Least Authority

PRIVACY MATTERS

Smart Contracts

Security Audit Report

FilFi

Final Audit Report: 25 September 2023

Table of Contents

Overview

Background

FilFi has requested that Least Authority perform a security audit of their smart contracts.

Project Dates

- **September 1, 2023 - September 18, 2023:** Initial Code Review *(Completed)*
- **September 20, 2023:** Delivery of Initial Audit Report *(Completed)*
- **September 24, 2023:** Verification Review *(Completed)*
- **September 25, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Mukesh Jaiswal, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the FilFi smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- zip file (FilFi0830.zip) *(shared with Least Authority via email on 30 August 2023*)

For the review, this repository was cloned for use during the audit and for reference in this report:

- FilFi Smart Contracts:
  https://github.com/LeastAuthority/filfi-smart-contracts-review

This repository was cloned and examined for the verification :

- zip file (FilfiContract_01.zip) *(shared with Least Authority via email on 23 September 2023*)
- https://github.com/LeastAuthority/filfi-smart-contracts-review-verification

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Project website:
  https://filfi.io

In addition, this audit report references the following documents:

- Downsizing Contracts To Fight the Contract Size Limit:
  https://ethereum.org/en/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit/

- Solidity Style Guide: https://docs.soliditylang.org/en/stable/style-guide.html
- NatSpec Guidelines: https://docs.soliditylang.org/en/v0.8.21/natspec-format.html
- Best Practices: https://docs.filecoin.io/smart-contracts/developing-contracts/best-practices#contracts-interacting-with-built-in-actors

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the smart contracts' code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of the FilFi smart contracts, which manage the storage node joint construction programs in the Filecoin network. The implementation can be described as a co-mining product that achieves its goal through different stages – the fundraising period, encapsulation progress, and distribution of rewards.

In addition to reviewing the security of the system and the areas of concern listed above, we reviewed the smart contracts for vulnerabilities and implementation errors and to assess adherence to best practice recommendations. We identified several issues and suggestions in the coded implementation relating to the deviation from recommended best practices, implementation errors, and input validation that could affect the security and the functionality of the contracts, as detailed below.

System Design

Our team examined the design of the FilFi smart contracts and found that security has generally been taken into consideration, as demonstrated by the implementation of proper access control. However, our team identified a pattern of lack of consideration for gas optimization as demonstrated by the usage of unnecessary conditions, redundant functions, and checks (Suggestion 1, Suggestion 2, Suggestion 7, Suggestion 8, Suggestion 9, Suggestion 10, Suggestion 11). We also identified a missing check for invalid addresses, which can lead to the loss of funds (Issue C).

In addition, our team found an instance where input is processed without any validation, which can result in unintended behavior (Issue E).

Code Quality

In our review of the codebase, we identified several instances of deviations from development best practices. We found that state variables and events are not properly named, and that functions are written with a level of abstraction that makes the code difficult to navigate and understand. We recommend adhering to a naming convention for variables and functions that is descriptive and accurate, and that is consistent with generally accepted best practices (Suggestion 6). We also found unused code, which reduces readability and can confuse reviewers and maintainers, possibly resulting in security vulnerabilities (Suggestion 2).

Tests

During our review, our team found no tests within the codebase. We recommend implementing a test suite, which helps identify implementation errors that could lead to security vulnerabilities (Issue G).

Documentation and Code Comments

The documentation consisted of an outdated paper, which lacks accuracy and insufficiently describes the system, each of its components, and the interaction between those components. Additionally, there were no code comments describing security-critical components and functions in the codebase. We recommend that the project documentation be improved and that the code comments in the repository be rewritten to comprehensively describe the intended behavior of each function and component, and how these components interact with each other (Issue F).

Scope

The scope of this review was sufficient and included all security-critical components. However, our team's ability to perform a comprehensive security analysis of the system was reduced due to the lack of documentation. Lack of access to appropriate resources inhibited our ability to comprehensively understand the business logic of the system and made it difficult to reason about the overall soundness of the implementation. Due to these limitations, we strongly recommend that a follow-up security audit of the smart contracts be conducted, once the Issues and suggestions in this report are addressed (Suggestion 12).

Dependencies

Our team did not identify any security issues in the use of dependencies. The FilFi team utilizes the upgraded version for the Filecoin Solidity library, in accordance with best practices.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: startPreSeal Function Can Be Called After Fundraising Plan Has Been Successfully Ended	Resolved
Issue B: pushFinalProgress Function Can Be Called More Than Once	Resolved
Issue C: Missing Check for the CommonType.FileAddress	Resolved

Issue D: getBalance Withdraws All the Balance of a Miner	Resolved
Issue E: Lack of Input Validation for the CreateRaisePlan Function	Resolved
Issue F: Insufficient Project Documentation and Code Comments	Unresolved
Issue G: Missing Test Suite	Unresolved
Suggestion 1: Move the Duplicate Code Into a Private Function	Resolved
Suggestion 2: Remove Redundant Functions	Resolved
Suggestion 3: Implement the Steps To Downsize the Contract	Unresolved
Suggestion 4: Use Fit Data Types for Storage Variables	Unresolved
Suggestion 5: Avoid Accessing Same Storage Multiple Times	Unresolved
Suggestion 6: Adhere to Solidity and NatSpec Guidelines	Unresolved
Suggestion 7: Remove Redundant Checks	Resolved
Suggestion 8: Make opsFundReward Function Return Zero When RaiseState Is Failure	Resolved
Suggestion 9: Optimize getBack Function	Unresolved
Suggestion 10: Return Early To Save Gas	Unresolved
Suggestion 11: Avoid Duplicated Calculations	Resolved
Suggestion 12: Conduct Follow-up Security Audit	Unresolved

Issue A: startPreSeal Function Can Be Called After Fundraising Plan Has Been Successfully Ended

Location

`filfi/LetsFilProcess.sol#L366`

Synopsis

When a fundraising plan is successfully ended by either one of the functions, `closeRaisePlan` or `raiseExpire`, or due to `staking`, the `_raiseSuc` function is called to trigger the successful ending of the fundraising plan and start the sealing phase by changing the `raiseState[planId]` to `RaiseState.Success`. At this stage, the `startPreSeal` function calls for this fundraising plan should revert since the sealing period has started. However, an incorrect check at the start of the `startPreSeal` function allows the function to proceed.

Impact

Incorrect values for a fundraising plan may be set since the function assumes that it is the pre-sealing period.

Preconditions

For this Issue to occur, the fundraising plan must have been successfully ended.

Technical Details

```
require(raiseState[id] == RaiseState.Raising || raiseState[id] ==
RaiseState.Success, "Process: state err.");
```

Remediation

We recommend removing the check `raiseState[id] == RaiseState.Success` in the first `require` statement in the `startPreSeal` function.

Status

The FilFi team has removed the aforementioned check.

Verification

Resolved.

Issue B: pushFinalProgress Function Can Be Called More Than Once

Location

`filfi/LetsFilProcess.sol#L401-L409`

Synopsis

The `pushFinalProgress` function should be called one last time, two days after the sealing period ends for a particular fundraising plan. However, this function can be called more than once for the plan during the aforementioned period, resetting an already finalized sealing progress.

Impact

Already finalized states dependent on the sealing amount may be set to new values, resulting in unexpected behavior.

Preconditions

This Issue is likely if the `pushFinalProgress` function for a fundraising plan was already called once.

Remediation

We recommend preventing the function from being called more than once for a fundraising plan by adding the following check at the start of the function:

```
require(!progressEnd[id], "Final progress already pushed");
```

Status

The FilFi team has added the recommended check.

Verification

Resolved.

Issue C: Missing Check for the CommonType.FileAddress

Location

`filfi/LetsFilMiner.sol#L42`

Synopsis

The function `changeOwner(uint64 minerId, CommonTypes.FilAddress memory addr)` lacks input validation for the address, due to which an address of arbitrary length can be passed.

Impact

An invalid address can be passed to `FilAddress`, which will set the invalid address for the owner and result in the loss of funds.

Remediation

We recommend using the `validate` function from `utils/FilAddresses.sol#L63` to validate the address.

Status

The FilFi team has implemented the remediation as recommended.

Verification

Resolved.

Issue D: getBalance Withdraws All the Balance of a Miner

Location

`filfi/LetsFilMiner.sol#L50`

Synopsis

The function `getbalance`, which should be used to get the balance of a miner, withdraws all the available balance of a miner and moves it to the contract here.

Impact

When a miner attempts to verify their balance, they would inadvertently initiate a withdrawal of the available amount, which was not their intended action.

Remediation

We recommend removing the `MinerAPI.withdrawBalance` call from the function so that it only fetches the available balance.

Status

The FilFi team has removed the `getBalance` function.

Verification

Resolved.

Issue E: Lack of Input Validation for the CreateRaisePlan Function

Location

filfi/LetsFilRaiseFactory.sol#L67

Synopsis

There is no input validation for the function createRaisePlan, which can result in _raiseInfo.targetAmount or _nodeInfo.opsSecurityFund being zero.

Impact

1. If _raiseInfo.targetAmount is zero, then the security deposit here will be zero as well, and raiser will be able to deposit zero as a security fund amount, as follows:

   ```
   function paySecurityFund(uint256 id) public payable onlyRaiser
   ```

2. If msg.value is not zero and nodeInfo.opsSecurityFund is zero then the storage provider (SP) will only have to pay spSafeSealFund. Additionally, if msg.value is zero and nodeInfo.opsSecurityFund is not zero, then the entire condition in the require function will not be true, which can trigger a revert.

Remediation

We recommend adding input validation for all the parameters present in the createRaisePlan function and implementing a check to verify that msg.value is not zero.

Status

The FilFi team has added zero checks for _raiseInfo.targetAmount, _raiseInfo.securityFund, _nodeInfo.opsSecurityFund and added the check to verify that msg.value is not zero.

Verification

Resolved.

Issue F: Insufficient Project Documentation and Code Comments

Synopsis

The general documentation provided by the FilFi team was minimal. Robust and comprehensive documentation allows a security team to assess the in-scope components and understand the expected behavior of the system being audited. In addition, clear and concise user documentation provides users with a guide to utilize the application according to security best practices.

Additionally, the codebase lacks explanation in some areas. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation explaining, for example, expected function behavior and usage, input arguments, variables, and code branches can greatly benefit the readability, maintainability, and auditability of the codebase. This allows both maintainers and reviewers of the codebase to comprehensively understand the intended functionality of the implementation and system design, which increases the likelihood for identifying potential errors that may lead to security vulnerabilities.

Remediation

We recommend that the FilFi team improve the project's general documentation by creating a high-level description of the system, each of the components, and the interactions between those components. This can include developer documentation and architectural diagrams. We additionally recommend expanding and improving the code comments within the components to facilitate reasoning about the security properties of the system.

Status

The FilFi team has decided not to improve the documentation and code comments at the current stage due to time limitations.

Verification

Unresolved.

Issue G: Missing Test Suite

Synopsis

Our team found no tests in the repository in scope. Sufficient test coverage should include tests for success and failure cases, which helps identify potential edge cases, and protect against errors and bugs that may lead to vulnerabilities or exploits. A test suite that includes a minimum of unit tests and integration tests adheres to development best practices. In addition, end-to-end testing is also recommended to assess if the implementation behaves as intended.

Mitigation

We recommend that the FilFi team create a test suite for the FilFi smart contracts to facilitate identifying implementation errors and potential security vulnerabilities by developers and security researchers.

Status

The FilFi team has decided not to add a test suite at the current stage due to time limitations.

Verification

Unresolved.

Suggestions

Suggestion 1: Move the Duplicate Code Into a Private Function

Location

`filfi/LetsFilControler.sol#L543-L547`

`filfi/LetsFilControler.sol#L560-L564`

`filfi/LetsFilControler.sol#L577-L581`

Synopsis

The functions `availableRewardOf`, `totalRewardOf`, and `willReleaseOf` in `LetsFilControler` share the same `if/else` statements, which can be moved to a private function to save gas and improve readability.

Mitigation
We recommend creating a private function, moving the referenced `if/else` statements to this function, and calling it appropriately.

Status
The FilFi team has implemented the remediation as recommended.

Verification
Resolved.

Suggestion 2: Remove Redundant Functions

Location
`filfi/LetsFilProcessSecond.sol#L194-L202`

`filfi/LetsFilControler.sol#L293-L296`

Synopsis
The functions `_raiseSuc`, `setSealTime`, and `setSectorPackage` referenced above are redundant in the smart contracts and should be removed.

Mitigation
We recommend removing the aforementioned functions to save gas and improve readability.

Status
The FilFi team has removed the aforementioned functions.

Verification
Resolved.

Suggestion 3: Implement the Steps To Downsize the Contract

Location
`filfi/LetsFilControler.sol#L31`

`LetsFilProcess.sol#L33`

`LetsFilProcessSecond.sol#L33`

Synopsis
The `LetsFilControler` contract code size is 37284 bytes, the `LetsFileProcess` contract code size is 36517 bytes, and the `LetsFileProcessSecond` contract code size is 25270 bytes. All of these contracts exceed the standard contract size limit of 24576 bytes, due to which the contracts may not be deployable on the mainnet.

Mitigation
We recommend implementing steps to decrease the size of the contracts.

Status

The FilFi team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 4: Use Fit Data Types for Storage Variables

Location

`filfi/interfaces/ILetsFilPackInfo.sol#L4`

Synopsis

This interface defines structs. However, the variables in the struct could fit into smaller data types and be packed together, along with other variables, into fewer slots for gas optimization.

Mitigation

We recommend using fit data types for storage variables.

Status

The FilFi Team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 5: Avoid Accessing Same Storage Multiple Times

Location

Examples (non-exhaustive):

`filfi/LetsFilControler.sol#L256-L259`

`filfi/LetsFilProcess.sol#L210-L212`

Synopsis

Our team confirmed that multiple read access attempts were made to the same storage variables in many parts in the contracts. Reading from the storage variable incurs much more cost than using memory variables.

Mitigation

We recommend using temporary memory variables, instead of storage variables that may be accessed multiple times, to ensure that they are accessed only once.

Status

The FilFi Team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 6: Adhere to Solidity and NatSpec Guidelines

Location

Examples (non-exhaustive):

`filfi/LetsFilProcess.sol#L96-L108`

Synopsis

The codebase does not adhere to the Solidity style guide. For example, constant variables in the aforementioned location are written in `camelCase`, which makes it difficult to distinguish between variables and constants. Additionally, the codebase does not follow NatSpec guidelines for code comments.

Mitigation

We recommend utilizing UPPERCASE and following the style guide for building Solidity smart contracts.

Status

The FilFi Team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 7: Remove Redundant Checks

Location

`filfi/LetsFilProcess.sol#L377-L378`

`filfi/LetsFilProcess.sol#L429-L433`

`filfi/LetsFilProcess.sol#L286-L292`

Synopsis

Our team found some instances of redundant checks:

1. The referenced checks in the `startPreSeal` function are redundant and should be removed;
2. The second check in the referenced `if` statement in the `_pushFinalProgress` function is redundant because before the code execution flow reaches this statement, a check is implemented at the start of the `pushFinalProgress` function to ensure that `block.timestamp >= sealEndTime[id] + 2 days`; and
3. The referenced `if` condition in the `staking` function will always be true if `msg.value` is greater than zero. Therefore, it is redundant since the execution would revert due to other checks in the function if `msg.value` is zero.

Mitigation

We recommend optimizing how `startSealTime[id]` and `sealEndTime[id]` are set in the `startPreSeal` function to avoid redundancy. We also recommend removing the unnecessary checks to save gas and improve readability.

Status

The FilFi team has removed the aforementioned checks.

Verification

Resolved.

Suggestion 8: Make opsFundReward Function Return Zero When RaiseState Is Failure

Location

[filfi/LetsFilControler.sol#L339]()

Synopsis

The function `opsFundReward` is called from `withdrawOpsSecurityFund` in the `LetsFilControler` smart contract to calculate the operations fund reward. However, when the value of `RaiseState` is `Failure`, there is no reward to calculate. Therefore, this calculation is unnecessary.

Mitigation

We recommend adding a check in the `opsFundReward` function to return zero if `RaiseState` is `Failure` to improve gas efficiency and readability.

Status

The FilFi team has implemented the remediation as recommended.

Verification

Resolved.

Suggestion 9: Optimize getBack Function

Location

[filfi/LetsFilControler.sol#L376-L412]()

Synopsis

The function `getBack` in the `LetsFilControler` smart contract can be optimized by removing unnecessary or duplicate checks.

Mitigation

We recommend optimizing the aforementioned function to save gas and improve readability.

Status

The FilFi Team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 10: Return Early To Save Gas

Location

filfi/LetsFilProcess.sol#L539

Synopsis

The function `_pushSpFine` in the `LetsFilProcess` smart contract can return early when `fineRemain` is zero.

Mitigation

We recommend returning the function call early when `fineRemain` is zero to save gas and improve readability. Necessary states can be set to zero before returning.

Status

The FilFi team has acknowledged this suggestion and stated that they do not intend to implement these changes at the present time.

Verification

Unresolved.

Suggestion 11: Avoid Duplicated Calculations

Location

Examples (non-exhaustive):

filfi/LetsFilControler.sol#L272-L274

filfi/LetsFilControler.sol#L281-L283

Synopsis

Duplicated calculations in the codebase spend unnecessary gas to get the same results.

Mitigation

We recommend using a variable to store a value from the duplicated calculations.

Status

The FilFi team has implemented the remediation as recommended.

Verification

Resolved.

Suggestion 12: Conduct Follow-up Security Audit

Synopsis

For complex implementations, comprehensive documentation and code comment coverage is imperative to reason about the functionality and security characteristics of the system. The lack of code comments and implementation documentation evokes security by obfuscation, which is not a valid approach to securing systems potentially controlling large amounts of value.

During this review, our ability to perform a comprehensive security analysis of the system was reduced due to the lack of documentation and code comments. Consequently, this inhibited our ability to accurately reason about the business logic of the implementation.

Mitigation

We strongly recommend performing a comprehensive, follow-up security audit by an independent third-party team once the Issues and suggestions in this report have been adequately addressed.

Status

At the time of this verification, the FilFi smart contracts have not undergone a separate security review. We recommend that the FilFi team continue to consider a follow up audit of the smart contracts.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.