `uint256`.
- ☑ Append the `validator` to the `validatorAddresses` array.

Negative behavior

- Should not allow a validator to opt in if they have been kicked. Currently not checked.
  - ☐ Negative test
- Should not allow a validator to opt in if they have already opted in.
  - ☐ Negative test
- Should not allow an unregistered validator to opt in.
  - ☑ Negative test
- Should not allow a validator to opt in if they do not have enough stake.
  - ☑ Negative test

- Should now allow a validator to opt in if they have already called `registerValidator`.
  - ☑ Negative test
- Should not allow calling this while the contract is paused.
  - ☑ Negative test

Function: `optOutModeler()`

This allows modelers to opt out.

Branches and code coverage

Intended branches

- Reset the `medianPerformanceResults` of the modeler. Currently not performed.
  - ☐ Test coverage
- Reset the `modelerSubmissionBlock` of the modeler. Currently not performed.
  - ☐ Test coverage
- Assure that the modeler is registered.
  - ☑ Test coverage
- Assure that the modeler has not already opted out.
  - ☑ Test coverage
- Assure that the modeler has not already begun the opt-out process.
  - ☑ Test coverage
- Update the `optOutTime` of the modeler to the current time plus one day.
  - ☑ Test coverage
- Transfer the modeler's stake back to them.
  - ☑ Test coverage

Negative behavior

- Should not allow a top N modeler to opt out. Currently not checked.
  - ☐ Negative test

Function: `optOutValidator()`

This allows validator to opt out.

Branches and code coverage

Intended branches

- Assure that there is at least one validator left. Currently not performed.
  - ☐ Test coverage
- Pop the validator from the `validatorAddresses` array. Performed in `removeValida-`

`torRewardList`.
  - ☑ Test coverage
- Assure that the validator has not already opted out.
  - ☑ Test coverage
- Assure that the validator has not already begun the opt-out process.
  - ☑ Test coverage
- Set the `optOutTime` of the validator to the current time plus one day.
  - ☑ Test coverage
- Reset the `currentStake` of the validator to zero.
  - ☑ Test coverage
- Transfer the validator's stake back to them.
  - ☑ Test coverage
- Remove the validator from the `validatorRewardList`.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than a validator to call this function.
  - ☑ Negative test

Function: `postGraded(address modeler, string _ipfsGraded, uint256 _performanceResult, string _ipfsGradeDetailLink)`

This facilitates grading a specific modeler.

Inputs

- `modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the modeler has been challenged and responded.
  - **Impact**: The modeler to be graded.
- `_ipfsGraded`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The IPFS of the grade.
- `_performanceResult`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The performance result of the modeler.
- `_ipfsGradeDetailLink`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

- **Impact**: The IPFS of the grade detail link.

Branches and code coverage

Intended branches

- Assure that the modeler's response was issued for the specific challenge. Currently not performed.
  - ☐ Test coverage
- Assumed that the `topNModelers` array will properly update within the competition contract. Currently not checked.
  - ☐ Test coverage
- Assumed that the challenge's performance is graded for this contract's specific competition. Currently not explicitly checked.
  - ☐ Test coverage
- Set the `ipfsGraded` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage
- Set the `performanceResult` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage
- Set the `ipfsGradeDetailLink` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage
- Set the `medianPerformanceResults` of the `modelers` mapping for the `modeler`. Currently this simply refers to the `performanceResult` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage

Negative behavior

- Should not allow a validator that is not directly linked to the modeler to grade them. Currently not checked.
  - ☐ Negative test
- Should not allow overriding a grade. Currently not checked.
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☐ Negative test

Function: `provideDrandProof(DataTypes.DrandProof proof)`

This allows providing the proof for the specific round.

Inputs

- `proof`
  - **Control**: Fully controlled by calling function.

- **Constraints**: Checked that the round is not already submitted.
- **Impact**: The proof for the specific round.

Branches and code coverage

Intended branches

- Assure that the `previous_signature` matches the previous round `signature`. Currently not performed.
  - ☐ Test coverage
- Assure that the signature is legitimate and recovers into the `submitter`. Currently not performed.
  - ☐ Test coverage
- Should set the `drands` mapping for the `proof.round` to the `proof`.
  - ☑ Test coverage

Function: `registerModeler(address _modeler, string _modelCommitment, uint256 _stakedAmount)`

This facilitates the registration of a modeler.

Inputs

- `_modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the modeler is not already registered.
  - **Impact**: The modeler to be registered.
- `_modelCommitment`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The model commitment of the modeler.
- `_stakedAmount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens staked by the modeler.

Branches and code coverage

Intended branches

- Assumed most parameters are checked by the calling function. Currently not directly enforced or mentioned in the code.

- ☐ Test coverage
- Assumed that the necessary `stakedAmount` has been already transferred.
  - ☐ Test coverage
- Add the new entry in the `modelers` mapping.
  - ☑ Test coverage
- Append the modeler to the `modelerAddresses` array.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the competition contract.
  - ☑ Negative test
- Should not allow calling this while the contract is paused.
  - ☑ Negative test

Function: `registerValidator()`

This allows the registration of a validator for the Modeler contract.

Branches and code coverage

Intended branches

- Should add the caller to the `validatorAddresses` array. Currently not performed as the `append` is not reachable.
  - ☐ Test coverage
- Should increase the `currentStake` of the validator by `validatorStakeAmount`. Currently not performed as it is set to the stake amount, rather than increased.
  - ☐ Test coverage
- Set the `optOutTime` to `~uint256(0)`. Assumes that this is intended for the rest of the contract/system to work properly.
  - ☐ Test coverage
- Check whether `msg.sender` has already deposited the `validatorStakeAmount` in the contract. Currently not performed.
  - ☐ Test coverage
- Set `msg.sender` as a registered validator.
  - ☑ Test coverage
- Should decrease the `validatorToken` balance of the caller by `validatorStakeAmount`.
  - ☑ Test coverage

Negative behavior

- Should not allow everyone to register as a validator. Currently not performed due to one of the issues described in the audit report.

- ☐ Negative test
- Should not allow the addition of a validator if the `validatorAddresses` array's length is over `MAX_VALIDATORS`.
  - ☑ Negative test

Function: `replaceOptOutTopNModeler(uint256 oldModelerIndex, address oldModeler, address modeler)`

This allows replacing an opt-out top N modeler.

Inputs

- `oldModelerIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the old modeler.
- `oldModeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The old modeler to be replaced.
- `modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The new modeler to replace the old one.

Branches and code coverage

Intended branches

- Assure that the `oldModelerIndex` corresponds to the `oldModeler`. Currently not performed.
  - ☐ Test coverage
- Assure that the modeler's stake has been transferred back. Currently not performed.
  - ☐ Test coverage
- Assure that the old modeler is different than the new one. Currently not performed.
  - ☐ Test coverage
- Assure that the `modeler` has opted out.
  - ☑ Test coverage

Negative behavior

- Assure that the `oldModeler` is a top N modeler. Currently not performed.
  - ☐ Negative test

- Assure that the `oldModeler` has opted out. Currently not performed.
  - ☐ Negative test
- Assure that no one other than the validator can call this function.
  - ☑ Negative test
- Should not allow calling this while the contract is paused.
  - ☑ Negative test
- Should not allow replacing modeler to the modeler that has not opted out.
  - ☑ Negative test

Function: `respondToChallenge(address _validator, string _ipfsResponse)`

This facilitates responding to a challenge.

Inputs

- `_validator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the validator is registered.
  - **Impact**: The validator that gave the challenge.
- `_ipfsResponse`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The IPFS of the response.

Branches and code coverage

Intended branches

- Assumed that the `_validator` actually challenged the caller with the specific `ipfsChallenge`. Currently not explicitly checked.
  - ☐ Test coverage
- Assumed that the challenge was given for this contract's specific competition. Currently not explicitly checked.
  - ☐ Test coverage
- Set the `ipfsResponse` of the `modelerChallenges` mapping for the `modeler`.
  - ☑ Test coverage
- Assure that the `_validator` has been registered.
  - ☑ Test coverage

Negative behavior

- Should not reuse a response that has already been used. Currently not checked.

- ☐ Negative test
- Should not allow overriding a response to a challenge. Currently not checked.
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☐ Negative test
- Should not allow a `msg.sender` that has not registered as a modeler to respond to a challenge.
  - ☑ Negative test

Function: `respondToZKMLChallenge(address _validator, string _ipfsResponse)`

This allows responding to a ZKML challenge.

Inputs

- `_validator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the validator is registered.
  - **Impact**: The validator that gave the challenge.
- `_ipfsResponse`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The IPFS of the response.

Branches and code coverage

Intended branches

- Assure that the `_validator` actually challenged the caller with the specific `ipfsChallenge`. Currently not performed.
  - ☐ Test coverage
- Update the `zc.ipfsResponse` of the `ZKMLChallenges` mapping for the modeler. Currently not performed as memory is used instead of storage.
  - ☐ Test coverage
- Assumed that the challenge was given for this contract's specific competition. Currently not explicitly checked.
  - ☐ Test coverage
- Assure that the `_validator` has been registered.
  - ☑ Test coverage

Negative behavior

- Should not reuse a response that has already been used. Currently not checked.
  - ☐ Negative test
- Should not allow overriding a response to a challenge. Currently not checked.
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☐ Negative test
- Should not allow a `msg.sender` that has not registered as a modeler to respond to a challenge.
  - ☑ Negative test

Function: `setModelerNetPerformanceResultAndUpdate(address modeler)`

This facilitates setting the modeler's net performance result and updating the competition contract.

Inputs

- `modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the modeler has opted out.
  - **Impact**: The modeler to be updated.

Branches and code coverage

Intended branches

- Assure that the opt-out time of the modeler is in the future.
  - ☑ Test coverage
- Assure that the current stake of the modeler is greater than the staked amount of the competition contract.
  - ☑ Test coverage
- Call the `setModelerNetPerformanceResultAndUpdate` function of the competition contract.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than a validator to call this function.
  - ☑ Negative test
- Should not allow calling this while the contract is paused.
  - ☑ Negative test

Function: `setNextRandSlot(address _modeler, uint256 _futureRandSlot)`

This allows the validator to set a future rand slot for a modeler.

Inputs

- `_modeler`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The modeler for which the rand slot is set.
- `_futureRandSlot`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The future rand slot for the modeler.

Branches and code coverage

Intended branches

- Assure that the `futureRandSlots` mapping is unique for the validator and modeler. Currently not performed.
  - ☐ Test coverage
- Assure that the `modeler` does not already have a `futureRandSlot` set. Currently not performed.
  - ☐ Test coverage
- Should set the `futureRandSlots` mapping for the validator and modeler to the `_futureRandSlot`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the validator. Currently not performed.
  - ☑ Negative test

Function: `updateModel(string _modelCommitment)`

This facilitates the update of a modeler.

Inputs

- `_modelCommitment`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.

- **Impact**: The new model commitment of the modeler.

Branches and code coverage

Intended branches

- Reset the `ipfsResponse` of the modeler. Currently not performed.
  - ☐ Test coverage
- Assume that the competition is still ongoing. Currently not explicitly checked.
  - ☐ Test coverage
- Assume that the `_modelCommitment` is different than the previous one. Currently not checked.
  - ☐ Test coverage
- Assume that the `_modelCommitment` is unique. Currently not checked.
  - ☐ Test coverage
- Update the `modelers` mapping for the modeler.
  - ☑ Test coverage
- Update the `modelerChallenges` mapping for the modeler.
  - ☑ Test coverage

Negative behavior

- Should not call `modelers[msg.sender].medianPerformanceResults = 0;` once again, as that is set to 0 a few lines above.
  - ☐ Negative test
- Should not allow calling this while the contract is paused.
  - ☐ Negative test
- Should not allow a `topNModeler` to update their model.
  - ☑ Negative test
- Should not allow a modeler to update their model if they are not registered.
  - ☑ Negative test

5.5. Module: ServiceAgreement.sol

Function: `updateServiceAgreement(DataTypes.SAParams _newAgreement)`

This facilitates the update of the service agreement.

Inputs

- `_newAgreement`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The new service agreement.

Branches and code coverage

Intended branches

- Assure that the `stakedToken` address is the same as before, as otherwise accounting would be wrong across the system.
  - ☐ Test coverage
- Assure that `block.timestamp > endAt`.
  - ☐ Test coverage
- Perform general input checks on `_newAgreement`.
  - ☐ Test coverage
- Set the `lastUpdate` variable to the current block timestamp.
  - ☑ Test coverage
- Set the `agreement` variable to `_newAgreement`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the contract admin.
  - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Spectral Modelers contracts, we discovered eight findings. One critical issue was found. Two were of high impact, three were of medium impact, one was of low impact, and the remaining finding was informational in nature. Spectral Finance acknowledged all findings and implemented fixes.

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.