Zellic

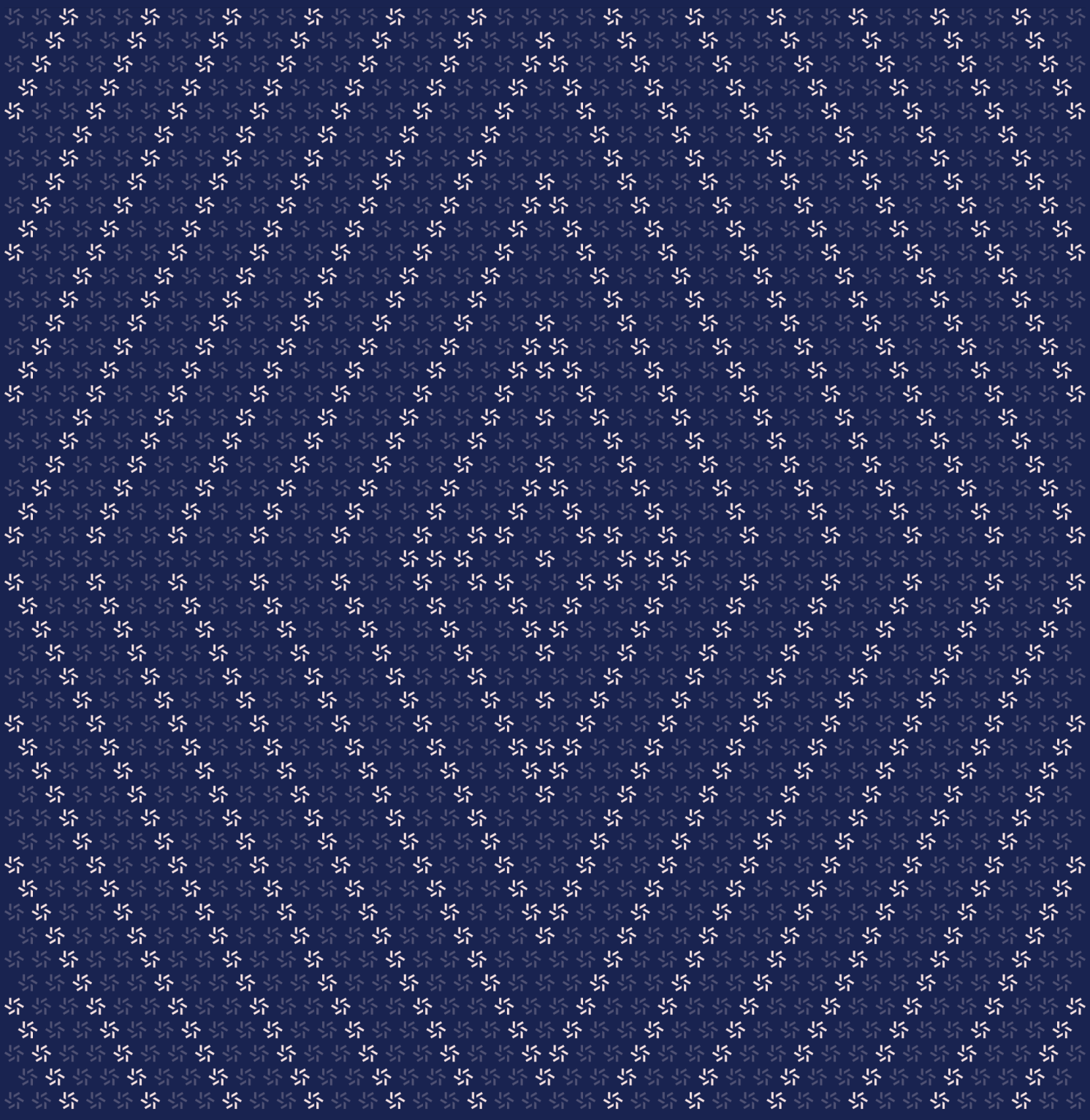
**Prepared for**
**Wyatt Daviau**
Filecoin

**Prepared by**
**Katerina Belotskaia**
**Kuilin Li**
Zellic

**March 18, 2025**

# Proof of Data Possession

## 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 Filecoin from March 3rd to March 10th, 2025. During this engagement, Zellic reviewed Proof of Data Possession'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:

- Could storage providers trigger fault events without submitting a valid proof?
- Could storage providers be blocked from submitting proofs or modifying proof sets due to protocol bugs?
- Has the upgradability functionality been implemented correctly?
- Has the sum-tree structure been implemented correctly? Could adding or removing operations corrupt the sum tree?
- Could a valid proof fail validation?
- Is the deployment of the contracts correct?

## 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.

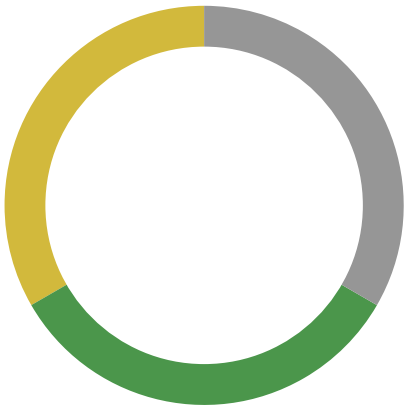
## 1.4. Results

During our assessment on the scoped Proof of Data Possession contracts, we discovered three findings. No critical issues were found. One finding was 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 the benefit of Filecoin in the Discussion section (4. ↗).

## Breakdown of Finding Impacts

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



# 2. Introduction

## 2.1. About Proof of Data Possession

Filecoin contributed the following description of Proof of Data Possession:

> Proof of Data Possession is a provable hot storage protocol facilitating trustless low latency data storage between storage providers and storage clients.

## 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:

### Proof of Data Possession Contracts

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Target** | pdp |
| **Repository** | https://github.com/FilOzone/pdp ↗ |
| **Version** | `830396cafd45e696348edcbce5c114725d6d9b32` |
| **Programs** | `BitOps`<br>`Fees`<br>`Proofs`<br>`Cids`<br>`SimplePDPService`<br>`PDPVerifier` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of 1.2 calendar weeks.

### Contact Information

The following project managers were associated with the engagement:

**Jacob Goreski**
Engagement Manager
jacob@zellic.io ↗

**Chad McDonald**
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

**Katerina Belotskaia**
Engineer
kate@zellic.io ↗

**Kuilin Li**
Engineer
kuilin@zellic.io ↗

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **March 3, 2025** | Kick-off call |
| **March 3, 2025** | Start of primary review period |
| **March 10, 2025** | End of primary review period |

# 3. Detailed Findings

## 3.1. Proof-set owner can reset `proofSetLastProvenEpoch` to avoid accumulated fees

| **Target** | PDPVerifier | | |
|---|---|---|---|
| **Category** | Business Logic | **Severity** | Medium |
| **Likelihood** | Low | **Impact** | Medium |

### Description

The `nextProvingPeriod` function, before initializing the next period, first removes all roots that were previously scheduled for removal by the proof-set owner.

If, as a result of `removeRoots`, all roots are deleted and the proof set becomes empty, the `proofSetLastProvenEpoch` and `nextChallengeEpoch` for this `setId` will be reset to zero.

```
function nextProvingPeriod(uint256 setId, uint256 challengeEpoch,
    bytes calldata extraData) public {
    [...]

    removeRoots(setId, removalsToProcess);
    [...]
    if (proofSetLeafCount[setId] == 0) {
        emit ProofSetEmpty(setId);
        proofSetLastProvenEpoch[setId] = NO_PROVEN_EPOCH;
        nextChallengeEpoch[setId] = NO_CHALLENGE_SCHEDULED;
    }
    [...]
}
```

The issue arises because `proofSetLastProvenEpoch` is used for fee calculation during `provePossession` execution. This value tracks how many blocks have passed since the last proof was provided. So, by design, this is a fee that is charged as time progresses, regardless of how often proofs are actually submitted.

However, if a long time has passed since the previous proving period, the proof-set owner can reset `proofSetLastProvenEpoch` to the current `block.number` by calling `nextProvingPeriod` after removing all roots. This effectively resets the block counter used in fee calculations, allowing them to avoid higher fees that would have accumulated over time had they not reset it first.

### Impact

If a significant amount of time has passed since the last `provePossession` call, the proof-set owner could remove and re-add all roots to reset `proofSetLastProvenEpoch` and avoid paying a portion of the accrued fees.

### Recommendations

We recommend to not reset `proofSetLastProvenEpoch` when all roots are removed. Instead, maintain the original timestamp to ensure that fee accumulation remains consistent.

This does not completely solve the issue, but the general problem of PDPVerifier fee avoidance through collusion between the market and the proof-set owner is not solvable.

This is because the PDPVerifier contract only sees snapshots of the execution of the storage agreement between the market and the proof-set owner, and the fee is integrated over time. These two properties mean that theoretically, the market could instruct the proof-set owner to use the PDPVerifier contract by submitting three checkpoints when it otherwise would submit one — the first one with zero data actually stored, the second one with the real checkpoint it wants proven, and the third with zero data stored. This would mean that the integral of the storage, as seen by the PDPVerifier, is close to zero, so it would charge a negligible fee — but, from the market's perspective, they are still obtaining the proper proofs they need. From the PDPVerifier's perspective, this use case is not distinguishable from a legitimate use case where storage is actually only needed and paid for in short bursts.

Additionally, without extra design elements that pin this contract to the reputation of the underlying protocol, the market can also redeploy this contract without any of the fee logic. This is a more likely solution than asking the proof-set owner to do anything unusual.

### Remediation

This issue has been acknowledged by Filecoin. The off-chain logic and documentation will ensure that removing and re-adding all the roots will be seen as a violation of the contract between the market and the proof-set owner, which will guard against this form of fee avoidance.

## 3.2. Fee refund can revert due to the gas limit or smart wallets

| | | | |
|---|---|---|---|
| **Target** | PDPVerifier | | |
| **Category** | Business Logic | **Severity** | Low |
| **Likelihood** | Medium | **Impact** | Low |

### Description

The `calculateAndBurnProofFee` and `createProofSet` functions automatically refund the excess funds to the `msg.sender` account using the `transfer` function.

```
function calculateAndBurnProofFee(uint256 setId, uint256 gasUsed) internal {
    [...]
    if (msg.value > proofFee) {
        // Return the overpayment
        payable(msg.sender).transfer(msg.value - proofFee);
    }
    emit ProofFeePaid(setId, proofFee, filUsdPrice, filUsdPriceExpo);
}
```

Since the `transfer` function is used, only a limited amount of gas is sent with the transfer, and if the call to the sender address reverts, then the current call context will also revert.

### Impact

Although the gas limit prevents most reentrancy attacks, using `transfer` is generally discouraged ↗ due to gas prices potentially changing in future Ethereum forks.

Additionally, if the sender is a smart contract such as a smart wallet, a governance contract, or a DAO, then it may have complex logic in its receive handler that costs more gas to execute, or it may not even have a payable receive handler. In both cases, this transfer will revert, which makes calling `provePossession` properly difficult for these users. They will either need to work with a block builder to accurately predict the price exactly when their transaction executes or implement custom logic that calculates the price exactly outside the call to `provePossession`.

### Recommendations

Instead of automatically returning funds, consider implementing the common pattern of storing a balance per user and allowing users to withdraw their balance at any time upon request. This approach improves compatibility because the funds will remain in the contract and, if the caller

contract is willing to pay the gas for the withdrawal or upgrades itself in the future to allow for payable transfers, then they can receive the funds they are owed. It can be implemented by tracking overpayments in a mapping, adding to the value whenever an overpayment would otherwise be sent, and then implementing a dedicated withdrawal function.

Separately, for the transfer itself, in order to send the rest of the gas instead of only a limited amount of gas, consider using the `msg.sender.call.value(value)("")` syntax instead of `transfer`:

```
(bool success, ) = msg.sender.call.value(msg.value - proofFee)("");
require(success, "Transfer failed.");
```

If more gas is supplied with the call, ensure that reentrancy cannot become an issue, especially if the second recommendation is implemented without the first. See Finding 3.3. ↗ for more information.

### Remediation

This issue has been acknowledged by Filecoin, and a fix was implemented in commit d6b0a639 ↗.

Filecoin provided the following response:

> Moving transfer call to the end of the `provePossession` call and `createProofSet` call.

### 3.3. Reentrancy risk in `provePossession` during fee refund

| | | | |
|---|---|---|---|
| **Target** | PDPVerifier | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The `provePossession` function verifies the provided proofs and calculates the amount of gas used to determine the fee amount. If the caller provides more `msg.value` than required, the excess funds are refunded using `transfer` function call. However, during this refund process, the caller can attempt to reenter the contract and perform arbitrary actions before the `PDPListener(listenerAddr).possessionProven` call and `proofSetLastProvenEpoch[setId]` update.

```
function provePossession(uint256 setId, Proof[] calldata proofs)
    public payable{
    [...]
    for (uint64 i = 0; i < proofs.length; i++) {
        [...]
        bytes32 rootHash = Cids.digestFromCid(getRootCid(setId,
    challenges[i].rootId));
        bool ok = MerkleVerify.verify(proofs[i].proof, rootHash,
    proofs[i].leaf, challenges[i].offset);
        require(ok, "proof did not verify");
    }
    uint256 gasUsed = (initialGas - gasleft())
    + ((calculateCallDataSize(proofs) + 32) * 1300);
    calculateAndBurnProofFee(setId, gasUsed);

    address listenerAddr = proofSetListener[setId];
    if (listenerAddr != address(0)) {
        PDPListener(listenerAddr).possessionProven(setId,
    proofSetLeafCount[setId], seed, proofs.length);
    }
    proofSetLastProvenEpoch[setId] = block.number;
    emit PossessionProven(setId, challenges);
}

function calculateAndBurnProofFee(uint256 setId, uint256 gasUsed) internal {
    [...]
    if (msg.value > proofFee) {
```

```
        // Return the overpayment
        payable(msg.sender).transfer(msg.value - proofFee);
    }
    emit ProofFeePaid(setId, proofFee, filUsdPrice, filUsdPriceExpo);
}
```

### Impact

Note that this reentrancy is currently not exploitable due to the `transfer` function being limited to 2,300 gas, as of the current Ethereum version. However, it is not good practice to rely on this gas limit to prevent reentrancy, because gas costs can change over time. See Finding 3.2. ↗ for more detailed information.

Also, while the current SimplePDPService implementation of the PDPListener contract prevents reentrancy exploitation due to `block.number` validation (if the caller tries to prove possession again and start a new proving period within the same transaction, the `PDPListener(listenerAddr).possessionProven` call will revert), other PDPListener implementations may not have such verification. This could lead to unintended state changes in the PDPListener contract.

### Recommendations

To mitigate this issue, external calls should only be performed after all state changes are finalized. The possible solution is to move the refund operation to the end of the function, after the `possessionProven` call, ensuring that no unintended reentrant behavior can occur before the PDPListener is notified.

Also, we recommend adding a reentrancy guard to absolutely protect against reentrant calls, both from the sender and from a potential malicious listener.

### Remediation

This issue has been acknowledged by Filecoin, and fixes were implemented in the following commits:

- 4b04bc08 ↗
- 70dbb0e8 ↗

Filecoin provided the following response:

> We've decided to still do refund transfer because we control the control proxy and don't want to control any resident funds. This is a good tradeoff as we don't expect very sophisticated wallets to act as senders of these messages. We've done remediation 2 to maximize compatibility with

different senders forwarding the remaining gas limit to the refunding call.

# 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. Test suite

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

However, several functions in the smart contracts are not covered by any unit or integration tests, to the best of our knowledge. The Forge tests cover most of the functions within the scope of this audit. However, the following functions and require statements have no test coverage.

**PDPVerifier.sol**

- `getProofSetListener`
  - No test case covers this function.
- `addRoots`
  - There is no test case for the scenario where the length of the provided `extraData` exceeds `EXTRA_DATA_MAX_SIZE`.
  - There is no test case ensuring that if someone other than the owner of the proof set tries to call this function, it will revert.
- `scheduleRemovals`
  - There is no test case for the scenario where the length of the provided `extraData` exceeds `EXTRA_DATA_MAX_SIZE`.
  - There is no test case for the scenario where the proof set is not valid.
  - There is no test case for the scenario where the current length of `scheduledRemovals`, after adding new `rootIds`, exceeds `MAX_ENQUEUED_REMOVALS`.
- `provePossession`
  - There is no test case ensuring that if someone other than the owner of the proof set tries to call this function, it will revert.
  - There is no test case for the scenario where `challengeEpoch` is not scheduled.
- `nextProvingPeriod`
  - There is no test case for the scenario where the length of the provided `extraData` exceeds `EXTRA_DATA_MAX_SIZE`.
  - There is no test case ensuring that if someone other than the owner of the

proof set tries to call this function, it will revert.
- There is no test case for the scenario where the provided `challengeEpoch` is less than `block.number + challengeFinality`.

**Cids.sol**

- `digestFromCid`
  - There is no test case for the scenario where the length of the provided `cid.data` is less than 32.

**MerkleVerify.sol**

- `verifyCalldata`
  - No test case covers this function.
- `processProofCalldata`
  - No test case covers this function.

**SimplePDPService.sol**

- `possessionProven`
  - There is no test case for the scenario where the `provingDeadlines[proofSetId]` is not set yet.

The test coverage for this project should be expanded to include all contracts, functions, and test cases provided above, not just surface-level functions.

Good test coverage has multiple effects.

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

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

### Remediation

This issue has been acknowledged by Filecoin, and fixes were implemented in the following commits:

- bfa26cb2 ↗
- eb3b69f5 ↗
- 85226507 ↗
- 5b867389 ↗

## 4.2. Lack of duplicate `rootId` check in `scheduleRemovals()`

The `scheduleRemovals()` function of the PDPVerifier contract allows a proof-set owner to enqueue `rootIds` for removal during the `nextProvingPeriod` function call. However, this function does not verify uniqueness of `rootIds`, allowing the same `rootId` to be added multiple times.

If the proof-set owner unintentionally or accidentally provides duplicate `rootIds`, it will not affect the execution of `removeRoots()` since `sumTreeRemove()` will not modify the state when processing an already removed root. However, these unnecessary entries still consume gas during `nextProvingPeriod()` execution, leading to inefficiencies.

```
function scheduleRemovals(uint256 setId, uint256[] calldata rootIds,
    bytes calldata extraData) public {
    require(extraData.length <= EXTRA_DATA_MAX_SIZE, "Extra data too large");
    require(proofSetLive(setId), "Proof set not live");
    require(proofSetOwner[setId] == msg.sender, "Only the owner can schedule
    removal of roots");
    require(rootIds.length + scheduledRemovals[setId].length
    <= MAX_ENQUEUED_REMOVALS, "Too many removals wait for next proving period
    to schedule");

    for (uint256 i = 0; i < rootIds.length; i++){
        require(rootIds[i] < nextRootId[setId], "Can only schedule removal of
    existing roots");
        scheduledRemovals[setId].push(rootIds[i]);
    }
    [...]
}
```

### Remediation

This issue has been acknowledged by Filecoin.

Filecoin provided the following response:

> Since duplicate removal messages is avoidable user error we'd prefer to make this case expensive for the tradeoff of not doing duplicate removal schedule filtering and making the common case slightly less expensive.

## 4.3. Listener misconduct and market expectations are out of scope

The scope of this audit was defined as the PDPVerifier and SimplePDPService, plus associated dependency contracts. However, the PDPVerifier can be used with any listener, not just the SimplePDPService — the listener is freely specified by the proof-set owner. An external off-chain agent, defined as the market, is responsible for ensuring that the proof-set owner uses the PDPVerifier correctly.

This correctness includes obvious constraints like specifying a listener that is trusted by the market, but it also includes less obvious constraints like ensuring that there are no duplicate or spurious roots in the proof set, that only one proof set exists per root so that the owner cannot reroll the challenge seed with multiple proof sets, or that the listener cannot allow the proof-set owner to control the execution and reenter the PDPVerifier at any time. (Since that can lead to, for example, a modification or deletion of the root set in the middle of the listener logic.)

So, although these constraints were carefully documented, the documentation does not appear to be a complete formal enumeration of these constraints — complete such that any implementation that satisfies all of the constraints can be considered *safe* without a further audit. The threat surface of the overall system can only be characterized completely in an audit that includes the entire system.

Because of this, we strongly recommend an end-to-end audit that includes the market, the SimplePDPService and all other listeners the market may allow, and this PDPVerifier contract. Without this comprehensive audit, differences in the interpretation of responsibilities between smaller audits of individual components may cause issues like these to remain unfound.

### Remediation

This issue has been acknowledged by Filecoin.

## 4.4. Gas optimization

Several functions in the PDPVerifier contract repeatedly access dynamically sized arrays within loops, leading to redundant storage and calldata reads, which increase gas costs. Specifically,

- in `provePossession()`, `proofs.length` is accessed multiple times.
- in `nextProvingPeriod()`, `removals.length` is accessed multiple times.
- in `addRoots()`, `rootData.length` is accessed multiple times.

Since array lengths do not change during function execution, caching these values in local variables at the beginning of execution would optimize gas consumption.

### Remediation

This issue has been acknowledged by Filecoin, and a fix was implemented in commit `dc33146a` ↗.

# 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: SimplePDPService.sol

### Function: `nextProvingPeriod(uint256 proofSetId, uint256 challengeEpoch, uint256, bytes)`

This function can be called by the trusted PDPVerifier contract to set the next proving period. As a result, if a proof has not been provided for the current period, the current period will be counted as skipped.

#### Inputs

- `proofSetId`
  - **Control**: Full control.
  - **Constraints**: No constraints.
  - **Impact**: The identifier of the proof set.
- `challengeEpoch`
  - **Control**: Full control.
  - **Constraints**: `challengeEpoch` must be within the next `challengeWindow`.
  - **Impact**: PDPVerifier sets the desired `challengeEpoch`, and `nextProvingPeriod` ensures that this epoch falls within the next `challengeWindow`.

#### Branches and code coverage

**Intended branches**

- If the current `provingDeadlines[proofSetId]` is zero, the first deadline is calculated as expected.
  - ☑ Test coverage
- If the current `provingDeadlines[proofSetId]` is not zero, the next deadline is calculated as expected.
  - ☑ Test coverage
- If several periods have been skipped, the next deadline is calculated as expected.

☑ Test coverage

- `provenThisPeriod[proofSetId]` is reset to `false`.

☑ Test coverage

- If `faultPeriods` is not zero, the `FaultRecord` event is emitted.

☑ Test coverage

- If `challengeEpoch` is zero, `provingDeadlines[proofSetId]` is set to zero.

☑ Test coverage

**Negative behavior**

- The provided `challengeEpoch` is outside the `challengeWindow`.

☑ Negative test

- The provided `challengeEpoch` is greater than `firstDeadline`.

☑ Negative test

- `block.number` is less than the previous deadline.

☑ Negative test

- The caller is not a trusted PDPVerifier.

☐ Negative test

### Function: `possessionProven(uint256 proofSetId, uint256, uint256, uint256 challengeCount)`

This function verifies that the caller provides the correct number of challenges and sets `provenThisPeriod` to `true` for the given `proofSetId`.

### Inputs

- `proofSetId`
  - **Control**: Full control.
  - **Constraints**: `provenThisPeriod[proofSetId]` should be false.
  - **Impact**: The identifier of the proof set.
- `challengeCount`
  - **Control**: Full control.
  - **Constraints**: Cannot be less than `getChallengesPerProof()`.
  - **Impact**: The number of the provided proofs.

## Branches and code coverage

**Intended branches**

- `provenThisPeriod[proofSetId]` has already been set to true.
  - ☑ Test coverage

**Negative behavior**

- The current period has already been proven.
  - ☑ Negative test
- `challengeCount` is less than `getChallengesPerProof()`.
  - ☑ Negative test
- `provingDeadlines[proofSetId]` has not been set.
  - ☐ Negative test
- `provingDeadlines[proofSetId]` is less than the current `block.number`.
  - ☑ Negative test
- The `challengeWindow` has not started yet.
  - ☑ Negative test
- The caller is not a trusted PDPVerifier.
  - ☐ Negative test

# 6. Assessment Results

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

During our assessment on the scoped Proof of Data Possession contracts, we discovered three findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining finding was informational in nature.

## 6.1. Disclaimer

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

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

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