Zellic

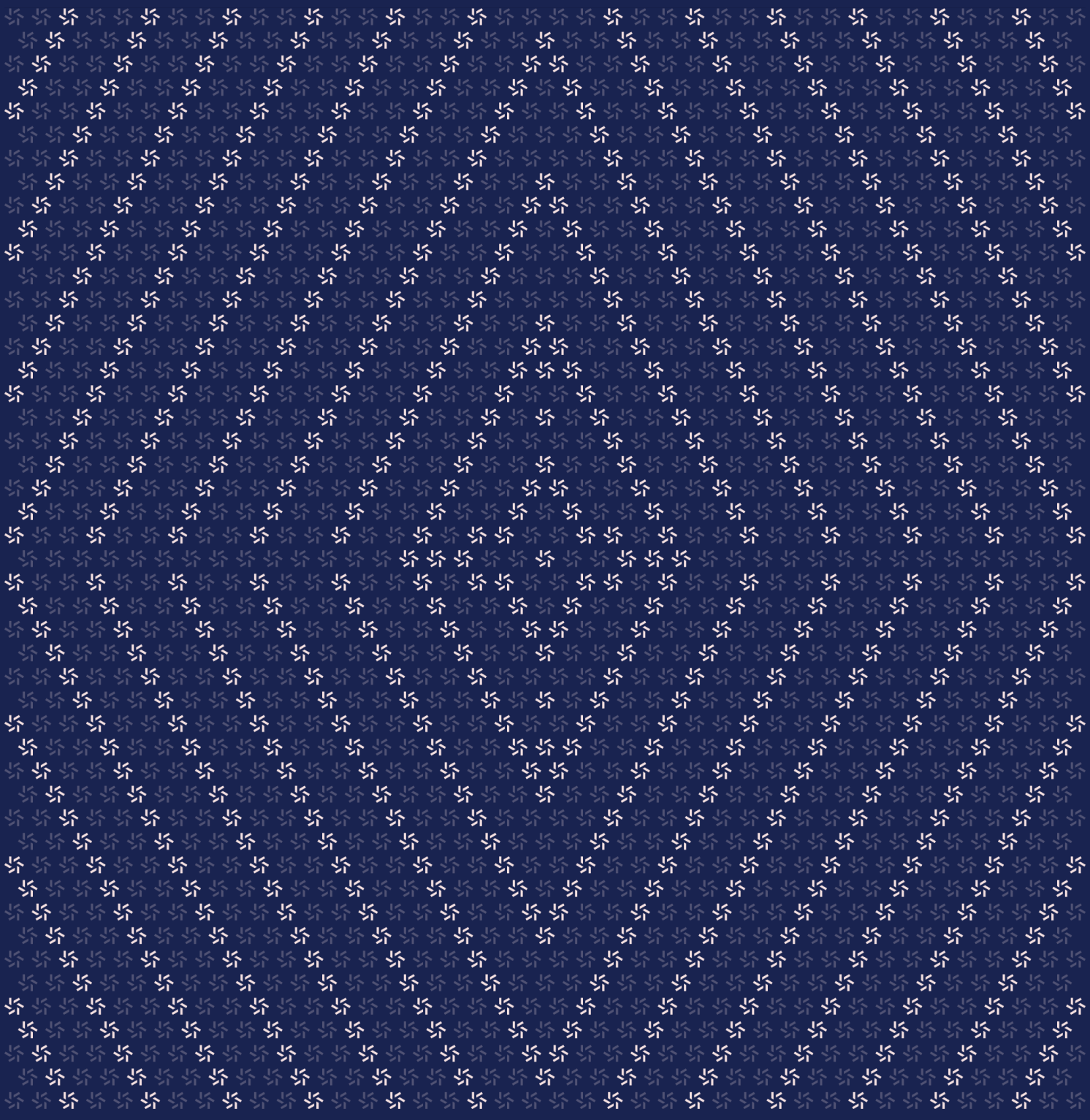
Prepared for
Ming Wu
ZeroGravity

Prepared by
Nipun Gupta
Jade Han
Mohit Sharma
Zellic

August 25, 2024

0G Storage and 0G DA

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 ZeroGravity from July 24th to August 6th, 2024. During this engagement, Zellic reviewed 0G Storage and 0G DA's code for security vulnerabilities, design issues, and general weaknesses in security posture.

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:

- Is it possible for an attacker to submit an incorrect PORA hash and still claim rewards from the contract?
- Could someone front-run the call to submit and create a new answer to claim rewards?
- Is the Merkle tree storing data as expected?
- Is it possible to claim rewards multiple times using same PORA hash?
- Is the data-unsealing process working as expected?

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
- 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 0G Storage and 0G DA contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for ZeroGravity's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	0

2. Introduction

2.1. About 0G Storage and 0G DA

ZeroGravity contributed the following description of 0G Storage and 0G DA:

> 0G is a fast modular AI chain with unlimited scalability by design. It consists of a decentralized storage, a DA, and a serving networks. The nodes in those networks provide variant forms of proofs to the smart contracts on 0g chain for verification to get incentive rewards. The system achieves horizontal scalability through well designed sharding mechanisms on all these networks. Both the storage network and DA network provide reliable data storage but with different mechanisms. Storage network achieves the data reliability through replication, while the DA network through erasure coding with kzg commitment.

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:

0G Storage and 0G DA Contracts

Type	Solidity
Platform	EVM-compatible
Target	0g-storage-contracts
Repository	https://github.com/0glabs/0g-storage-contracts ↗
Version	`dbeff538b949599c203e43be6ecc05e9e997d09d`

Programs	`contracts/dataFlow/*`
	`contracts/market/*`
	`contracts/miner/*`
	`contracts/reward/ChunkLinearReward.sol`
	`contracts/reward/ChunkRewardBase.sol`
	`contracts/reward/Reward.sol`
	`contracts/utils/*`
	`contracts/interfaces/*`

Target	0g-da-contract
Repository	https://github.com/0glabs/0g-da-contract ↗
Version	`37f9cb67c7f526ba5b583bae5133e4830c949a43`
Programs	`DAEntrance.sol` `contracts/libraries/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.1 person-weeks. The assessment was conducted by three consultants over the course of two 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:

Nipun Gupta
Engineer
nipun@zellic.io ↗

Jade Han
Engineer
hojung@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 24, 2024	Start of primary review period
July 26, 2024	Kick-off call
Aug 6, 2024	End of primary review period

3. Detailed Findings

3.1. Incorrect rewards updated in the reward mapping

Target	ChunkRewardBase		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The reward contracts are responsible for storing the rewards for every pricing chunk and distributing those rewards. When a new node is added to the Merkle tree, these newly added sectors are charged. The rewards mapping in ChunkRewardBase is responsible for holding the rewards for each pricing chunk. The rewards are stored for each `SECTORS_PER_PRICE` number of sectors, and when this amount of sectors are completed, the rewards are added to the next pricing index.

The function `fillReward` is responsible for finding the pricing index and adding rewards to the mapping. The function also deducts a small amount of `serviceFee`, which is sent to the treasury. In the case where the `firstPricingIndex` is the same as the `lastPricingIndex` (pricing index is not increased), the amount of rewards added is `msg.value`, as shown below:

```
function fillReward(uint beforeLength, uint chargedSectors) external payable {
    require(_msgSender() == market, "Sender does not have permission");

    uint serviceFee = (msg.value * serviceFeeRateBps) / 10000;
    if (serviceFee > 0) {
        Address.sendValue(payable(treasury), serviceFee);
    }
    uint restFee = msg.value - serviceFee;

    //...

    if (firstPricingIndex == lastPricingIndex) {
        rewards[firstPricingIndex].addReward(msg.value, finalizeLastChunk);
    } else {
        //...
    }
}
```

However, the entire amount is not available for the reward distribution as a part of it is sent to the treasury.

Impact

This might lead to extra rewards being distributed to the miners for some pricing indexes, leading to lesser rewards available for other pricing indexes. In certain scenarios, it might even lead to DOS in the `claimMineReward` and thus the PORA answer submission if enough rewards are not available for distribution.

Recommendations

Call `addReward` using the argument `restFee` instead of `msg.value`.

Remediation

This issue has been acknowledged by ZeroGravity, and a fix was implemented in commit `0d362b69` ↗.

3.2. Misplaced storage gap

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

Description

As the contracts are upgradable, there is a need to put storage gaps to allow developers to freely add new state variables in the future. Without this gap, if new variables are added in the upgraded base contract, they might overwrite the child contract's storage. To address this issue, storage gaps are placed at the end of the base contracts, allowing future versions of that contract to use up those slots without affecting the storage layout of the child contracts. In case of the Flow contract, the storage gap is placed in between the state variables, which could lead to storage collisions if base contracts are upgraded with new state variables.

Impact

There is a possibility of storage collision in case new variables are added to the base contracts.

Recommendations

We recommend either of the following:

1. Use storage gaps at the end of the base contracts, as recommended by OpenZeppelin.
2. If the child contract is setting the slots for the base contracts, set it at the top of the state variables defined in the child contract.

Remediation

This issue has been acknowledged by ZeroGravity, and a fix was implemented in commit b23537d8 ↗.

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. Unnecessary if block could be removed

There is an if block in the `insertNode` function of the FlowTreeLib library that serves no purpose, as shown below:

```
function insertNode(FlowTree storage tree, bytes32 nodeRoot, uint height)
    internal returns (uint) {
    //...
    for (uint i = height; i < totalHeight; i++) {
        if ((startIndex >> i) % 2 == 0) {
            tree.openNodes[i] = currentHeightHash;
            tree.unstagedHeight = i + 1;
            if (i != totalHeight - 1) {
                // console.log("early stop at height %d", i);
            }
            break;
    //...
```

This if block could be removed for some gas savings and improved readability of the code.

Remediation

This issue has been acknowledged by ZeroGravity, and a fix was implemented in commit c456412d ↗.

4.2. Nonlinear increment in `claimableReward` during reward claiming in 'ChunkLinearReward contract

A discussion was raised regarding the current implementation of the reward-claiming process in the `claimMineReward` function within the ChunkLinearReward contract, specifically focusing on the observed behavior of the `claimableReward` variable. Below is a detailed analysis based on the test results shared:

```
function test_Increment() public {
chunklinearreward = new ChunkLinearReward(60*60*24 * 100);
```

```
chunklinearreward.initialize();
chunklinearreward.fillReward{value:  10 ether}(1);

skip(60*60*24 * 10);
console.log(block.timestamp);
chunklinearreward.claimMineReward(1,payable(address(0x1337)),bytes32(0));

skip(60*60*24 * 10);
console.log(block.timestamp);
chunklinearreward.claimMineReward(1,payable(address(0x1337)),bytes32(0));

skip(60*60*24 * 10);
console.log(block.timestamp);
chunklinearreward.claimMineReward(1,payable(address(0x1337)),bytes32(0));

skip(60*60*24 * 10);
console.log(block.timestamp);
chunklinearreward.claimMineReward(1,payable(address(0x1337)),bytes32(0));

skip(60*60*24 * 10);
console.log(block.timestamp);
chunklinearreward.claimMineReward(1,payable(address(0x1337)),bytes32(0));
}
```

```
864001
expectedReleasedReward 1000000000000000000
expectedReleasedReward - releasedReward 1000000000000000000
reward.lockedReward 9000000000000000000
reward.claimableReward 1000000000000000000
1728001
expectedReleasedReward 2000000000000000000
expectedReleasedReward - releasedReward 1000000000000000000
reward.lockedReward 8000000000000000000
reward.claimableReward 1500000000000000000
2592001
expectedReleasedReward 3000000000000000000
expectedReleasedReward - releasedReward 1000000000000000000
reward.lockedReward 7000000000000000000
reward.claimableReward 1750000000000000000
3456001
expectedReleasedReward 4000000000000000000
expectedReleasedReward - releasedReward 1000000000000000000
reward.lockedReward 6000000000000000000
reward.claimableReward 1875000000000000000
4320001
expectedReleasedReward 5000000000000000000
```

```
expectedReleasedReward - releasedReward 1000000000000000000
reward.lockedReward 5000000000000000000
reward.claimableReward 1937500000000000000
```

The results indicated that while `lockedReward` decreases linearly, the `claimableReward` increases exponentially. Also, every time someone calls `claimMineReward` after a certain period has passed, half of the remaining rewards are distributed. We asked if this was intentional by the project team.

Remediation

The project team confirmed that this behavior is intentional. The rewards are designed to drip into the `claimableReward` pool linearly according to `releaseSeconds`. Also, the contract employs a lazy update mechanism, meaning the latest values are updated only when necessary.

Every time a reward is claimed, half of the current `claimableReward` is distributed. For example, after the first 10% of the reward period has elapsed, 1 Ether becomes claimable, of which 0.5 Ether is withdrawn. The remaining 0.5 Ether accumulates with the next 1 Ether over the next 10% period, allowing 0.75 Ether to be claimed, and so on.

Given that this behavior is by design, it is not considered an issue.

4.3. Epoch-range update failure in specific scenarios

A potential issue has been raised regarding the following if statement in the `_makeContext` function with the condition `nextEpochStart + 256 < block.number`.

```
function _makeContext() internal returns (bool) {
//...
    if (nextEpochStart + 256 < block.number) {
        contextDigest = EMPTY_HASH;
        blockDigest = EMPTY_HASH;
    } else {
        //...
    }
    epoch += 1;

    context = MineContext({
        epoch: epoch,
        mineStart: nextEpochStart,
        flowRoot: currentRoot,
        flowLength: tree.currentLength,
        blockDigest: blockDigest,
        digest: contextDigest
```

```
    });
    //...
```

The concern revolves around the `epochRange` not being updated as expected, which might lead to some issues. Below is the detailed reasoning behind this concern.

Assume `firstBlock` is 1,000, and `blocksPerEpoch` is 1,200, as per the documentation. If a `submit` is called at `block.number = 1001`, if the size of the tree becomes 50,000,000 from its initial size of 1, and if there are no new calls to this contract until `block.number = 2600`, calling `makeContext` at this point makes `nextEpochStart` equal to 2,200. Hence, the condition `nextEpochStart + 256 < block.number` becomes true, and the epoch will be updated to 1.

In this scenario,

- Both `context.blockDigest` and `context.digest` will be updated to `EMPTY_HASH`.
- The `context.flowLength` will be 50,000,000.
- The `epochRanges` of this context will remain as (0,0).

This might lead to the epoch not being mined due to the invalid `epochRanges`.

Remediation

Regarding this issue, the ZeroGravity team mentioned that three layers of protection have been implemented to mitigate the problem.

1. An official service will trigger `makeContext` by calling the appropriate interface.
2. Miners' submission of mining results will also trigger `makeContext`.
3. The interface is public, allowing anyone to pay gas fees to trigger `makeContext`.

Despite these measures, there can still be cases where the interface is not called in time, leading to an entire epoch being unable to mine.

Eventually, the data submitted during this period cannot be mined. Since the failure to trigger `makeContext` is considered a low-probability event, the ZeroGravity team will not implement a separate patch.

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: DAEntrance.sol

Function: `submitOriginalData(byte[32][] _dataRoots)`

This function is used to submit data. The user is charged as per the length of the data submitted.

Inputs

- `_dataRoots`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No constraints.
  - **Impact**: Emits the `DataUpload` event for `_dataRoots` and updates `currentEpochReward`, which is dependent on the length of `_dataRoots`.

Branches and code coverage

Intended branches

- Update `currentEpochReward` and `_quorumIndex` and emit correct `DataUpload` events.
  - ☐ Test coverage

Negative behavior

- Revert if the `msg.value` passed is less than expected.
  - ☐ Negative test

Function call analysis

- `this.sync() -> this._syncEpoch() -> DAEntrance.DA_SIGNERS.epochNumber()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.sync() -> this._syncEpoch() -> this._updateRewardOnNewEpoch() -> Address.sendValue(address payable(this.treasury), epochServiceFee)`
  - **What is controllable?** N/A.

- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `DAEntrance.DA_SIGNERS.quorumCount(this.currentEpoch)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value is not controllable.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `submitSamplingResponse(SampleResponse rep)`

This function submits a sampling response.

Inputs

- `rep`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The parameter cannot be reused. Additionally, the `sampleSeed` member must match the `currentSampleSeed` stored in storage, and the `quality` must match the `podasTarget` stored in storage.
    Moreover, the verified root obtained using the `dataRoot`, `epoch`, and `quorumId` members must be valid.
    The sum of `epoch` and `epochWindowSize` must be greater than or equal to the `currentEpoch` stored in storage, and `epoch` must be less than the `currentEpoch` stored in storage.
  - **Impact**: The parameter represents the sample response that needs to be verified.

Branches and code coverage (including function calls)

Intended branches

- Beneficiary can call `withdrawPayments` and withdraw their reward.
  - ☐ Test coverage

Negative behavior

- Reverts if `samplingResponse` for the same quorum is submitted twice.
  - ☐ Negative test
- Reverts if sample is not in the sampling window for the current epoch.
  - ☐ Negative test
- Reverts if commitment for response does not exist.
  - ☐ Negative test
- Reverts if the total number of submissions exceeds `targetRoundSubmissions*2`.

- ☐ Negative test

Function: `submitVerifiedCommitRoots(CommitRootSubmission[] _submissions)`

This function is used to submit commit roots.

Inputs

- `_submissions`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The signatures should be valid.
  - **Impact**: The submissions are updated in the `_verifiedErasureCommitment` mapping.

Branches and code coverage

Intended branches

- If the commitment does not already exist, verify the signature and update the `_verifiedErasureCommitment` mapping.
  - ☐ Test coverage

Negative behavior

- The condition `SLICE_NUMERATOR * total <= hit * SLICE_DENOMINATOR` must be satisfied for all the submissions.
  - ☐ Negative test

Function call analysis

- `this.commitmentExists(_submissions[i].dataRoot, _submissions[i].epoch, _submissions[i].quorumId) -> this.verifiedErasureCommitment(_dataRoot, _epoch, _quorumId) -> SubmissionLib.computeIdentifier(_dataRoot, _epoch, _quorumId)`
  - **What is controllable?** `_dataRoot`, `_epoch`, and `_quorumId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The function computes the identifier, which is the hash of these values.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SubmissionLib.validateSignature(_submissions[i], aggPkG1)`
  - **What is controllable?** `_submissions[i]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

- **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire function call will revert; this is expected.

- `SubmissionLib.identifier(_submissions[i])`
  - **What is controllable?** `_submissions[i]`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the identifier of the submission.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.2. Module: FixedPrice.sol

Function: `chargeFee(uint256 beforeLength, uint256 uploadSectors, uint256 paddingSectors)`

This function charges a fee for uploading sectors and distributes rewards accordingly.

Inputs

- `beforeLength`
  - **Constraints**: N/A.
  - **Impact**: The parameter represents the Merkle tree data length before the upload operation.
- `uploadSectors`
  - **Constraints**: `pricePerSector * uploadSectors` must not exceed `address(this).balance`.
  - **Impact**: The parameter represents the number of sectors to be uploaded.
- `paddingSectors`
  - **Constraints**: N/A.
  - **Impact**: The parameter represents the number of padding sectors to be accounted for.

Branches and code coverage

Intended branches

- The `totalSectors` calculates the sum of `uploadSectors` and `paddingSectors`.
  - ☐ Test coverage
- The `baseFee` calculates the fee based on `uploadSectors` and `pricePerSector`.
  - ☐ Test coverage
- The `bonus` calculates the remaining balance after the `baseFee` is deducted.
  - ☐ Test coverage
- The `paddingPart` and `uploadPart` calculates the proportional fee distribution.
  - ☐ Test coverage
- Conditional filling of rewards is based on the presence of `paddingSectors`.

- ☐ Test coverage

Negative behavior

- Revert if the sender is not `flow`.
  - ☐ Negative test
- Revert if the `baseFee` exceeds the contract balance.
  - ☐ Negative test

Function call analysis

- `IReward(this.reward).fillReward{value: paddingPart}(beforeLength, paddingSectors);`
  - **What is controllable?**: `paddingPart` and `paddingSectors`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact**: Add the reward to the Chunk Index calculated based on `beforeLength` and `paddingSector` by the amount of `paddingPart`.
- `IReward(reward).fillReward{value: bonus + uploadPart}(beforeLength + paddingSectors, uploadSectors);`
  - **What is controllable?**: `bonus + uploadPart`, `beforeLength + paddingSectors`, and `uploadSectors`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact**: Add the reward to the Chunk Index calculated based on `beforeLength + paddingSectors` and `uploadSectors` by the amount of `bonus + uploadPart`.

5.3. Module: Flow.sol

Function: `batchSubmit(Submission[] submissions)`

This function is used to batch-submit nodes to the Merkle tree.

Inputs

- `submissions`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The submissions have to be valid as per the `valid` function in the SubmissionLibrary.
  - **Impact**: These nodes are updated in the Merkle tree.

Branches and code coverage

Intended branches

- If the `block.number` has reached the next epoch, then update the `epoch`, `context`, and `epochStartPosition` and push new values in `epochRanges` and `epochRangeHistory`.
  - ☑ Test coverage
- Add new levels to the Merkle tree if it is not high enough.
  - ☑ Test coverage
- Increase the `tree.currentLength` of the Merkle tree.
  - ☑ Test coverage
- Update the `openNodes` and `unstagedHeight` of the Merkle tree if required.
  - ☑ Test coverage
- Insert the nodes at the heights specified and call `chargeFee` on the market contract.
  - ☑ Test coverage
- Update the `submissionIndex`.
  - ☑ Test coverage

Negative behavior

- Revert if any part of the submission is not valid.
  - ☐ Negative test
- Revert if enough native tokens are not provided during the call as per the size of the submissions.
  - ☐ Negative test

Function call analysis

- `this.submit(submissions[i]) -> SubmissionLibrary.valid(submission)`
  - **What is controllable?** `submission`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value could be true or false, depending on the validity of the submission. If it returns false, the transaction will fail.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> SubmissionLibrary.size(submission)`
  - **What is controllable?** `submission`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the size of the submission. The size is used to calculate the fee needed to insert these nodes.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> this.makeContext() -> this._makeContext() -> FlowTreeLib.commitRoot(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.

    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> this.makeContext() -> this._makeContext() -> FlowTreeLib.root(this.tree)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current root of the tree. The return value is used to update the context's root if the epoch is increased.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> this.makeContext() -> this._makeContext() -> this.rootHistory.insert(currentRoot)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the index of the `currentRoot` in `rootHistory`. It is used to verify that the epoch matches the index.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> this._insertNodeList(submission) -> FlowTreeLib.insertNode(this.tree, nodeRoot, height)`
    - **What is controllable?** `nodeRoot` and `height`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `startIndex`. This value is used to calculate the fee.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> this._insertNodeList(submission) -> IMarket(this.market).chargeFee(previousLength, chargedLength, paddedLength)`
    - **What is controllable?** `chargedLength` and `paddedLength`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.submit(submissions[i]) -> SubmissionLibrary.digest(submission)`
    - **What is controllable?** `submission`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Return value is used in an event.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `makeContextFixedTimes(uint256 cnt)`

This updates the epoch and context a fixed number of times.

Inputs

- `cnt`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints.

- **Impact**: This is the number of times `_makeContext` will be called.

Branches and code coverage

Intended branches

- If the `block.number` has reached the next epoch, then update the `epoch`, `context`, `epochStartPosition`, and push new values in `epochRanges` and `epochRangeHistory`.
  - ☑ Test coverage
- Add new levels to the Merkle tree if it is not high enough.
  - ☑ Test coverage
- Increase the `tree.currentLength` of the Merkle tree.
  - ☑ Test coverage
- Update the `openNodes` and `unstagedHeight` of the Merkle tree if required.
  - ☑ Test coverage

Negative behavior

- Return false if the next epoch has not been reached yet.
  - ☐ Negative test

Function call analysis

- `this._makeContext() -> FlowTreeLib.commitRoot(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._makeContext() -> FlowTreeLib.root(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current root of the tree. The return value is used to update the context's root if the epoch is increased.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._makeContext() -> this.rootHistory.insert(currentRoot)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the index of the `currentRoot` in `rootHistory`. It is used to verify that the epoch matches the index.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `makeContext()`

This updates the epoch and context if the next epoch has been reached.

Branches and code coverage

Intended branches

- If the `block.number` has reached the next epoch, then update the `epoch`, `context`, and `epochStartPosition` and push new values in `epochRanges` and `epochRangeHistory`.
  - ☑ Test coverage
- Add new levels to the Merkle tree if it is not high enough.
  - ☑ Test coverage
- Increase the `tree.currentLength` of the Merkle tree.
  - ☑ Test coverage
- Update the `openNodes` and `unstagedHeight` of the Merkle tree if required.
  - ☑ Test coverage

Negative behavior

- Return false if the next epoch has not been reached yet.
  - ☐ Negative test

Function call analysis

- `this._makeContext() -> FlowTreeLib.commitRoot(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._makeContext() -> FlowTreeLib.root(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current root of the tree. The return value is used to update the context's root if the epoch is increased.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._makeContext() -> this.rootHistory.insert(currentRoot)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the index of the `currentRoot` in `rootHistory`. It is used to verify that the epoch matches the index.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `submit(Submission submission)`

This function is used to submit nodes to the Merkle tree.

Inputs

- `submission`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The submission has to be valid as per the `valid` function in the SubmissionLibrary.
  - **Impact**: These nodes are updated in the Merkle tree.

Branches and code coverage

Intended branches

- If the `block.number` has reached the next epoch, then update the `epoch`, `context`, and `epochStartPosition` and push new values in `epochRanges` and `epochRangeHistory`.
  - ☑ Test coverage
- Add new levels to the Merkle tree if it is not high enough.
  - ☑ Test coverage
- Increase the `tree.currentLength` of the Merkle tree.
  - ☑ Test coverage
- Update the `openNodes` and `unstagedHeight` of the Merkle tree if required.
  - ☑ Test coverage
- Insert the nodes at the heights specified and call `chargeFee` on the market contract.
  - ☑ Test coverage
- Update the `submissionIndex`.
  - ☑ Test coverage

Negative behavior

- Revert if submission is not valid.
  - ☐ Negative test
- Revert if enough native tokens are not provided during the call as per the size of the submission.
  - ☐ Negative test

Function call analysis

- `SubmissionLibrary.valid(submission)`
  - **What is controllable?** `submission`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value could be true or false, depending on the validity of the submission. If it returns false, the transaction will fail.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SubmissionLibrary.size(submission)`
  - **What is controllable?** `submission`.
  - **If the return value is controllable, how is it used and how can it go wrong?**

Returns the size of the submission. The size is used to calculate the fee needed to insert these nodes.

- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.makeContext() -> this._makeContext() -> FlowTreeLib.commitRoot(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.makeContext() -> this._makeContext() -> FlowTreeLib.root(this.tree)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current root of the tree. The return value is used to update the context's root if the epoch is increased.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.makeContext() -> this._makeContext() -> this.rootHistory.insert(currentRoot)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the index of the `currentRoot` in `rootHistory`. It is used to verify that the epoch matches the index.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._insertNodeList(submission) -> FlowTreeLib.insertNode(this.tree, nodeRoot, height)`
  - **What is controllable?** `nodeRoot` and `height`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the `startIndex`. This value is used to calculate the fee.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._insertNodeList(submission) -> IMarket(this.market).chargeFee(previousLengt chargedLength, paddedLength)`
  - **What is controllable?** `chargedLength` and `paddedLength`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SubmissionLibrary.digest(submission)`
  - **What is controllable?** `submission`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return value is used in an event.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.4. Module: Mine.sol

Function: `requestMinerId(address beneficiary, uint64 seed)`

This function generates a new `minerId` and assigns it to a specified beneficiary.

Inputs

- `beneficiary`
  - **Constraints**: N/A.
  - **Impact**: The parameter represents the address that will be assigned as a beneficiary of the new `minerId`.
- `seed`
  - **Constraints**: Must be an arbitrary unique value used for ID generation.
  - **Impact**: The parameter is used to generate a unique `minerId`.

Branches and code coverage

Intended branches

- The `minerId` is generated using the `blockhash`, `msg.sender`, and `seed`.
  - ☐ Test coverage
- The `beneficiaries` mapping variable is updated to assign the newly generated `minerId` to the `beneficiary` address.
  - ☐ Test coverage

Negative behavior

- Revert if the `minerId` has already been registered.
  - ☐ Negative test

Function: `submit(MineLib.PoraAnswer memory answer)`

The function processes and validates a mining submission, adjusts difficulty, and rewards the miner if the submission is valid.

Inputs

- `answer`
  - **Constraints**: `answer.minerId` must not be zero; `beneficiaries[answer.minerId]` must not be zero; `answer.range.numShards()` must be less than or equal to `maxShards`; `flowRoot` must match `context.flowRoot`; `poraOutput` must not have been submitted before; and `uint(poraOutput)` must be less than or equal to `(poraTarget / scaleX64) << 64`.

- **Impact**: The parameter contains the miner's submission details, including `minerId`, nonce, `contextDigest`, range, and more.

Branches and code coverage

Intended branches

- Check if `answer.minerId` is valid and registered.
  - ☐ Test coverage
- Create mining context and validate epoch.
  - ☐ Test coverage
- Reset submissions and adjust difficulty if epoch changes.
  - ☐ Test coverage
- Validate submission details and recall range.
  - ☐ Test coverage
- Optionally unseal data and validate Merkle root.
  - ☐ Test coverage
- Compute PORA hash and validate submission quality.
  - ☐ Test coverage
- Reward miner and update submission count.
  - ☐ Test coverage
- Adjust difficulty if required.
  - ☐ Test coverage

Negative behavior

- Revert if `answer.minerId` is zero.
  - ☐ Negative test
- Revert if `beneficiaries[answer.minerId]` is zero.
  - ☐ Negative test
- Revert if `context.digest` does not match `answer.contextDigest`.
  - ☐ Negative test
- Revert if `context.digest` is equal to `EMPTY_HASH`.
  - ☐ Negative test
- Revert if `currentSubmissions` is greater than or equal to `targetSubmissions`.
  - ☐ Negative test
- Revert if `answer.range.startPosition % SECTORS_PER_PRICE` is not zero.
  - ☑ Negative test
- Revert if `answer.range.startPosition + answer.range.mineLength` exceeds `maxEndPosition`.
  - ☑ Negative test
- Revert if `answer.range.mineLength` exceeds `MAX_MINING_LENGTH * answer.range.numShards()`.
  - ☐ Negative test
- Revert if `answer.range.mineLength` is less than `requiredLength`.

- ☑ Negative test
- Revert if `answer.range.shardId & answer.range.shardMask` is not zero.
  - ☑ Negative test
- Revert if `epochRange.start` is not less than `recallEndPosition` or `epochRange.end` is less than `recallEndPosition`.
  - ☑ Negative test
- Revert if `flowRoot` does not match `context.flowRoot` (if `dataProofEnabled`).
  - ☐ Negative test
- Revert if `chunkOffset * SECTORS_PER_LOAD` exceeds `range.mineLength` in the `recallChunk` function.
  - ☑ Negative test
- Revert if `answer.recallPosition` does not match `answer.range.recallChunk(keccak256(abi.encode(padDigest))) + answer.sealOffset * SECTORS_PER_SEAL`.
  - ☑ Negative test
- Revert if `uint(poraOutput)` exceeds `(poraTarget / scaleX64) << 64`.
  - ☑ Negative test
- Revert if `poraOutput` has been submitted before.
  - ☐ Negative test

Function call analysis

- `IFlow(this.flow).makeContextWithResult()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact** The function checks the start of a new epoch, creates and stores the context for that epoch, and updates the root hash and block hash if necessary.
- `basicCheck(answer, context)`
  - **What is controllable?** `answer`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact** The function checks the basic fields and the validity of the recall range and verifies that the sealed context is within the correct range.
- `MineLib.recoverMerkleRoot(answer, unsealedData)`
  - **What is controllable?** `answer` and `unsealedData`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current root of the tree, which plays an important role in verifying whether the user's submitted answer has a valid proof. If this value can be manipulated, the user could fraudulently claim rewards.
  - **Impact** The function computes the leaf of the hash and recovers the Merkle root through the Merkle tree, then returns it.
- `pora(answer)`

- **What is controllable?** `answer`.
- **If the return value is controllable, how is it used and how can it go wrong?** Returns the PORA hash. This return value is used to verify whether the user's submitted answer is valid according to the PORA mechanism. If this value can be manipulated, the user could fraudulently claim rewards
- **Impact** The function calculates and returns the hash required for verification in PORA.

- `IReward(reward).claimMineReward(answer.recallPosition / SECTORS_PER_PRICE, payable(beneficiary), answer.minerId)`
  - **What is controllable?** `answer.recallPosition / SECTORS_PER_PRICE`, `payable(beneficiary)`, and `answer.minerId`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact** The function claims the mining reward for a specific chunk index and distributes it to the beneficiary.
- `_adjustDifficulty(context)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **Impact** The function adjusts the difficulty based on the current block and the mining start block.

Function: `transferBeneficial(address to, bytes32 minerId)`

The function transfers beneficial ownership of a given `minerId` to a new address.

Inputs

- `to`
  - **Constraints**: N/A.
  - **Impact**: The parameter represents the new beneficiary address.
- `minerId`
  - **Constraints**: Must be owned by the `msg.sender`.
  - **Impact**: The parameter represents the unique identifier of the miner.

Branches and code coverage

Intended branches

- The `beneficiaries` mapping variable is updated to assign `minerId` to the new beneficiary address.
  - ☐ Test coverage

Negative behavior

- Revert if the sender does not own `minerId`.
  - ☐ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the 0G chain.

During our assessment on the scoped 0G Storage and 0G DA contracts, we discovered two findings. No critical issues were found. One finding was of high impact and one was of medium impact.

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.