# Security Review Report
# NM-0073 Fileverse

NETHERMIND

(Apr 10, 2023)

# Contents

# 1 Executive Summary

This document presents the security review performed by Nethermind on the Fileverse Contracts. Fileverse is a tool for Web3 communities and individuals. It is a file-sharing and access management system between blockchain addresses. Fileverse solves the major pain points of Web2 file sharing services while breaking new ground in access rights management and content encryption. It leverages p2p storage networks (like IPFS) and encryption enabled via crypto wallets' (e.g., MetaMask) private key signatures. The audit focuses on Fileverse smart contracts. These smart contracts allow users to manage their interactions with peers in the system.

**The audited code consists of** 412 lines of Solidity with code coverage of 98.8%. The Fileverse team provided three documents to assist the audit presenting an overview of the smart contracts, how contracts interact with each other, and how to run the test suite. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 13 points of attention, where two are classified as `Medium`, five are classified as `Low`, and six points of attention are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

**This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 presents the assumptions for this audit. Section 4 summarizes the issues. Section 5 presents the system overview. Section 6 discusses the risk rating methodology adopted for this audit. Section 7 details the issues. Section 8 discusses the documentation provided by the client for this audit. Section 9 presents the compilation, tests, coverage, and automated tests. Section 10 concludes the document.

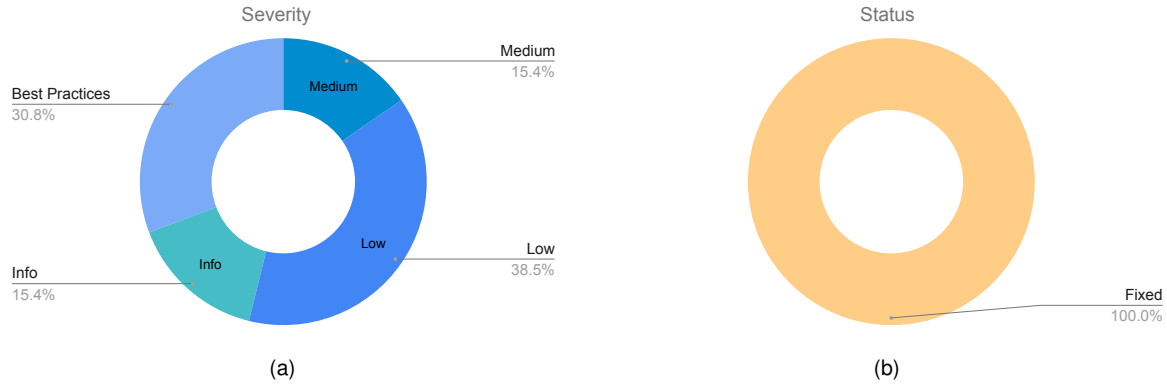


**Fig. 1: Distribution of issues: Critical** (0), **High** (0), **Medium** (2), **Low** (5), **Undetermined** (0), **Informational** (2), **Best Practices** (4). **Distribution of status: Fixed** (13), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | Feb. 07, 2023 |
| **Final Response from Client** | Apr. 03, 2023 |
| **Final Report** | Apr. 10, 2023 |
| **Methods** | Manual Review, Automated Analysis |
| **Repository** | Fileverse |
| **Commit Hash (Initial Audit)** | f9d608b6bfb4e5656139086709d8903508cb8fad |
| **Commit Hash (Reaudit)** | fcadc8abb4784a073a11185ca06ff079929db8f8 |
| **Documentation** | Docs Folder |
| **Documentation Assessment** | Medium |
| **Test Suite Assessment** | High |

# 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | contracts/FileversePortal.sol | 309 | 182 | 58.9% | 47 | 538 |
| 2 | contracts/FileversePortalRegistry.sol | 94 | 57 | 60.6% | 12 | 163 |
| 3 | contracts/structs/PortalKeyVerifiers.sol | 9 | 1 | 11.1% | 1 | 11 |
| | **Total** | **412** | **240** | **58.3%** | **60** | **712** |

# 3 Assumptions

The prepared security review is based on the following assumptions:

- The **Trusted Forwarder** is a contract trusted by `FileversePortalRegistry` and `FileversePortal` contracts to correctly check signatures and nonces before forwarding the request from users and correctly implements **EIP2771**;
- The **Trusted Forwarder** is an optional feature of the protocol;

# 4 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Function `editFile(...)` can create new files | Medium | Fixed |
| 2 | State variable `memberCount` is not ensured to have proper value | Medium | Fixed |
| 3 | Constructor parameters are not validated | Low | Fixed |
| 4 | Files can have an invalid verifier `version` | Low | Fixed |
| 5 | Iteration over an unbounded array may cause the transactions to revert | Low | Fixed |
| 6 | Lack of a two-step process for transferring ownership | Low | Fixed |
| 7 | Removed collaborator remains as a member | Low | Fixed |
| 8 | Any type of file can have the `gateIPFSHash` field | Info | Fixed |
| 9 | Owner can renounce ownership | Info | Fixed |
| 10 | Avoid using floating pragmas | Best Practices | Fixed |
| 11 | State variable could be declared constant | Best Practices | Fixed |
| 12 | Unused mapping `_allPortalIndex` | Best Practices | Fixed |
| 13 | The function `setupCollaborators(...)` introduces unnecessary complexity | Best Practices | Fixed |

# 5 System Overview

Fileverse enables users to create portals by calling the function `mint(...)`. Users can also self-deploy the portal contract to manage their portal on other chains. **The audit is based on two contracts:** a) *FileversePortalRegistry.sol*; and b) *FileversePortal.sol*. The *FileversePortalRegistry.sol* contract is the first one that users interact with. Particularly, it inherits Openzeppelin's `ReentrancyGuard` and `ERC2771Context` contracts. `ERC2771Context` is an implementation of the `EIP2771` specification which is a standardized approach to sending gasless transactions. The *FileversePortal.sol* is the portal contract that is deployed by *FileversePortalRegistry* contract. It also inherits Openzeppelin's `ERC2771Context` contract. The contracts are described below in accordance with the structural diagram presented in Fig.2.

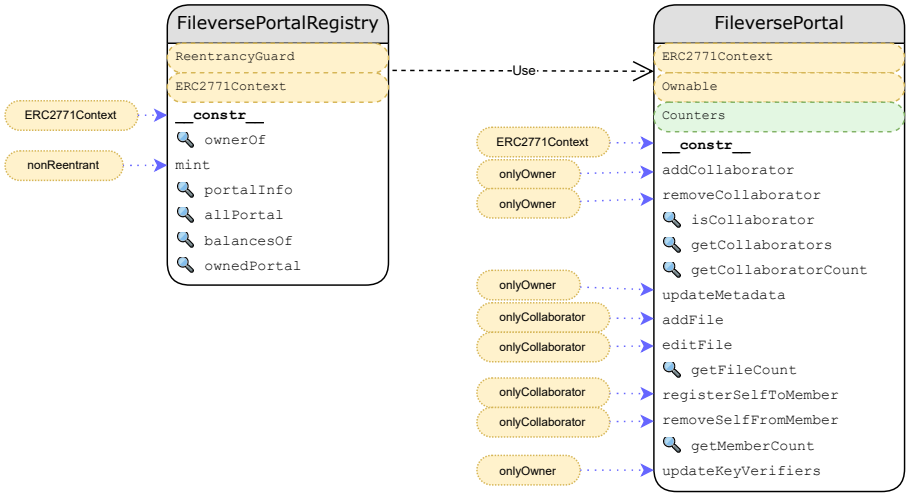


**Fig. 2: Structural Diagram of the Contracts**

## 5.1 FileversePortalRegistry.sol

The admin deploys this contract only once. `FileversePortalRegistry` accepts meta-transactions through a **Trusted Forwarder**. Basically, users interact with it to create a portal by calling the function `mint(...)`. The Registry does not receive special access and it is used as read-only directory for deployed portals. Particularly, the function `mint(...)` uses the `FileversePortal` contract and `PortalKeyVerifiers` library. The `FileversePortal` contract creates a portal and assigns it to the owner (who calls this function) while `PortalKeyVerifiers` holds key verifiers.

**Portal Creation -** The `mint(...)` function calls the constructor of the `FileversePortal` contract to deploy the portal contract. This constructor receives the **Truster Forwarder** address and key verifiers. After creating the portal, the `mint(...)` function stores a `tokenId`, which represents the global index of the portal in the Registry. The `FileversePortalRegistry` contract has a struct named `Portal` to hold the portal `address`, `index`, and `tokenId` (the global index). The struct is reproduced below.

```
struct Portal {
    address portal;
    uint256 index;
    uint256 tokenId;
}
```

This contract has the following public function.

- `mint(...)`

In addition to this function, the contract also provides five `getters` which are not described here. These `getters` allow retrieving data such as the owner address of the portal, a list of portals that are owned by the address `_owner`, among others. Fig. 2 lists all the functions implemented in the contract `FileversePortalRegistry.sol`.

## 5.2 FileversePortal.sol

`FileversePortal` is the portal contract that is deployed by the registry contract named `FileversePortalRegistry`. In case users want to manage their portal on other chains they can use this contract to deploy it by themselves setting proper values to the constructor. The contract consists of the following file entities:

- **Collaborators.** The portal includes collaborators that are users allowed to call functions for adding and editing files to the contract. The contract implements a circular linked list to store collaborators.

- **Members.** The portal also has members. Basically, members are collaborators that self-register their DIDs in the contract. Multiple members may have the same **viewDid** and **editDid**.
- **Key Verifiers.** These are `sha256` hash of the portal and member keys.
- **Files.** The portal includes files that are generated by the client side. **Files** with the same attributes may be registered in the Portal multiple times and any collaborator may edit any **File** struct stored in the portal. This entity consists of five components, as described below:

```
Metadata: metadataIPFSHash is a JSON
Content:  contentIPFSHash
Gate:     gateIPFSHash is a JSON that consists of IPFS Hash with json in the following format:
    { gateId: "", params: ["erc721:"] }

File_Type: Determines how the file will be stored and which key material will be used to encrypt the file. It is an Enum
 ↪  with the following values:
    enum FileType {
        PUBLIC,
        PRIVATE,
        GATED,
        MEMBER_PRIVATE
    }
Key_Verifier_Version: Which version of key was used to encrypt the file content Getter Functions dealing with this. The
 ↪  functions below depend on version passed as parameter:

    Anyone: files(uint256 fileId):
        Returns the full file object with proper data Setter Functions dealing with this

    OnlyCollaborator: function addFile(string calldata _metadataIPFSHash,string calldata _contentIPFSHash,string
 ↪  calldata _gateIPFSHash,FileType filetype,uint256 version)

    OnlyCollaborator: function editFile(uint256 fileId, string calldata _metadataIPFSHash,string calldata
 ↪  _contentIPFSHash,string calldata _gateIPFSHash,FileType filetype,uint256 version)
```

The `FileversePortal.sol` contract has two structs, `Member` and `File`. The `Member` struct holds info about users that register their edit and view DIDs in the contract: `viewDID: ED25519 key` and `editDID: ED25519 key`. The struct is reproduced below:

```
struct Member {
    string viewDid;
    string editDid;
}
```

The struct `Files` holds the IPFS hash of the metadata file, the IPFS hash of the file's content, the IPFS hash of the gate file, the type of the file (public, private, gated, member_private), and a value describing which version of the key was used to handle the file.

```
struct File {
    string metadataIPFSHash;
    string contentIPFSHash;
    string gateIPFSHash;
    FileType fileType;
    uint256 version;
}
```

This contract has the following public functions.

- `addCollaborator()`
- `removeCollaborator()`
- `updateMetadata()`
- `addFile()`
- `editFile()`
- `registerSelfToMember()`
- `removeSelfFromMember()`
- `updateKeyVerifiers()`
- `renounceOwnership()`
- `transferOwnership()`

In addition to these functions, the contract also provides several `getters` which are not described here. These `getters` allow users to check if an account is a collaborator or not, get a list of the portal collaborators, among others.

# 6 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

**Likelihood** is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

**Impact** is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

| | | **Severity Risk** | | |
|---|---|---|---|---|
| **Impact** | **High** | Medium | High | Critical |
| | **Medium** | Low | Medium | High |
| | **Low** | Info/Best Practices | Low | Medium |
| | **Undetermined** | Undetermined | Undetermined | Undetermined |
| | | **Low** | **Medium** | **High** |
| | | **Likelihood** | | |

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

# 7 Issues

## 7.1 [Medium] Function `editFile(...)` can create new files

**File(s)** : `FileversePortal.sol`

**Description** : The function `editFile(...)` does not have any checks to ensure that `fileId` exists. As a result, collaborators can edit files that don't exist (i.e: `fileId > _fileIdCounter.current()` ). When editing a file that does not exist, it effectively creates a new file without incrementing the `_fileIdCounter` . This can lead to an inconsistency between the number of files according to `_fileIdCounter` and the actual number of files in the `files` mapping. The function is reproduced below:

```
function editFile(...) public onlyCollaborator {
    require(bytes(_metadataIPFSHash).length != 0, "FV206");
    require(bytes(_contentIPFSHash).length != 0, "FV206");

    files[fileId] = File(
        _metadataIPFSHash,
        _contentIPFSHash,
        _gateIPFSHash,
        filetype,
        version
    );
    emit EditedFile(
        fileId,
        _metadataIPFSHash,
        _contentIPFSHash,
        _gateIPFSHash,
        _msgSender()
    );
}
```

**Recommendation(s)** : Consider adding a check to ensure that the collaborator cannot edit a file that doesn't exist.

**Status** : Fixed

**Update from the client** : Added check for `fileId` to be always less than `fileIdCounter` .

Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.2 [Medium] State variable `memberCount` is not ensured to have proper value

**File(s)** : `FileversePortal.sol`

**Description** : The state variable `memberCount` stores the number of members of the portal. `memberCount` can be modified through the functions `registerSelfToMember(...)` and `removeSelfFromMember(...)` . These functions can be called multiple times from the same calling address. After the first call, the `memberCount` variable will continue to increment/decrement without any changes to the `members` mapping. The functions are reproduced below:

```
function registerSelfToMember(...) public onlyCollaborator {
    require(bytes(viewDid).length != 0, "FV201");
    require(bytes(editDid).length != 0, "FV201");
    address sender = _msgSender();
    ////////////////////////////////////////////////////
    // @audit No check if members[sender] is already set //
    ////////////////////////////////////////////////////
    members[sender] = Member(viewDid, editDid);
    memberCount++;
    emit RegisteredMember(sender);
}

function removeSelfFromMember() public onlyCollaborator {
    address sender = _msgSender();
    ////////////////////////////////////////////////////////
    // @audit No check if members[sender] is already removed //
    ////////////////////////////////////////////////////////
    delete members[sender];
    memberCount--;
    emit RemovedMember(sender);
}
```

This can lead to an inconsistency between `memberCount` and the actual number of members inside the `members` mapping. A malicious collaborator could call these functions repeatedly to modify the `memberCount` to an incorrect value. Note that if `memberCount == 0`, no other collaborator would be able to call `removeSelfFromMember(...)` since operation `memberCount-` will always revert because of overflow.

**Recommendation(s)** : Consider checking if the sender is already set in the function `registerSelfToMember(...)` and checking if the sender was already removed in `removeSelfFromMember(...)`.

**Status** : Fixed

**Update from the client** : Refactored code and made terminology change from member mapping to `collaboratorKeys` Mapping.

Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.3 [Low] Constructor parameters are not validated

**File(s)** : `FileversePortal.sol` `FileversePortalRegistry.sol`

**Description** : Some constructor parameters in `FileversePortalRegistry` and `FileversePortal` contracts are not validated. The immutable state variable `_trustedForwarder` in `FileversePortalRegistry` can be set as `address(0)` by mistake and as it is immutable it cannot be changed.

```
//FileversePortalRegistry.sol
address private immutable trustedForwarder;

constructor(address _trustedForwarder) ERC2771Context(_trustedForwarder) {
    trustedForwarder = _trustedForwarder;
}
```

The constructor in `FileversePortal` is also missing a zero address check. As users can self-deploy the `FileversePortal` contract to manage their portal on other chains, the `owner` parameter must be validated, since it can be set as `address(0)` by mistake and no collaborators could be added or removed.

```
//FileversePortal.sol
constructor(
    string memory _metadataIPFSHash,
    string memory _ownerViewDid,
    string memory _ownerEditDid,
    address owner,
    address _trustedForwarder,
    PortalKeyVerifiers.KeyVerifier memory _keyVerifier
) ERC2771Context(_trustedForwarder) {
    ////////////////////////////////////////////////
    // @audit Validate owner since this           //
    //        contract can be self-deployed       //
    ////////////////////////////////////////////////
    metadataIPFSHash = _metadataIPFSHash;
    address[] memory _collaborators = new address[](1);
    _collaborators[0] = owner;
    setupCollaborators(_collaborators);
    _transferOwnership(owner);
    setupMember(owner, _ownerViewDid, _ownerEditDid);
    ...
}
```

**Recommendation(s)** : Ensure that the parameters `_trustedForwarder` and `owner` are not `address(0)` .

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.4 [Low] Files can have an invalid verifier `version`

**File(s)** : `FileversePortal.sol`

**Description** : When using `addFile(...)` or `editFile(...)` , the `version` argument is used to indicate which version of the key verifier should be used to handle the file. The latest key verifier is tracked using the counter `_keyVerifierCounter` . When adding or editing a file, the argument `version` is not checked to be a known key verifier version (i.e: up to the latest key verifier version). This allows collaborators to pass an invalid key verifier version through the `version` argument when adding or editing a file. The functions are shown below:

```
function addFile(...) public onlyCollaborator {
    require(bytes(_metadataIPFSHash).length != 0, "FV206");
    require(bytes(_contentIPFSHash).length != 0, "FV206");

    uint256 fileId = _fileIdCounter.current();
    _fileIdCounter.increment();
    files[fileId] = File(_metadataIPFSHash, _contentIPFSHash, _gateIPFSHash, filetype, version);
    emit AddedFile(fileId, _metadataIPFSHash, _contentIPFSHash, _gateIPFSHash, _msgSender());
}

function editFile(...) public onlyCollaborator {
    require(bytes(_metadataIPFSHash).length != 0, "FV206");
    require(bytes(_contentIPFSHash).length != 0, "FV206");

    files[fileId] = File(_metadataIPFSHash, _contentIPFSHash, _gateIPFSHash, filetype, version);
    emit EditedFile(fileId, _metadataIPFSHash, _contentIPFSHash, _gateIPFSHash, _msgSender());
}
```

**Recommendation(s)** : Ensure that the `version` argument represents a valid key verifier version when adding or editing a file.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.5 [Low] Iteration over an unbounded array may cause the transactions to revert

**File(s)** : `FileversePortalRegistry.sol` , `FileversePortal.sol`

**Description** : The contracts `FileversePortalRegistry` and `FileversePortal` contain functions with a loop iterating over an unbounded dynamic array, we list these functions below:

- `allPortal(...)` ;
- `ownedPortal(...)` ;

As an array grows, it can reach a large length. Consequently, any iteration through it could exceed the maximum amount of gas in the block. Then the contract can be compromised by having this function never working again.

**Recommendation(s)** : Consider implementing a pagination strategy, i.e: define a start index and the pagination length, which would only return a part of a collection.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.6 [Low] Lack of a two-step process for transferring ownership

**File(s)** : `FileversePortal.sol`

**Description** : The `FileversePortal` contract inherits from OpenZeppelin's Ownable contract, which provides a one-step ownership transfer. Transferring ownership in a single step is error-prone and can severely harm the portal if a mistake happens.

**Recommendation(s)** : We suggest implementing a two-step process for transferring ownership, such as the propose-accept scheme. You can check the `Ownable2Step.sol` contract from OpenZeppelin.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.7 [Low] Removed collaborator remains as a member

**File(s)** : `FileversePortal.sol`

**Description** : Every collaborator can register and unregister themselves as a member. However, if a user loses the collaborator role, they will remain as a member. This violates an invariant of the protocol that each member must be a collaborator. Also, the member that is not a collaborator can't remove their member role.

**Recommendation(s)** : Consider updating the member role when removing the collaborator role for a given address.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.8 [Info] Any type of file can have the `gateIPFSHash` field

**File(s)** : `FileversePortal.sol`

**Description** : The struct `File` is declared as:

```
struct File {
    string metadataIPFSHash;
    string contentIPFSHash;
    string gateIPFSHash;
    FileType fileType;
    uint256 version;
}
```

The values for the `FileType` enum can be seen below.

```
enum FileType {
    PUBLIC,
    PRIVATE,
    GATED,
    MEMBER_PRIVATE
}
```

The field `gateIPFSHash` from the `File` struct is only used when the `fileType` is `GATED` . However, there is no check when adding or editing a file to ensure that files of different types don't have this field populated.

**Recommendation(s)** : Consider ensuring that only `GATED` files have this field populated.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.9 [Info] Owner can renounce ownership

**File(s)** : `FileversePortal.sol`

**Description** : The `FileversePortal` contract inherits ownable features through the OpenZeppelin `Ownable.sol` contract. This contract implements the function `renounceOwnership(...)` which can be called at any point by the owner. Once this function is called, the owner renounces the ownership and the contract will be without an owner. Consequently, all `onlyOwner` functions will not be able to be called anymore. The functions that are only available to the owner are listed below:

- `addCollaborator(...)` ;
- `removeCollaborator(...)` ;
- `updateMetadata(...)` ;
- `updateKeyVerifiers(...)` ;

**Recommendation(s)** : If this is an intentional part of the contract design consider clarifying this in the documentation, otherwise we recommend overriding the `renounceOwnership(...)` function to disable it.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.10 [Best Practice] Avoid using floating pragmas

**File(s)** : `FileversePortal.sol` , `FileversePortalRegistry.sol`

**Description** : While floating pragmas make sense for libraries to allow them to be included with multiple different versions of applications, it may be a security risk for application implementations.

**Recommendation(s)** : Consider using a strict Solidity version pragma with version `0.8.16` . The recommendation takes into account: a) risks related to recent releases; b) risks of complex code generation changes; c) risks of new language features; d) risks of known bugs.

```
- pragma solidity ^0.8.9;
+ pragma solidity 0.8.16;
```

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.11 [Best Practice] State variable could be declared constant

**File(s)** : FileversePortalRegistry.sol

**Description** : The state variable `name` in the `FileversePortalRegistry` contract does not change. Any predetermined values that do not change throughout the contract lifetime should be set to `constant` . The code is shown below:

```
string public name = "Fileverse Portal Registry";
```

**Recommendation(s)** : Consider changing the state variable `name` to `constant` .

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.12 [Best Practice] Unused mapping `_allPortalIndex`

**File(s)** : FileversePortalRegistry.sol

**Description** : The contract `FileversePortalRegistry` contains a `private` mapping called `_allPortalIndex` . The declaration of this variable can be seen in the code snippet below.

```
...
mapping(address => mapping(uint256 => address)) private _ownedPortal;
// Array with all token ids, used for enumeration
address[] private _allPortal;
// Mapping from FNS to position in the allFNS array
mapping(address => uint256) private _allPortalIndex;
// Mapping from address of portal to Portal Data
mapping(address => Portal) private _portalInfo;
// address of trusted forwarder
address private immutable trustedForwarder;
...
```

The `_allPortalIndex` state variable is never read inside the contract and its visibility is internal, so it is not reachable outside the contract. The only place where this variable is used is the `_mint` function, which can be seen in the code snippet below.

```
function _mint(...) internal {
    require(_ownerOf[_portal] == address(0), "FV200");
    uint256 length = _balances[_owner];
    uint256 _allPortalLength = _allPortal.length;
    _ownerOf[_portal] = _owner;
    _allPortal.push(_portal);
    _ownedPortal[_owner][length] = _portal;
    _allPortalIndex[_portal] = ++_allPortalLength;
    _portalInfo[_portal] = Portal(_portal, length, _allPortalLength);
    ++length;
    _balances[_owner] = length;
}
```

As shown in the code snippet, the only use of the variable is for storing information that is never read.

**Recommendation(s)** : If the contract `FileversePortalRegistry` is not meant to be inherited, consider removing unused variables from the code to reduce complexity and save gas.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

## 7.13 [Best Practices] The function `setupCollaborators(...)` introduces unnecessary complexity

**File(s)** : FileversePortal.sol

**Description** : The function `setupCollaborators(...)` is responsible for adding the owner of the portal as a collaborator, and is only called once in the constructor. However, the function takes a list of addresses to add as an argument and adds those addresses in a loop. We reproduce the function below:

```
function setupCollaborators(...) internal { // @audit this function doesn't make sense, in constructor they only set up one collaborator
    // Initializing Subdomain collaborators.
    uint256 len = _collaborators.length;
    require(len != 0, "FV202");
    address currentCollaborator = SENTINEL_COLLABORATOR;

    for (uint256 i; i < len; ++i) {
        // Owner address cannot be null.
        address collaborator = _collaborators[i];
        require(
            collaborator != address(0) &&
                collaborator != SENTINEL_COLLABORATOR &&
                collaborator != address(this) &&
                currentCollaborator != collaborator,
            "FV203"
        );
        // No duplicate collaborators allowed.
        require(collaborators[collaborator] == address(0), "FV204");
        collaborators[currentCollaborator] = collaborator;
        currentCollaborator = collaborator;
    }
    collaborators[currentCollaborator] = SENTINEL_COLLABORATOR;
    collaboratorCount = len;
}
```

We also reproduce part of a constructor code where the function is used:

```
address[] memory _collaborators = new address[](1);
_collaborators[0] = owner;
setupCollaborators(_collaborators);
```

**Recommendation(s)** : Since the function is only called once with an array of size `1` , consider simplifying `setupCollaborators(...)` to handle one `address` argument rather than an `address` array. This will reduce code complexity and increase readability.

**Status** : Fixed

**Update from the client** : Fixed in commit `fcadc8abb4784a073a11185ca06ff079929db8f8` .

# 8 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a README.md but also using code as documentation (to write clear code), diagrams, websites, research papers, videos, and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit.

The **Fileverse** team provided three documents to assist the audit process, describing (a) `PortalKeyVerifiers` library, the `FileversePortal` and `FileversePortalRegistry` contracts; (b) a diagram describing the flow of the portals' creation; and (c) instructions to run the test suite. These documents covered the most common terms used in the source code, explanations for the core business logic and functions flow, and guides for running tests. By reading the whole documentation suite, we could get a proper understanding on how *Fileverse Portal Registry* and *Fileverse Portal* contracts should operate.

The provided documentation is a complete source of resources for developers and auditors, and we are satisfied with it. The codebase has sufficient inline comments, helping the audit team to understand the function flow and to detect some issues. **Fileverse** could benefit from clearly stating each use case, allowing the audit team to investigate if all use cases are working as expected.

## 8.1 Documentation inconsistencies

Along this investigation, we noticed some areas in the code where the implementation differs from the specification and/or inline comments.

### 8.1.1 Incorrect documentation in function `addCollaborator(...)`

The function `addCollaborator(...)` is described with inline documentation, which does not match its implementation. We reproduce the function with inline documentation below:

```
/**
 * @notice The function adds a collaborator to the list of collaborators
 * @dev If the collaborator is not the smart contract address and is not
 * sentinel collaborator and the collaborator is not the zero address,
 * then add it.
 * It also emits an event AddedCollaborator with params account and by addresses
 * It can be called only by the owner of the portal
 * @param collaborator - The address of the collaborator is to be added.
 */
function addCollaborator(address collaborator) public onlyOwner {
    require(
        collaborator != address(0) &&
            collaborator != SENTINEL_COLLABORATOR &&
            collaborator != address(this),
        "FV203"
    );
    // No duplicate owners allowed.
    require(collaborators[collaborator] == address(0), "FV204");
    collaborators[collaborator] = collaborators[SENTINEL_COLLABORATOR];
    collaborators[SENTINEL_COLLABORATOR] = collaborator;
    collaboratorCount++;
    emit AddedCollaborator(collaborator, _msgSender());
}
```

The inline documentation states that: *"If the collaborator is not the smart contract address [...] then add it"* however the function only checks if `collaborator` is not zero, `collaborator` is not `SENTINEL_COLLABORATOR`, and if `collaborator` is not the contract `FileversePortal`, but there is no check if address `collaborator` is not a smart contract address.

### 8.1.2 Function `_mint(...)` visibility inconsistency

The inline documentation of the function `_mint(...)` states that its visibility is `private`, but the visibility of the function is `internal`.

```
    /**
     * @notice A private function called by the public function `mint`.
     * It is used to create a new portal and assign it to the owner of the
     * contract.
     * @param _owner - The address of the owner
     * @param _portal - The address of the portal
     */
    function _mint(address _owner, address _portal) internal {
```

# 9 Test Suite Evaluation

## 9.1 Contracts Compilation Output

```
$ npx hardhat compile
Downloading compiler 0.8.9
Generating typings for: 8 artifacts in dir: typechain for target: ethers-v5
Successfully generated 11 typings!
Compiled 8 Solidity files successfully
```

## 9.2 Tests Output

```
$ npx hardhat test
No need to generate any newer typings.

  Fileverse Portal: Owner
    should be able to deploy with correct number of parameters
    should be collaborator by default
    should be member by default
    should be able add collaborator by owner
    should be able add and remove collaborator by owner
    should be able to update metadata
    should be able to add file
    should be able to edit file
    should be able to deploy with correct key verifiers
    should be able to update the key verifiers of the deployed portal
```

```
  Fileverse Portal: Collaborator
     should be able to register self as member
     should be able to register and remove self as member
     should be able to add file
     should be able to edit file

  Fileverse Portal: Fake Collaborator
     should not be able to register self as member
     should not be able to remove self as member
     should not be able to add file
     should not be able to edit file

  Fileverse Portal Registry
     should be able to deploy with correct number of parameters
     should be able to deploy with correct empty state
     should have the same trusted forwarder
     should be able to mint a fileverse portal with proper parameters
     should not be able to mint a fileverse portal with improper parameters
     should be able to fire a mint event on creating a fileverse portal
     should be able to create two fileverse portal
     should be able to create two fileverse portal by two addresses
     should have the sender of the txn as owner
     should have functional getter functions as owner
     should have functional getter functions as reader

  Fileverse Portal Registry: Deployed Portal
     should be able to deploy with correct initial state
     should be able to deploy with correct key verifiers
     should be able to update the key verifiers of the deployed portal
     should be able to deploy with correct collaborator set

  33 passing (5s)
```

## 9.3 Code Coverage

```
npx hardhat coverage
```

The relevant output is presented below.

```
-----------------------------|----------|----------|----------|----------|----------------|
File                         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-----------------------------|----------|----------|----------|----------|----------------|
 contracts/                  |     98.8 |    59.38 |    96.55 |    99.16 |                |
  FileversePortal.sol        |    98.41 |       60 |    95.24 |    98.89 |            536 |
  FileversePortalRegistry.sol |      100 |       50 |      100 |      100 |                |
 contracts/structs/          |      100 |      100 |      100 |      100 |                |
  PortalKeyVerifiers.sol     |      100 |      100 |      100 |      100 |                |
-----------------------------|----------|----------|----------|----------|----------------|
All files                    |     98.8 |    59.38 |    96.55 |    99.16 |                |
-----------------------------|----------|----------|----------|----------|----------------|
```

## 9.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

# 10 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

**Blockchain Security:** At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

**Blockchain Core Development:** Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

**DevOps and Infrastructure Management:** Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

**Cryptography Research:** At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

**Smart Contract Development & DeFi Research:** Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

**Our suite of L2 tooling:** Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

**Learn more about us at nethermind.io**.

### Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.