Zellic

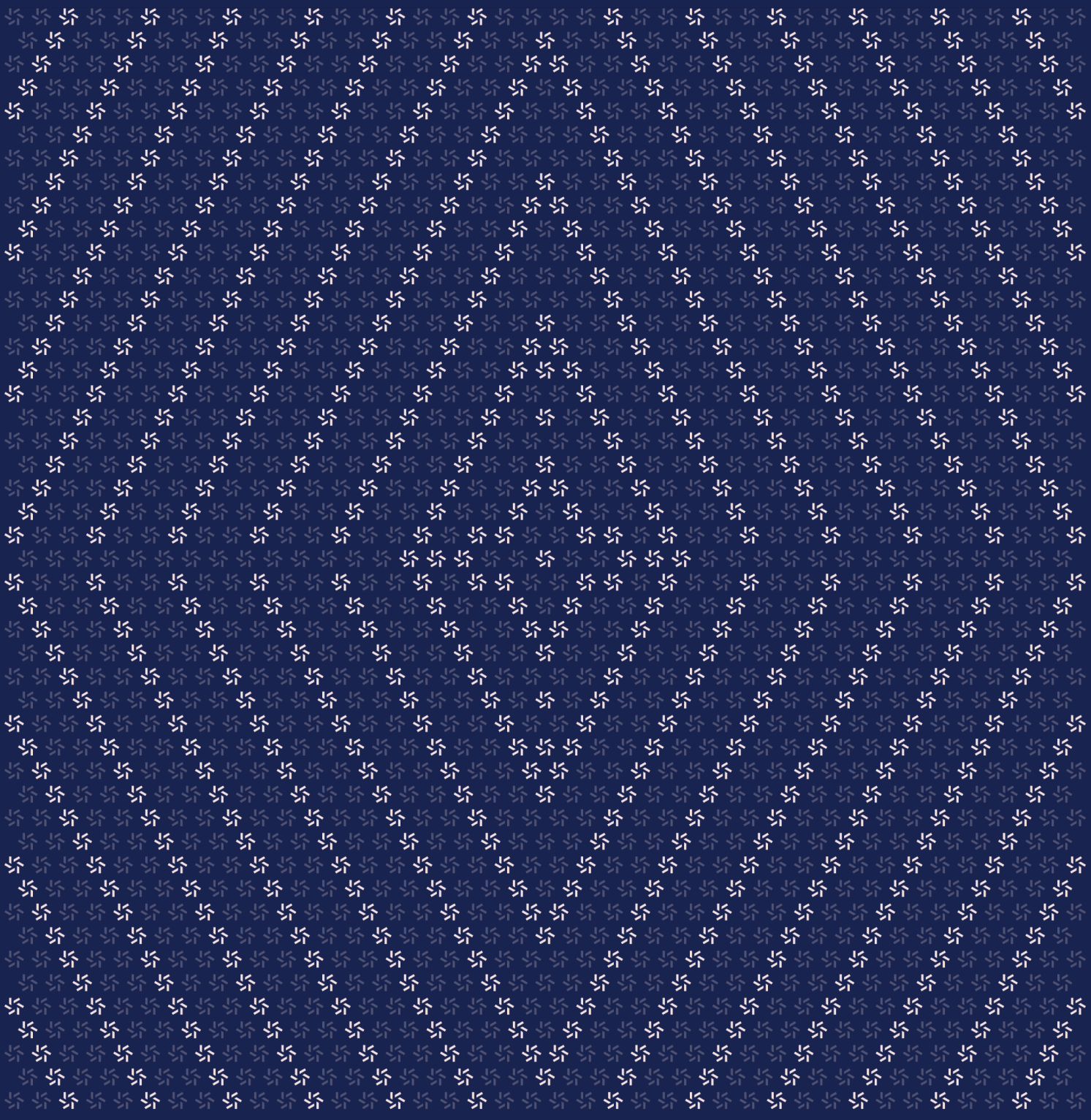
**Prepared for**
Aarsh Shah
Wyatt Daviau
Jakub Sztandera
Filecoin Services

**Prepared by**
Quentin Lemauf
Ulrich Myhre
Zellic

**August 22, 2025**

# Filecoin Services Payments

## Smart Contract Security Assessment



# Contents

**About Zellic** **4**

**1. Overview** **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

**2. Introduction** **6**

2.1. About Filecoin Services Payments 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

**3. Detailed Findings** **10**

3.1. Infinite array could lead to out-of-gas issues 11

3.2. Unused mapping `hasCollectedFees` in Payments 13

3.3. Documentation inconsistencies 14

**4. Threat Model** **14**

4.1. Module: Payments.sol 15

**5. Assessment Results** **33**

5.1. Disclaimer 34

## 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 Services from August 4th to August 11th, 2025. During this engagement, Zellic reviewed Filecoin Services Payments'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 the rail life cycle implemented correctly?
- For the `settleRail` flow, does it properly handle settlements in all configuration combinations?
- Is rail termination safely and securely implemented?
- Are rails compartmentalized in a way such that, for example, a faulty ERC-20 token cannot break anything else than the rails that use it?

## 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 Filecoin Services Payments contracts, we discovered three findings. No critical issues were found. One finding was of low impact and the other findings were informational in nature.

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 0 |
| Low | 1 |
| Informational | 2 |

# 2. Introduction

## 2.1. About Filecoin Services Payments

Filecoin Services contributed the following description of Filecoin Services Payments:

> The Filecoin Services Payments contract is a smart contract that implements point to point payments with lockup and programmable SLA validation before payment settlement. It supports both rolling rate based payments and fixed one time payments.

## 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.

## 2.3. Scope

The engagement involved a review of the following targets:

### Filecoin Services Payments Contracts

| | |
|---|---|
| **Type** | Solidity |
| **Platform** | EVM-compatible |
| **Target** | filecoin-pay |
| **Repository** | https://github.com/FilOzone/filecoin-pay ↗ |
| **Version** | `e4e7dcbc135ba97833ad8f2a574329dbb9c3e721` |
| **Programs** | `Errors.sol`<br>`Payments.sol`<br>`RateChangeQueue.sol` |

## 2.4. Project Overview

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

### 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

**Pedro Moura**
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

**Quentin Lemauf**
Engineer
quentin@zellic.io

**Ulrich Myhre**
Engineer
unblvr@zellic.io

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **August 4, 2025** | Kick-off call |
| **August 4, 2025** | Start of primary review period |
| **August 11, 2025** | End of primary review period |

# 3. Detailed Findings

## 3.1. Infinite array could lead to out-of-gas issues

| | | | |
|---|---|---|---|
| **Target** | Payments.sol | | |
| **Category** | Coding Mistakes | **Severity** | Low |
| **Likelihood** | Low | **Impact** | Low |

### Description

The mapping `payerRails` is defined as `mapping(address => mapping(address => uint256[])) private payerRails;`, which maps an address to another address<->array mapping. During creation of rails, the `railId` of the new rail is pushed onto this array.

```
payeeRails[token][to].push(railId);
payerRails[token][from].push(railId)
```

The array only keeps growing, and elements from it are never removed.

To fetch the information stored in this array, the contract provides these functions:

```
function getRailsForPayerAndToken(address payer, address token)
    external view returns (RailInfo[] memory) {
    return _getRailsForAddressAndToken(payer, token, true);
}

function _getRailsForAddressAndToken(address addr, address token,
    bool isPayer)
    internal
    view
    returns (RailInfo[] memory)
{
    // Get the appropriate list of rails based on whether we're looking for
    payer or payee
    uint256[] storage allRailIds = isPayer ? payerRails[token][addr] :
    payeeRails[token][addr];
    uint256 railsLength = allRailIds.length;

    RailInfo[] memory results = new RailInfo[](railsLength);
    uint256 resultCount = 0;

    for (uint256 i = 0; i < railsLength; i++) {
        uint256 railId = allRailIds[i];
```

```
        Rail storage rail = rails[railId];

        // Skip non-existent rails
        if (rail.from == address(0)) continue;

        // Add rail info to results
        results[resultCount] = RailInfo({railId: railId, isTerminated:
    rail.endEpoch > 0, endEpoch: rail.endEpoch});
        resultCount++;
    }
    // ...
}
```

Note that the function loops over the entire length of the array every time, skipping non-existent rails. Every round of the loop consumes some amount of gas, and at some point iterating over the array length will get a very high gas cost.

### Impact

Currently the function is not called internally, and is meant to be invoked externally instead. Since this is an external view function, there are two types of calls, depending on if the call is part of a transaction or not. In a transaction, the total gas is quite limited and the entire transaction will revert when calling the function on an array that is too long. When using `eth_call` via the RPC, it is up to the RPC to set a gas limit, and this limit can be much higher to the point where this likely won't be a problem.

### Recommendations

Add pagination or a start/stop index as parameters to the function.

### Remediation

This issue has been acknowledged by Filecoin Services. Filecoin Services has stated that they intend to address this issue at a later date.

### 3.2. Unused mapping `hasCollectedFees` in Payments

| | | | |
|---|---|---|---|
| **Target** | Payments.sol | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The `hasCollectedFees` mapping is declared but never read or written anywhere, so it is a dead state.

```
// Tracks whether a token has ever had fees collected, to prevent duplicates in
    feeTokens

mapping(address => bool) public hasCollectedFees;
```

#### Impact

There is no security impact since it is an unused state. However, unused code reduces readability and increases code size. Cleaning it up keeps the codebase simpler and easier to work with for future developers.

#### Recommendations

Remove the mapping and the related comment.

#### Remediation

This issue has been acknowledged by Filecoin Services, and a fix was implemented in commit e4446347 ↗.

## 3.3. Documentation inconsistencies

| | | | |
|---|---|---|---|
| **Target** | Payments.sol | | |
| **Category** | Code Maturity | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The function documentation is sometimes incorrectly stating what the function does, or when it can be called. Examples:

`settleTerminatedRailWithoutValidation` is documented as (our emphasis):

> Settles payments for a terminated rail without validation. This may only be called by the **payee** and after the terminated rail's max settlement epoch has passed. It's an escape-hatch to unblock payments in an otherwise stuck rail (e.g., due to a buggy validator contract) and it always pays in full.

Of course, the payee should never be able to forcibly terminate a rail without validation. This is an emergency function used for the **client** or the **payer**, not the **payee**. The function is correctly implemented, but documented wrong.

### Impact

N/A

### Recommendations

Remove the mapping and the related comment

### Remediation

This issue has been acknowledged by Filecoin Services, and a fix was implemented in PR #206 ↗.

# 4. 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.

## 4.1. Module: Payments.sol

### Function: `createRail(address token, address from, address to, address validator, uint256 commissionRateBps, address serviceFeeRecipient)`

Create a new rail from `from` to `to`, operated by the caller. Can only be called by an approved operator.

#### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: No direct constraints, but the caller has to be approved to create rails using the given token.
  - **Impact**: The ERC20 token address for payments on this rail.
- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Non-zero, and the caller has to be approved to create rails using the given given address.
  - **Impact**: The client address (payer) for this rail.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Non-zero.
  - **Impact**: The recipient address for payments on this rail.
- `validator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Optional address of an validator contract (can be address(0) for no validation).
- `commissionRateBps`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: Lower than or equal to COMMISSION_MAX_BPS.
  - **Impact**: Optional operator commission in basis points (0-10000).
- `serviceFeeRecipient`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Non-zero if there is no commission..
  - **Impact**: Address to receive operator commission.

### Branches and code coverage

**Intended branches**

- Approved operator creates a rail with commission.
  - ☑ Test coverage
- Approved operator creates a rail without commission.
  - ☑ Test coverage

**Negative behavior**

- Called by non-approved operator.
  - ☐ Negative test
- Called with zero address for `from` or `to`.
  - ☐ Negative test
- Called with too high commission.
  - ☐ Negative test
- Called with valid, positive commission but zero address for `serviceFeeRecipient`.
  - ☐ Negative test

## Function: `depositWithPermitAndApproveOperator(address token, address to, uint256 amount, uint256 deadline, uint8 v, byte[32] r, byte[32] s, address operator, uint256 rateAllowance, uint256 lockupAllowance, uint256 maxLockupPeriod)`

This function performs a permit-based deposit and simultaneously approves `operator` with the provided rate, lockup, and max lockup-period allowances for `to`.

### Inputs

- `token`

- **Control**: Fully controlled by the caller.
- **Constraints**: Must be a non-native ERC-20 supporting EIP-2612.
- **Impact**: Asset to deposit and token namespace for operator approval.

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero and equal to `msg.sender` (permit recipient check).
  - **Impact**: Account credited and permit signer.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Intended amount to deposit (credited via balance delta).
- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid per token's permit rules.
  - **Impact**: Permit-validity window.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero.
  - **Impact**: Operator to approve.
- `rateAllowance`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Max cumulative payment rate the operator may configure.
- `lockupAllowance`

- **Control**: Fully controlled by the caller.
- **Constraints**: N/A.
- **Impact**: Max cumulative lockup the operator may configure/use.

- `maxLockupPeriod`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Max lockup period (in epochs) the operator may set.

### Branches and code coverage

**Intended branches**

- Use EIP-2612 permit to grant allowance, perform ERC-20 `transferFrom` into the contract, and increase `to`'s internal balance by `amount` (credit delta).
  - ☑ Test coverage
- Reject zero recipient / incorrect permit recipient.
  - ☑ Test coverage
- Reject invalid/expired permit.
  - ☑ Test coverage
- Success path (sets flags/allowances, emits `OperatorApprovalUpdated`).
  - ☑ Test coverage
- Zero-operator-address revert.
  - ☑ Test coverage

## Function: `depositWithPermitAndIncreaseOperatorApproval(address token, address to, uint256 amount, uint256 deadline, uint8 v, byte[32] r, byte[32] s, address operator, uint256 rateAllowanceIncrease, uint256 lockupAllowanceIncrease)`

This function performs a permit-based deposit and simultaneously increases the rate and lockup allowances for an already approved `operator` for `to`.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a non-native ERC-20 supporting EIP-2612.
  - **Impact**: Asset to deposit and token namespace for operator approval.

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero and equal to `msg.sender` (permit recipient check).
  - **Impact**: Account credited and permit signer.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Intended amount to deposit (credited via balance delta).
- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid per token's permit rules.
  - **Impact**: Permit-validity window.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero and already approved.
  - **Impact**: Operator whose allowances are increased.
- `rateAllowanceIncrease`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount to increase the operator's rate allowance.
- `lockupAllowanceIncrease`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount to increase the operator's lockup allowance.

### Branches and code coverage

**Intended branches**

- Use EIP-2612 permit to grant allowance, perform ERC-20 `transferFrom` into the contract, and increase `to`'s internal balance by `amount` (credit delta).
  - ☑ Test coverage
- Reject zero recipient / incorrect permit recipient.
  - ☑ Test coverage
- Reject invalid/expired permit.
  - ☑ Test coverage
- Success path (increase allowances, emits `OperatorApprovalUpdated`).
  - ☑ Test coverage
- Zero-operator-address revert.
  - ☑ Test coverage
- Operator-not-approved revert.
  - ☑ Test coverage

### Function: `depositWithPermit(address token, address to, uint256 amount, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

This function uses an EIP-2612 permit from `to` to approve and deposit ERC-20 tokens to `to`'s account in a single transaction.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a non-native ERC-20 supporting EIP-2612 — native tokens not supported.
  - **Impact**: Asset to deposit via permit.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero and equal to `msg.sender` (permit recipient check).
  - **Impact**: Account credited and permit signer.
- `amount`
  - **Control**: Fully controlled by the caller.

- **Constraints**: N/A.
- **Impact**: Intended amount to deposit (credited via balance delta).

- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be valid per token's permit rules.
  - **Impact**: Permit-validity window.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Part of a valid ECDSA signature per EIP-2612.
  - **Impact**: Authorizes the allowance used for transfer.

### Branches and code coverage

**Intended branches**

- Use EIP-2612 permit to grant allowance, perform ERC-20 `transferFrom` into the contract, and increase `to`'s internal balance by `amount` (credit delta).
  - ☑ Test coverage
- Reject zero recipient / incorrect permit recipient.
  - ☑ Test coverage
- Reject invalid/expired permit.
  - ☑ Test coverage

## Function: `deposit(address token, address to, uint256 amount)`

This function deposits native or ERC-20 tokens to `to`'s account (using balance before/after for fee-on-transfer tokens) and records the actual amount received.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: If native, `msg.value` must equal the amount; if ERC-20, `msg.value` must be 0.
  - **Impact**: Selects native vs ERC-20 deposit path.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero.
  - **Impact**: Account to credit with the deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: If native, `msg.value` must equal the amount; if ERC-20, `msg.value` must be 0.
  - **Impact**: Nominal amount to deposit (actual amount may be reduced by fee-on-transfer tokens).

### Branches and code coverage

**Intended branches**

- Native token path.
  - ☐ Test coverage
- ERC-20 token path (fee-on-transfer supported).
  - ☑ Test coverage

## Function: `increaseOperatorApproval(address token, address operator, uint256 rateAllowanceIncrease, uint256 lockupAllowanceIncrease)`

This function increases the rate and lockup allowances for an already approved operator for the token and caller.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Token namespace for existing operator approval.

- `operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero and already approved.
  - **Impact**: Target operator whose allowances are increased.
- `rateAllowanceIncrease`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount to increase the operator's rate allowance.
- `lockupAllowanceIncrease`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount to increase the operator's lockup allowance.

### Branches and code coverage

**Intended branches**

- Success path (increases allowances, emits `OperatorApprovalUpdated`).
  - ☑ Test coverage
- Zero-operator-address revert.
  - ☑ Test coverage
- Operator-not-approved revert.
  - ☑ Test coverage

## Function: `modifyRailLockup(uint256 railId, uint256 period, uint256 lockupFixed)`

This function modifies a rail's lockup configuration. If the rail is terminated, only the fixed lockup may be reduced and the period must remain unchanged. If the rail is active, the period and/or fixed lockup may be adjusted subject to account-funding and operator-allowance constraints. The function settles the payer's account before and after.

### Inputs

- `railId`
  - **Control**: Chosen by the operator.
  - **Constraints**: Must reference an active rail (`validateRailActive(railId)`), and `onlyRailOperator(railId)` must hold. Account lockup is settled

before/after via `settleAccountLockupBeforeAndAfterForRail(railId, false, 0)`.
    - **Impact**: Selects which `Rail` is modified and which event is emitted.
- `period`
    - **Control**: Chosen by the operator.
    - **Constraints**: If the rail is terminated, it must equal `rail.lockupPeriod`. If the rail is active and the payer's account is not fully settled, it must also equal `rail.lockupPeriod`. When increasing the period on an active rail, it must be ≤ `operatorApproval.maxLockupPeriod`.
    - **Impact**: Determines the dynamic component of the new lockup — may change `rail.lockupPeriod` (active rails only) and adjust `payer.lockupCurrent` and operator lockup usage accordingly.
- `lockupFixed`
    - **Control**: Chosen by the operator.
    - **Constraints**: If the rail is terminated, it must be ≤ current `rail.lockupFixed`. If the rail is active and the payer's account is not fully settled, it must be ≤ current `rail.lockupFixed`. Any increase requires sufficient operator lockup allowance and sufficient payer funds.
    - **Impact**: Contributes to the new lockup — updates `payer.lockupCurrent`, operator lockup usage, and `rail.lockupFixed`.

## Branches and code coverage

**Intended branches**

- For a terminated rail — reduce `lockupFixed` only, keep `period` unchanged, update payer lockup and operator usage, and emit `RailLockupModified`.
    - ☑ Test coverage
- For an active rail, payer fully settled — change `period` and/or `lockupFixed` within operator limits, adjust payer lockup and operator usage, and emit event.
    - ☑ Test coverage
- For an active rail, payer not fully settled — only allow `lockupFixed` to decrease and `period` to remain unchanged and emit event.
    - ☑ Test coverage
- Revert — terminated rail with `period` change or `lockupFixed` increase.
    - ☑ Test coverage
- Revert — increasing `period` beyond `operatorApproval.maxLockupPeriod`.
    - ☑ Test coverage
- Revert — insufficient `payer.lockupCurrent` for old/new lockup requirements or postsettlement check failure.

☑ Test coverage

### Function: `modifyRailPayment(uint256 railId, uint256 newRate, uint256 oneTimePayment)`

Modifies the payment rate and optionally makes a one-time payment.

#### Inputs

- `railId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid, active rail that the caller is an operator of.
  - **Impact**: The ID of the rail to modify.
- `newRate`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: For terminated rails, it cannot exceed the existing rate. Otherwise, the rail must be settled in order to modify it.
  - **Impact**: The new payment rate (per epoch). This new rate applies starting the next epoch after the current one.
- `oneTimePayment`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot exceed the rail's current `lockupFixed`.
  - **Impact**: Optional one-time payment amount to transfer immediately, taken out of the rail's fixed lockup.

#### Branches and code coverage

**Intended branches**

- Caller is a valid operator, new rate is equal to old rate, no one time payment.
  ☑ Test coverage
- Caller is a valid operator, new rate is equal to old rate, has one time payment.
  ☑ Test coverage
- Caller is a valid operator, new rate is lower than the old rate, no one time payment.
  ☑ Test coverage
- Caller is a valid operator, new rate is lower than the old rate, has one time payment.
  ☑ Test coverage
- Caller is a valid operator, new rate is higher than the old rate, no one time payment.

- ☑ Test coverage
- Caller is a valid operator, new rate is higher than the old rate, has one time payment.
  - ☑ Test coverage
- Caller is a valid operator, the rail is terminated and the rate is equal to the old rate.
  - ☑ Test coverage

**Negative behavior**

- Caller is a valid operator and tries to do a one time payment with insufficient funds.
  - ☐ Negative test
- Caller is not a valid operator.
  - ☐ Negative test
- Caller is a valid operator and tries to call the function on a terminated rail after the end epoch.
  - ☑ Negative test
- Caller is a valid operator and tries to increase the rate of a terminated rail.
  - ☐ Negative test
- Rail is not terminated, but lockup rate is less than the old rate.
  - ☐ Negative test

### Function call analysis

None of the functions do external calls, just internal book keeping.

- Payments.modifyRailPayment->enqueueRateChange(rail, oldRate, newRate)
  - Inserts the rate change into the rail's `rail.rateChangeQueue`.
- Payments.modifyRailPayment->updateOperatorRateUsage
  - If rail rate is modified, this internally updates `approval.rateUsage` to account for the increased or decreased rate.
- Payments.modifyRailPayment->updateOperatorLockupUsage
  - If rail rate is modified, this internally updates `approval.lockupUsage` to account for the increased or decreased rate.
- Payments.modifyRailPayment->updateOperatorAllowanceForOneTimePayment
  - If a one time payment is included, this subtracts the payment from `approval.lockupUsage` and `approval.lockupAllowance`.
- Payments.modifyRailPayment->processOneTimePayment
  - If there is an optional one time payment, this is subtracted from the payer's

funds, crediting it with any fees paid while doing so.

### Function: `setOperatorApproval(address token, address operator, bool approved, uint256 rateAllowance, uint256 lockupAllowance, uint256 maxLockupPeriod)`

This function sets or updates the caller's operator approval for a token, including the approved flag, rate allowance, lockup allowance, and maximum lockup period.

#### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Token namespace for operator approval and allowances.
- `operator`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be nonzero (enforced by modifier).
  - **Impact**: Target operator whose approval and allowances are set.
- `approved`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Enables or disables the operator.
- `rateAllowance`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Max cumulative payment rate the operator may configure across rails.
- `lockupAllowance`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Max cumulative lockup the operator may configure/consume.
- `maxLockupPeriod`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Max lockup period (in epochs) the operator may set per rail.

### Branches and code coverage

**Intended branches**

- Success path (sets flags/allowances, emits `OperatorApprovalUpdated`).
  - ☑ Test coverage
- Zero-operator-address revert.
  - ☑ Test coverage

## Function: `settleRail(uint256 railId, uint256 untilEpoch)`

Settles payments for a rail up to the specified epoch. Settlement may fail to reach the target epoch if either the client lacks the funds to pay up to the current epoch or the validator refuses to settle the entire requested range.

Can only be one of the three participants in the rail, either the payer, payee or operator.

### Inputs

- `railId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must reference an active rail (`from != address(0)`) — caller must be a rail participant (client, operator, or payee).
  - **Impact**: Selects the rail to settle and determines token/payer/payee/validator context used by downstream logic.
- `untilEpoch`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be ≤ current `block.number`. Effective target is bounded by rail state — for active rails, `min(untilEpoch, payer.lockupLastSettledAt)`, and for terminated rails, `min(untilEpoch, rail.endEpoch)`.
  - **Impact**: Upper bound for requested settlement window — influences how much is attempted to be settled.

### Branches and code coverage

**Intended branches**

- Called by client/payer.
  - ☐ Test coverage
- Called by payee.
  - ☐ Test coverage

- Called by operator.
  - ☐ Test coverage
- Called on a terminated rail that is settled, but not finalized.
  - ☐ Test coverage
- Called on a non-terminated rail.
  - ☑ Test coverage

**Negative behavior**

- Caller is not a participant (payer, payee, operator).
  - ☐ Negative test
- `untilEpoch` is in the future.
  - ☐ Negative test
- Rate change queue is empty, but settlement has not progressed (i.e. `startEpoch <= rail.settledUpTo`).
  - ☐ Negative test

### Function call analysis

- this.burnAndRefundRest(Payments.NETWORK_FEE) -> Payments.burnAndRefundRest
  - **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?** The function consumes and native tokens and send these to the burn address. The remainder is refunded.
- this.settleRailInternal(railId, untilEpoch, skipValidation) -> Payments.settleRailInternal
  - **What is controllable?** `railId` and `untilEpoch` are fully controllable. `skipValidation` is always `false` here.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns only calculated values that are relayed to the initial caller directly. `railId` is already assumed valid, but an erroneous one would fail the first `isRailTerminated` check as addresses would be zero.
  - **What happens if it reverts, reenters or does other unusual control flow?** Function is internal and only called from non-reentrant functions, where the second caller is an emergency function locked behind operator capabilities.

### Function: `settleTerminatedRailWithoutValidation(uint256 railId)`

Settles payments for a terminated rail without validation. This may only be called by the payee and after the terminated rail's max settlement epoch has passed. It's an escape-hatch to unblock payments in an otherwise stuck rail (e.g., due to a buggy validator contract) and it always pays in full.

Can only be called by the rail client, i.e. `rails[railId].from == msg.sender`.

### Inputs

- `railId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be a valid, active rail that is also terminated (has `endEpoch > 0`).
  - **Impact**: The ID of the rail to settle.

### Branches and code coverage

**Intended branches**

- Called on active, terminated rail, by its client, after the maximum settlement epoch.
  - ☐ Test coverage

**Negative behavior**

- Called by a non-client.
  - ☐ Negative test
- Called on a non-active rail.
  - ☐ Negative test
- Called on a non-terminated rail.
  - ☐ Negative test
- Called before the maximum settlement epoch.
  - ☐ Negative test

### Function call analysis

- Payments.settleTerminatedRailWithoutValidation->settleRailInternal
  - Same as the external function `settleRail()`, but with `skipValidation == true`.

### Function: `terminateRail(uint256 railId)`

This function terminates a rail by setting its end epoch, optionally notifying the validator, removing its rate from the payer's lockup rate, and updating operator rate usage. It is callable by the operator or a fully settled client.

### Inputs

- `railId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Rail must be active and not already terminated; caller must be the operator or the rail client is fully settled.
  - **Impact**: Rail is terminated and will close at the end of the grace period.

### Branches and code coverage

**Intended branches**

- Terminate as client (settled) or operator.
  - ☑ Test coverage
- Reject unauthorized caller.
  - ☑ Test coverage
- Notify validator (if set).
  - ☑ Test coverage
- Reduce lockup rate and operator usage.
  - ☑ Test coverage

### Function: `withdrawTo(address token, address to, uint256 amount)`

This function withdraws the caller's available (unlocked) funds of `token` to the specified `to` address after settling lockups.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Asset to withdraw from the caller's account.
- `to`

- **Control**: Fully controlled by the caller.
- **Constraints**: Must be nonzero.
- **Impact**: Recipient of withdrawal.

- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than or equal to the unlocked balance after settlement.
  - **Impact**: Amount to withdraw to the `to` address.

### Branches and code coverage

**Intended branches**

- Success — withdraw to `to` (native/ERC-20).
  - ☑ Test coverage
- Revert — `to == address(0)`.
  - ☑ Test coverage
- Revert — amount > unlocked after settlement.
  - ☑ Test coverage

### Function: `withdraw(address token, uint256 amount)`

This function withdraws the caller's available (unlocked) funds of `token` to themselves after settling lockups.

### Inputs

- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Asset to withdraw from caller's account.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be less than or equal to the unlocked balance after settlement.
  - **Impact**: Amount to withdraw to the caller.

## Branches and code coverage

**Intended branches**

- Success — withdraw to self (native/ERC-20).
  - ☑ Test coverage
- Revert — amount > unlocked after settlement.
  - ☑ Test coverage

# 5. Assessment Results

During our assessment on the scoped Filecoin Services Payments contracts, we discovered three findings. No critical issues were found. One finding was of low impact and the other findings were informational in nature.

At the time of our assessment, the reviewed code was deployed to the calibration net but not yet deployed to mainnet.

Overall, the codebase is very mature. Every function is thoroughly documented, every condition and constraint clearly listed, and there are a multitude of test cases for each function that cover nearly every situation.

### Key areas for enhancement

While code coverage is good, there are still a few uncovered corner cases. These can be found, for example, by using the lcov report of the `forge coverage` tooling, which shows some missing spots. Examples include depositing native tokens or attempting various operations on a terminated rail. We recommend getting as close to 100% test coverage as possible, as this inspires greater confidence that every corner case has been considered.

We highly recommend writing larger scenario-based test cases that simulate as close to real-life scenarios as possible, including multiple users and multiple rails in various states. This is often called end-to-end or acceptance testing. Testing a deployment locally can uncover issues that a configuration net deployment does not exhibit, for instance that configurations are correctly tweaked during the deployment process.

Tests should — where possible — explicitly wait for the correct revert reason where a custom reason is used. As of the time of writing, tests are doing `vm.expectRevert()` without an expected message in quite a few places in the test code.

## 5.1. 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.