Centrifuge

Security Assessment

March 29, 2019

Prepared For:
Philip Stehlik | *Centrifuge*
philip@centrifuge.io

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Sam Moelius | *Trail of Bits*
sam.moelius@trailofbits.com

Changelog:
March 29, 2019: Initial report delivered to Centrifuge
July 9, 2019: Added re-test results and reworded last three issues.

Executive Summary

From March 18th through March 29th, 2019, Centrifuge engaged Trail of Bits to review the security of the Centrifuge node and smart contracts. Trail of Bits conducted this assessment over the course of four person-weeks with three engineers working from the Centrifuge git repositories.

Trail of Bits looked for flaws in the smart contracts and the Go code using static analysis, fuzzing and manual review. During the first week, emphasis was placed on high-level architectural considerations, the anchor registry, and the NFT token contracts. The second week was focused on the P2P protocol and the precise proof library. Additionally, we developed several fuzzing tests for internal functions using go-fuzz and libFuzzer. Appendix C documents these deliverables.

Trail of Bits identified 30 findings ranging from high-severity to informational. Several of the issues require malicious collaborators, and can lead to invalid updating and anchoring of documents. Additionally, we found issues in the precise proof library, including incorrect Merkle tree creation due to a misused standard library call. We also found issues at the P2P level, including the collaborators list leaking through an error message received by untrusted users.

The number and severity of the discovered vulnerabilities are expected for a system at Centrifuge's development stage. Trail of Bits commends Centrifuge for organizing an assessment on a work in progress state of the protocol. We acknowledge that some of the findings would have been mitigated with upcoming features. Trail of Bits identified the following axes of improvement for the protocol:

- *Reduce the trust required between collaborators of a document.* The protocol requires the collaborators of a given document to trust each other entirely. A malicious or compromised collaborator could inflict critical damage.
- *Improve the signature-verification procedure.* The signature procedure was designed to be only a proof of receipt and does not indicate the signer's validation. The benefit of the signature mechanism is therefore limited and users might misunderstand its purpose.
- *Provide secure key management and authentication.* It will be required prior to a deployment in a real context of the protocol.
- *Document the assumptions and limitations of the protocol.* The protocol requires users to be aware of the core limitations and the expected behaviors to use the system reliably.

Centrifuge should fix all the issues, add static analyzers and fuzzing to the development process, and consider the improvements stated above. Finally, we recommend additional review prior to the production deployment of the system.

Project Dashboard

Application Summary

Name	go-centrifuge, centrifuge-ethereum-contracts, precise-proofs, centrifuge-protobufs
Version	187ba86154e4298f52b294310b3e4f42c4e9b0ee c5e55d16c4cfe058ca641d2a746a665f277f0c9f 13d3af957299c614237c42cdc331f7acd7c7d201 864a8ef4039324cebf3f23df115f50db12009d4c
Type	Go and Solidity smart contracts
Platforms	Ethereum, Go, Protobuf

Engagement Summary

Dates	March 18th- 29th, 2019
Method	Whitebox
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	7	■■■■■■■
Total Medium-Severity Issues	6	■■■■■■
Total Low-Severity Issues	4	■■■■
Total Informational-Severity Issues	5	■■■■■
Total Undetermined-Severity Issues	8	■■■■■■■■
Total	30	

Category Breakdown

Access Controls	3	■■■
Timing	2	■■
Data Validation	13	■■■■■■■■■■■■■
Data Exposure	5	■■■■■

Patching	3	■■■
Undefined Behavior	1	■
Auditing and Logging	1	■
Denial of Service	1	■
Error Reporting	1	■
Total	30	

Engagement Goals

The engagement was scoped to provide a security assessment of the Centrifuge node and its smart contracts.

The major security concern was to ensure that the protocol is sound and the implementation of the hashing, signatures, signature checking, document state validation, interactions with Ethereum are secure.

Specifically, we sought to answer the following questions:

Smart contracts

- Are the keys correctly managed?
- Is the NFT minting correct?
- Can a token be minted two times?

Precise-proof

- Are all the fields uniquely identified by a property?
- Can a valid proof of a property always be produced, regardless of its value?
- Can a proof be reused on multiple fields of the same core document?
- Are the salts always correctly re-generated?
- Can the merkle tree/proof generation crash?

go-centrifuge

- Can a remote user gain read or write access to a non-authorized document?
- Can a user crash an external node?
- Are the validators enough to ensure that the documents are valid?
- Can an attacker abuse the P2P wire message protocol?

Since the system is a work in progress, all the areas not mentioned above were not targeted by the assessment, including the local storage, the REST API, and the callbacks via webhooks.

Coverage

Smart contract. Trail of bits reviewed the smart contracts using manual review and Slither, the static analyzer. We looked for flaws in the authorization and the handling of keys. We checked the Merkle tree verification library and its correct usage in the other contracts. We also reviewed the minting process. We did not review the dependencies' code (including the openzepellin ERC721 contract, and the zos dependencies), and did not validate the unit tests.

Precise-proof. Trail of bits reviewed the precise-proof library using manual review and fuzzing. A focus was placed to find the creations of incorrect Merkle trees, and to compromise the proofs verifications. We also looked for code panics, with a lower priority. Efforts were made to find collisions in the leaves names. We reviewed the library using the provided formats, and briefly consider formats that were not in the provided with the Centrifuge codebase. We did not fully investigate the impact of malformed new protobuf formats, and did not validate the unit tests.

Go-centrifuge. Trail of bits reviewed go-centrifuge using manual review and fuzzing. We focused our efforts on the network interaction, and on the functionalities reachable by external peers. We looked for bypasses of privilege, leaks of information, and crashes. We reviewed how the node handles external requests and checked the correct access validations. We briefly reviewed the smart contract interactions, in particular with the identity and the NFT contracts . We investigated with a low priority the Kademlia protocol usage, the REST API, and the key management of the node. We did not validate the unit tests.

Due to the time constraint and the number of bugs found, we expect other bugs to be present in the covered areas.

Smart contracts Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Check that the Management and Action keys are not revoked prior to their usage in KeyManager and Identity.** Revoked keys can still be used, preventing mitigation of a compromised key.

❑ **Disallow the use of the `commit` function without having a valid `preCommit`.** Direct calls to `commit` allow for a race condition wherein an attacker may steal the anchor id prior to its anchoring.

❑ **When committing an anchor, ensure the validity of the document root by checking `sha256(signingRoot+signatureRoot) == documentRoot`.** The current check allows for an incorrect document root to be committed.

❑ **Check the contract's existence prior to the call in the Identify contract, and consider adding a separate function to transfer ethers.** The lack of contract existence check may lead to unexpected behavior for the caller.

❑ **Prevent anchors with a Merkle tree equal to zero from being committed.** Anchoring a Merkle tree with root equal to zero allows for multiple anchoring on the same anchor id.

❑ **Check for a minimal length of proofs for all calls to `verifySha256` and `verify`.** Empty lists of proofs will allow to bypass some of the Merkle tree verification.

❑ **Update the smart contract build process dependencies to the latest version wherever possible.** Out of date dependencies might lead to missed critical bug fixes.

❑ **Measure the contracts' gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.** The Solidity compiler has a history of compilation-related bugs, which should be carefully considered.

❑ **Remove the usage of the ABI encoder V2 if the contracts are meant to be production deployed in the short term.** The encoder is still experimental and not ready for production code.

❑ **Implement an onchain mapping from document IDs to token IDs, and check that a token cannot be minted multiple times. Prevent the document ID to be changed**

during update. Multiple tokens can be minted for the same document if the token ID associated to the document is changed.

Long Term

❑ **Add unit tests to ensure the correct authorization schema of the Identity contract, including scenarios where the keys are compromised.** Thorough unit tests on the authorization schema will have prevent issues like TOB-Centrifuge-001.

❑ **Document the race condition risk on `preCommit` and ensure that users are aware of it. Closely monitor it by inspecting Anchor events.** A race condition attack on `preCommit` would temporarily spam the anchor registry.

❑ **Carefully review the Solidity documentation.** In particular, any section that contains a warning must be carefully understood since it may lead to unexpected or unintented behavior.

❑ **Identify the smart contract properties that should always be true or false and test them using Manticore and Echidna.** Automated testing framework will help to identify ahead of time future bugs.

❑ **Add integration tests on all functions relying on the Merkle tree verification.** Unit tests should highlight failing scenarios, including empty and incorrect proofs.

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess its maturity.** The Solidity compiler has a history of compilation-related bugs, which should be carefully considered.

❑ **Monitor the development and adoption of Solidity ABIEncoderV2 to assess its maturity.** The encoder is still experimental and not ready for production code.

Precise-proofs Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Cast the call to `reflect.Len()` to uint64 prior calling `toBytesArray`.** `toBytesArray` requires fixed-size data and will return incorrect result otherwise.

❑ **Check the return error of all the calls to `binary.Write`.** `binary.Write` can fail, so its return must be checked for errors.

❑ **Prevent a leaf from being added if its name is already present in the tree.** Adding a leaf with an existing name will lead to incorrect proof generation and verification.

❑ **Handle the return of 0 in all the uses of `reflect.ValueOf`.** `ValueOf` can return zero and the return value will trigger a `nil` pointer dereference if not checked properly.

❑ **Document how to build a correct protobuf document format.** If the user is not aware of the expected format structure, it can generate a format with name collisions in the tree.

Long Term

❑ **Improve the unit tests coverage of the API usage of the precise proof library.** The unit tests must cover different API usage, including manual leaves add.

❑ **Consider implementing a tool to validate the correctness of a protobuf document format.** If the user does not follow the expected format, it can generate a schema leading to name collisions in the tree.

go-centrifuge Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Have the Centrifuge node listen for REST API calls on just the loopback interface (i.e., 127.0.0.1).** The REST API is exposed on all interfaces, allowing anyone to steal the node private key and password.

❑ **Ask the user to enter the password through a prompt.** Requiring the password on the command line and a configuration file is error prone and dangerous.

❑ **Ensure that created private keys are readable only by the user who created them.** World-readable private keys is dangerous. Instead, use file permissions `600` on Unix.

❑ **Return the same error message from a node in case of privilege access error and missing documents.** Attackers can determine if a node is a collaborator by reading the error message when requesting a document.

❑ **Check that `did` has at least one P2P key in `CurrentP2PKey`.** Interacting with a peer without P2P key will lead to a node crash.

❑ **Ensure that the timestamp of a document always increases during an update**. The update procedure allows documents to decrease their timestamp, leading to unexpected behavior.

❑ **Return the `validateDocumentAccess` error if the function call fails in `GetDocument`.** The error message is incorrect in case of invalid document access.

❑ **Update go build process dependencies to the latest versions wherever possible.** Out-of-date dependencies might lead to missed critical bug fixes.

❑ **Prevent the P2P routing leak of information when asking a collaborator location**. Peers asked for a route can deduce that the caller and the destination are collaborators. A solution is to communicate with other users randomly to introduce noise in the network.

❑ **Consider adding a random delay when returning an error message from a node.** Constant-time return values will prevent timing attacks on the system that reveal important information about the collaborators.

❑ **Disable automatic signing of documents. Require explicit approval from users using the REST API.** Automatic signing of documents can be abused to sign a malicious update.

❑ **Disable automatic addition of collaborator. Require explicit approval from users using the REST API.** Adding collaborators without their consent can be misused by malicious peers.

Long Term

❑ **Consider a method of communicating with the Centrifuge node that does not involve TCP/IP sockets.** The current model is risky and error-prone. Consider using Unix domain sockets, they are much easier to control access to.

❑ **Consider requiring the user to unlock their Ethereum accounts using geth's `--unlock` feature.** It will relieve `createconfig` to deal directly with Ethereum private keys.

❑ **Refactor Centrifuge node code to safely load and store private keys.** The current keys management is error prone and follows dangerous practices (TOB-Centrifuge-008, TOB-Centrifuge-009, TOB-Centrifuge-010).

❑ **Integrate the use of `npm audit` into the CI testing to avoid the use of vulnerable dependencies.** Out of date dependencies might lead to miss critical bug fixes.

❑ **Consider that information can be leaked through side channel when the nodes interact.** Even if an attacker cannot directly retrieve information, they might be able to deduce it by looking at how the system reacts.

❑ **Implement a user interface that allows a user to accept or reject individual document-signing requests.** Automatic signing of documents can be abused to sign malicious update.

❑ **Implement a user interface that allows a user to accept or reject individual collaboration requests.** Adding collaborators without their consent can be misused by malicious peers.

❑ **Add unit tests to cover all the possible message errors.** Unit tests must cover expected failure and check if the message errors are correctly generated.

❑ **Investigate alternative private routing solutions.** Information can be leaked if the routing algorithm is not designed to preserve the privacy of the collaborators.

General Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Clearly document the collaborator trust requirement to the users.** To work reliably, the system requires all collaborators to trust each other. Users who are unaware of this requirement will misuse Centrifuge.

❑ **Clearly document that the location of the leaf in the Merkle tree is not checked**. Users creating document checks must be aware of this restriction and take the necessary precautions when writing document checks.

❑ **Document the interactions and the expected fields of the Centrifuge components.** The lack of documentation on the component interactions and expected fields make its review difficult.

Long Term

❑ **Research and implement approaches to fork documents if there is a disagreement between the collaborators.** Forking a document will allow for recovery from an incorrect or malicious document update.

❑ **Research and implement approaches to verify the presence of the signature without revealing them.** Such a system would allow to verify the collaborators' signatures when anchoring a document.

❑ **Investigate solutions to confirm the leaf location in the Merkle tree, or alternative verification format.** The lack of leaf location check might lead to misuse or compromise of the verification.

❑ **Periodically run the fuzzer provided in Appendix C to identify code panics.** Fuzzing will help to detect code panics issues during the development.

❑ **Add gosec to the CI.** gosec is fast and can detect security issues with precision.

❑ **Investigate what core document fields should be automatically checked.** Core document fields must be carefully considered and automatically checked to prevent an incorrect update.

❑ **Implement a CI check to detect out-of-date go dependencies.** Out of date dependencies might lead to miss critical bug fixes.

Findings Summary

#	Title	Type	Severity
1	Revoked Management and Action keys can still be used	Access Controls	High
2	A race condition on commit can break document updates	Timing	Medium
3	User can commit anchor with incorrect Merkle root proof	Data Validation	Undetermined
4	Lack of contract existence check may lead to unexpected behavior	Data Validation	Medium
5	An anchor can be committed multiple times if its Merkle root is zero	Data Validation	Low
6	Merkle root verification can be done on empty proofs	Data validation	Undetermined
7	REST API is exposed on all interfaces	Data Exposure	High
8	Centrifuge “createconfig” requires password to be passed on command line	Data Exposure	High
9	Centrifuge “createconfig” stores a plaintext password in a configuration file	Data Exposure	High
10	Private keys are world-readable by default	Data Exposure	High
11	Smart contract build dependencies are not up to date	Patching	Informational
12	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
13	User can commit anchor without requiring collaborators’ signatures	Data Validation	Undetermined

14	Lack of location verification in the Merkle Tree is error prone	Data Validation	Undetermined
15	ABIEncoderV2 is not production-ready	Patching	Undetermined
16	Lack of fixed-size data cast on binary.Write call leads to incorrect leaves	Data Validation	Medium
17	Manually adding leaves can lead to name collisions in the Merkle tree	Data Validation	Medium
18	Nil pointer dereferencing can lead the precise proof library to panic when flattened by protobuf	Data Validation	Low
19	The lack of documentation on the protobuf format invites for incorrect document format	Auditing and Logging	Informational
20	Error messages can be used to leak the collaborators list	Data Validation	High
21	Timing attack can be used to leak the collaborators list	Timing	High
22	Centrifuge nodes sign documents without users' consent	Access Controls	Undetermined
23	Messages from an identity with no associated P2P keys leads to a node crash	Denial of Service	Medium
24	Updated timestamps can decrease	Data Validation	Low
25	Incorrect message error handling on invalid document access	Error Reporting	Low
26	libp2p dependencies are not up to date	Patching	Informational
27	Collaboration possibly leaked at the P2P level	Data Exposure	Undetermined

28	Documentation should indicate who the consumers of fields are	Data Validation	Informational
29	Consider requiring consent to become a collaborator	Access Controls	Informational
30	Anchor id update allows for multiple tokens mint for the same document	Data Validation	Medium

1. Revoked Management and Action keys can still be used

Severity: High
Difficulty: High
Type: Access Controls
Finding ID: TOB-Centrifuge-001
Target: KeyManager.sol, Identity.sol

Description
`Identify` defines two privileged key types: Management and Action. The revocation of a key of one of these types has no effect. As a result, it is not possible to mitigate a key compromise.

The Management key allows adding new keys of arbitrary purpose. The Action key allows the execution of arbitrary commands from the contract. `keyHasPurpose` checks if a key has a given purpose:

```
function keyHasPurpose(
  bytes32 key,
  uint256 purpose
)
public
view
returns (bool found)
{

  Key memory key_ = _keys[key];
  if (key_.purposes.length == 0) {
    return false;
  }
  for (uint i = 0; i < key_.purposes.length; i++) {
    if (key_.purposes[i] == purpose) {
      return true;
    }

  }

}
```
Figure 1: KeyManager.sol#L151-L169

Keys can be revoked using `revokeKey`. There is no mechanism to ensure that a Management or Action key was not revoked. As a result, revoking these keys will have no effect, and it will not be possible to recover from a compromise.

Exploit Scenario
Eve compromises Bob's Action key and registers an incorrect Merkle root. Bob revokes the key. Eve continues to take advantage of the compromised key.

Recommendation
Check that the keys are not revoked prior to their usage.

Add unit tests to ensure the correct authorization schema of the Identity contract, including scenarios where the keys are compromised.

2. A race condition on commit can break document updates

Severity: Medium
Difficulty: High
Type: Timing
Finding ID: TOB-Centrifuge-002
Target: *AnchorRepository.sol*

Description
The `commit` function was designed to store a document root in a given anchor but a race condition can block any future updates to the document.

The `commit` function can be called with or without previously calling `preCommit` as shown in Figure 1. In cases of using `preCommit` function, the code will verify that the caller is the `preCommit` call owner identified as `_preCommits[anchorId].identity`.

```
function commit(
   uint256 anchorIdPreImage,
   bytes32 documentRoot,
   bytes32[] calldata documentProofs
 )
 external
 {


   uint256 anchorId =
uint256(sha256(abi.encodePacked(anchorIdPreImage)));


   //not allowing to write to an existing anchor
   require(_commits[anchorId].docRoot == 0x0);


   // Check if there is a precommit and enforce it
   if (hasValidPreCommit(anchorId)) {
     // check that the precommit has the same _identity
     require(_preCommits[anchorId].identity ==
msg.sender,"Precommit owned by someone else");
     require(
       MerkleProof.verifySha256(
         documentProofs,
         documentRoot,
         _preCommits[anchorId].signingRoot
       ),
       "Signing root validation failed"
     );


   }


   _commits[anchorId] = Anchor(
     documentRoot,
     uint32(block.number)
   );
   emit AnchorCommitted(
     msg.sender,
     anchorId,
     documentRoot,
     uint32(block.number)
   );


 }
```
Figure 1: The `commit` function in AnchorRepository.sol#L45-L70

However, if the user did not use `preCommit`, the commit transaction can be frontrun by an attacker.

Exploit Scenario
Alice calls the `commit` function with the `anchorIdPreImage`, which requires her to update her document. Bob observes and frontruns the unconfirmed transaction using the same `anchorIdPreImage` value with a different `documentRoot`. As a result, Alice is unable to perform any updates in her document.

Recommendation
In the short term, disallow the use of the `commit` function without having a valid `preCommit`.

The use of `preCommit` is a partial mitigation since this transaction could be also be frontrun by the attacker but it is more expensive. Therefore, in the long term, document this risk and ensure that users are aware of it. Closely monitor it by inspecting Anchor events.

3. User can commit anchor with incorrect Merkle root proof

Severity: Undetermined
Type: Data Validation
Target: AnchorRepository.sol

Difficulty: High
Finding ID: TOB-Centrifuge-003

Description
`preCommit` reserves an anchor id with a signing root. To commit the anchor, the user must provide a document root containing the signing root. Users can commit malformed or malicious document roots by crafting a Merkle tree that contains the signing root in any leaf.

It is assumed that anchors that are committed after a `preCommit` call will follow Centrifuge's Merkle tree schema:

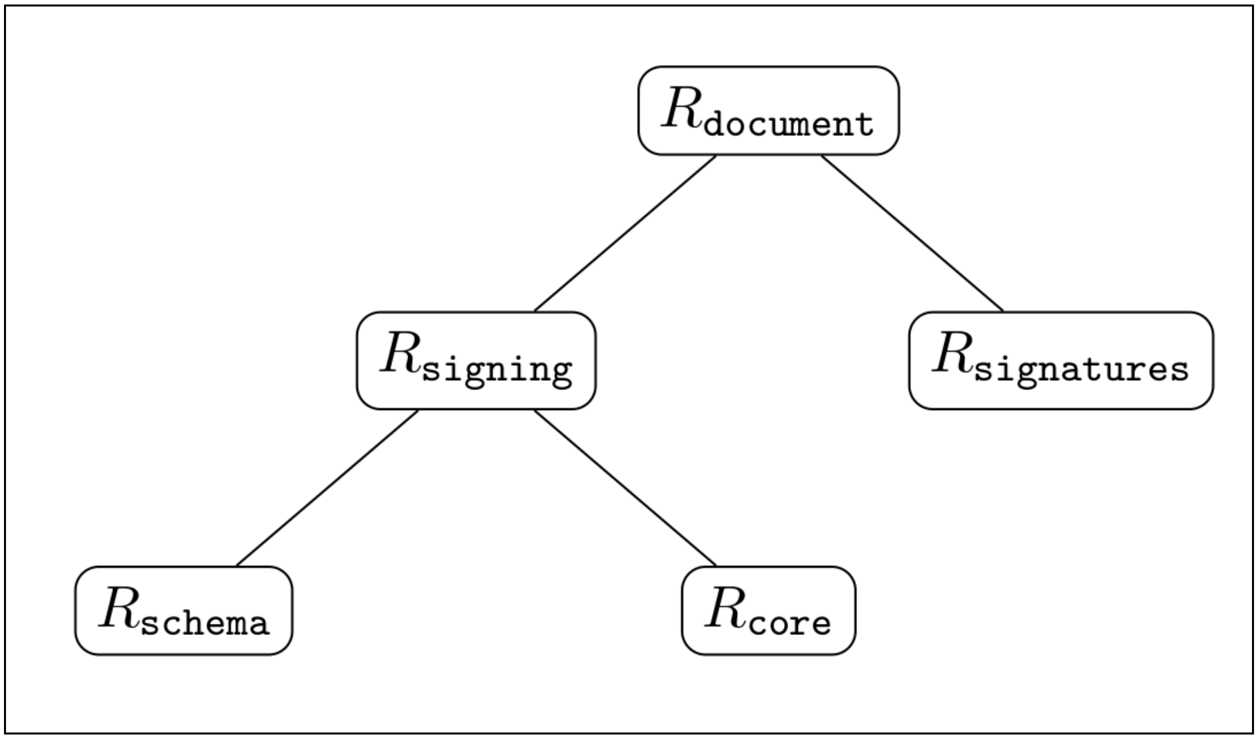

Figure 1: Root hash schema (Figure 3 of the Centrifuge yellow paper)

However, a malicious user can commit a Merke proof containing the `Rsigning` in any leaf, breaking the assumption that the value revealed on `preCommit` call is the `Rsigning` of the tree.

Exploit Scenario
Bob call preCommit to reverse the anchor id with the valid signing root. Eve has access to Bob's identity contract. Eve calls commit with an invalid document root. As a result, Eve blocks Bob's document update.

Recommendation
Ensure the validity of the document root by checking that:

```
      sha256(signingRoot+signatureRoot) == documentRoot
```

is true.

Be aware that checking that an element is inside a Merkle tree does not ensure that the element is at the expected location.

4. Lack of contract existence check may lead to unexpected behavior

Severity: Medium
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-Centrifuge-004
Target: Identity.sol

Description
A failure to check for a contract's existence in the Identity contract may lead to incorrect assumptions in the code execution.

`Identity.execute` calls external contracts using a low-level call:

```
return to.call.value(value)(data);
```

Figure 1: Identity.sol#L78

The Solidity documentation warns:

The low-level call, delegatecall, and callcode will return success if the calling account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.

As a result, `execute` will return true if it points to an address without code, while no code is executed.

Note that the existence must be checked only for non-empty data, to allow the send of ether to non-contract address.

Exploit Scenario
Bob's smart contract calls `execute` with 10 ethers and an incorrect destination. The ethers are lost. Bob's smart contract incorrectly assumes the execution was successful.

Recommendation
For a non-empty data call, check the contract's existence prior to the call, with the assembly opcode extcodesize.

Alternatively, consider adding a separate function, `send_ethers`, which purpose will be to only send ether (with potential data associated), and prevent any call to `execute` for non-existing contract.

Avoid low-level calls. If they are necessary, carefully review the Solidity documentation, in particular, the Warnings section.

5. An anchor can be committed multiple times if its Merkle root is zero

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Centrifuge-005
Target: `AnchorRepository.sol`

Description

`AnchorRepository` was designed to store the anchors. These are meant to be committed only once. If an anchor is committed with a Merkle root equal to zero, it can be committed multiple times.

`AnchorRepository.commit` stores a Merkle root of a given anchor. To ensure that `commit` is called only once per anchor, `_commits[anchorId].docRoot` is checked:

```
//not allowing to write to an existing anchor
require(_commits[anchorId].docRoot == 0x0);
```

Figure 1: AnchorRepository.sol#L85

If an anchor is committed with a Merkle root equal to zero, `commit` can be called a second time, breaking the contract's assumption.

Exploit Scenario

The Centrifuge team develops off-chain code that watches the events associated with the `AnchorRepository` contract. Eve calls `commit` twice on the same anchor destination. This breaks an important invariant of the `commit` function. The code does not handle the double call and crashes.

Recommendation

Prevent anchors with a Merkle tree equal to zero from being committed.

Identify the code properties and test them using Manticore and Echidna.

6. Merkle root verification can be done on empty proofs

Severity: Undetermined
Type: Data validation
Target: MerkleProof.sol

Difficulty: Low
Finding ID: TOB-Centrifuge-006

Description

`MerkleProof.verifySha256` and `verifySha` uses proofs to check that a node is present in a Merkle root. If no proof is provided, both functions will return `true` if the node is the root. While this behavior is correct, it might lead to unexpected behavior.

As shown in Figure 1, the `verifySha256` function uses the proofs to check the presence of a node in a Merkle tree.

```
function verifySha256(
  bytes32[] memory _proof,
  bytes32 _root,
  bytes32 _leaf
)
  internal
  pure
  returns (bool)
{
  bytes32 computedHash = _leaf;


  for (uint256 i = 0; i < _proof.length; i++) {
    bytes32 proofElement = _proof[i];


    if (computedHash < proofElement) {
      // Hash(current computed hash + current element of the proof)
      computedHash = sha256(abi.encodePacked(computedHash, proofElement));
    } else {
      // Hash(current element of the proof + current computed hash)
      computedHash = sha256(abi.encodePacked(proofElement, computedHash));
    }
```

```
   }


   // Check if the computed hash (root) is equal to the provided root
   return computedHash == _root;
 }
```
Figure 1: MerkleProof.sol#L17-L42

If an empty list is provided as a proof, the leaf is compared to the root. This corner case might allow an attacker to provide incorrect information if both the tree and the leaf are controlled.

For example, `Anchor.commit` will check that the signing root is contained in the document root. Here both values are being controlled by the user:

```
     require(_preCommits[anchorId].identity == msg.sender,"Precommit owned
by someone else");
     require(
       MerkleProof.verifySha256(
         documentProofs,
         documentRoot,
         _preCommits[anchorId].signingRoot
       ),
       "Signing root validation failed"
     );
```
Figure 2: AnchorRepository.sol#L93-L101

As a result, the user can provide any value for the document root (and the signing root), if he provides an empty proof.

Exploit Scenario

Eve calls `precommit` with a signing root equal to 1. Eve calls `commit` with a document root equal to 1. As a result, `verifySha256` returns `true` and the document root is 1.

Recommendation

Short term, consider checking for a minimal length of proofs for all calls to `verifySha256` and `verify`.
Long term, add unit-tests on all functions dependent on the Merkle tree verification that highlight failing scenarios, including empty and incorrect proofs.

7. REST API is exposed on all interfaces

Severity: High
Difficulty: Low
Type: Data Exposure
Finding ID: TOB-Centrifuge-007
Target: `Access Controls`

Description
It is possible to make REST API calls to a Centrifuge node from an external machine. This is concerning. Sensitive information, such as the node private key, is exposed through the REST API.

For example, sending an `HTTP GET` request for `/accounts` in the port 8082 produces a response like that in Figure 1.

```
{
    "data": [
        {
            "eth_account": {
                "key": "<JSON-PRIVATE-KEY>",
                "password": "<PASSWORD>"
            },
            ...
        },
        ...
    ]
}
```

Figure 1: Example Centrifuge REST API call response

Note that the response includes both the Ethereum private key used to sign Centrifuge transactions and the password used to secure that private key. This is sufficient information for an attacker to syphon off all of the Ether associated with that key.

Exploit Scenario
An attacker finds that Centrifuge is running on some machine and makes an `HTTP GET` request for `/accounts` using the 8082 port, which produces a response like the one in Figure 1. The attacker obtains the details of the Ethereum private key associated with that node, and steals all of its ethers.

Recommendation
Short term, have the Centrifuge node listen for REST API calls on just the loopback interface (i.e., 127.0.0.1).
Long term, consider a method of communicating with the Centrifuge node that does not involve TCP/IP sockets. For example, Unix local domain sockets are, in general, much easier to control access to.

8. Centrifuge “createconfig” requires password to be passed on command line

Severity: High
Difficulty: High
Type: Data Exposure
Finding ID: TOB-Centrifuge-008
Target: `Centrifuge “createconfig”`

Description
The command line `createconfig` uses a plaintext password. As a result, an attacker can steal the password by watching the list of processes.

If the Ethereum private key used to sign Centrifuge transactions is secured by a password, then that password must be passed to `createconfig` on the command line. From the instructions for installing a Centrifuge node:

> If you have entered a password when creating the geth node in the previous step, you will need to enter this password at this step as well:
>
> ```
> $ centrifuge createconfig \
> -z  ~/.ethereum/keystore/<KEY-FILE> \
> -e ws://127.0.0.1:8546 \
> -t <DEFINE_CONFIG_DIR_NAME> \
> -a 8082 -p 38204 -k <PASSWORD>
> ```

Figure 1: Centrifuge node installation instructions

This makes `<PASSWORD>` available in a process listing (e.g., via “`ps -ef`”), viewable by all users logged into the machine on which `createconfig` is run. Moreover, in our experiments, `createconfig` took approximately 20 seconds to run. Thus, there would be ample opportunity for a malicious user to recover such a password by repeatedly generating process listings.

Exploit Scenario
An attacker has advanced knowledge that Centrifuge is going to be installed on some machine. The attacker gains local access to that machine and runs a script to repeatedly generate process listings. Centrifuge is installed. The attacker recovers the password for the Ethereum key used to sign the node’s transactions.

Recommendation
Short term, if a password is required, then prompt the user to enter it rather than requiring the user to pass it on the command line.

Long term, consider requiring the user to unlock their Ethereum accounts using geth's `--unlock` feature. Doing so could relieve `createconfig` from having to deal with Ethereum private keys at all.

9. Centrifuge “createconfig” stores a plaintext password in a configuration file

Severity: High
Difficulty: High
Type: Data Exposure
Finding ID: TOB-Centrifuge-009
Target: Centrifuge configuration file

Description
The centrifuge node stores paintext the passwords. This practice is highly risky and will easily lead to compromise.

After obtaining the password used to secure an Ethereum private key, `createconfig` stores that password in plaintext in a `config.yaml` configuration file. The password is later read-in by `centrifuge` (the command) and used to unlock the private key.

Note that `config.yaml` is world-readable by default. Thus, any user with read access to a file’s enclosing directory can recover the password. Also note that world read(-only) directory access is a common default on many Unix-based systems.

Exploit Scenario
An attacker learns that Centrifuge is installed on some machine. The attacker gains local access that machine. With the default directory permissions still in place, the attacker reads the plaintext password out of the `config.yaml` file.

Recommendation
Short term, if a password is required, then prompt the user for it each time that `centrifuge` is run, rather than store the password in a configuration file.

Long term, as mentioned in TOB-Centrifuge-008, consider requiring the user to unlock their Ethereum accounts using geth’s `--unlock` feature. Doing so could relieve `createconfig` from having to deal with Ethereum private keys at all.

10. Private keys are world-readable by default

Severity: High
Difficulty: Medium
Type: Data Exposure
Finding ID: TOB-Centrifuge-010
Target: Centrifuge private keys

Description
The centrifuge node keys are world-readable by default, which increases the risk of compromise.

Centrifuge's `createconfig` utility creates two public-private key pairs: one pair to sign Centrifuge documents (`signing.pub.pem` and `signing.key.pem`) and one pair to secure P2P connections (`p2p.pub.pem` and `p2p.key.pem`). All four keys are world-readable by default, including the two private keys (`signing.key.pem` and `p2p.key.pem`). Thus, any user with read access to a files' enclosing directories can recover the private keys. Also note that world read(-only) directory access is a common default on many Unix-based systems.

It is also worth mentioning that the `loadKeyPair` and `loadCertPool` functions will load hard-coded public-private key pairs from the `insecureKey` and `insecureCert` constants defined in the API package as shown in Figure 1.

```
const (
      insecureKey = `-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAyEnDbL/RxZrgDN85W958GvCnWYfLIl/yf3OnzRpSlhz5oKg6
…
-----END RSA PRIVATE KEY-----`
      insecureCert = `-----BEGIN CERTIFICATE-----
MIIDmDCCAoACCQDHr6ZuK9By7zANBgkqhkiG9w0BAQsFADCBjTELMAkGA1UEBhMC
…
-----END CERTIFICATE-----`
)
```

Figure 1: api/insecure.go#L8-L58

Exploit Scenario
Eve learns that Centrifuge is installed on Bob's machine. Eve gains unprivileged local access to Bob's machine. With the default directory permissions still in place, she recovers the private keys. Eve uses the key to sign documents as though she were Bob.

Recommendation
Short term, ensure that when private keys are created, they are readable only by the user who created them (e.g., Unix file permissions `600`).

Long term, refactor Centrifuge node code to safely load and store private keys.

11. Smart contract build dependencies are not up to date

Severity: Informational
Difficulty: Undetermined
Type: Patching
Finding ID: TOB-Centrifuge-011
Target: Centrifuge Ethereum Contracts repository

Description
Updated node modules are available for many of the Centrifuge Ethereum Contracts' build dependencies.

Dependency	Version currently in use	Latest version available
`ethereumjs-util`	5.2.0	6.1.0
`ethereumjs-wallet`	0.6.2	0.6.3
`ganache-cli`	6.1.8	6.4.1
`husky`	1.0.0-rc.15	1.3.1
`solium`	1.2.2	1.2.3
`truffle`	5.0.6	5.0.9
`truffle-hdwallet-provider`	1.0.2	1.0.5
`zos`	2.1.2	2.2.2
`zos-lib`	2.1.2	2.2.2

In particular, `npm audit` indicates that the `zos` 2.1.2 dependency contains a "critical" vulnerability related to a "Sandbox Breakout."

Exploit Scenario
An attacker learns of a exploitable vulnerability in an old version of a build dependency. The attacker forks the Centrifuge Ethereum Contracts repository and modifies the code in a way that looks benign, but actually exploits the machines of the developers who download and build the fork.

Recommendation
Short term, update build process dependencies to the latest version wherever possible.
Long term, integrate the use of `npm audit` into the CI testing to avoid the use of vulnerable dependencies.

References
- Sandbox Breakout
- npm-check

12. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `truffle.js`

Difficulty: Low
Finding ID: TOB-Centrifuge-012

Description
The compilation of the Centrifuge smart contracts has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until just a few months ago. The fix for this bug was not reported in the Solidity CHANGELOG.

A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Centrifuge contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess its maturity.

13. User can commit anchor without requiring collaborators' signatures

Severity: Undetermined Difficulty: High
Type: Data Validation Finding ID: TOB-Centrifuge-013
Target: AnchorRepository.sol

Description
To preserve the collaborators, no signatures are revealed when committing an anchor. As a result, it is possible to anchor a document root without collaborators' consent.

This design choice is required until more private signature verification solutions are used, but forces users to entirely trust their collaborators entirely.

The severity of the issue depends on the interpretation of the purpose of the signatures, which was unclear from the yellow paper.

Exploit Scenario
Bob updates a document with `amountToPay == 0`. Bob does not have the agreement of the other collaborators. Bob anchors the Merkle root of the document. As a result, Bob updates the onchain version of the document without any collaborators consent. The other participants are forced to re-create the document excluding Bob as collaborator.

Recommendation
Short term, clearly document the trust requirement to the users.

Long term, research and implement approaches to:

- Fork documents if there is a disagreement between the collaborators,
- Verify the presence of the signature without revealing them

14. Lack of location verification in the Merkle Tree is error prone

Severity: Undetermined
Difficulty: High
Type: Data Validation
Finding ID: TOB-Centrifuge-014
Target: MerkleProof.sol

Description
Merkle proofs check if an element is present in the Merkle tree. The verification does not check that the location of the element is correct. As a result, an element can be misplaced, leading to unexpected behavior.

In Centrifuge, all the users are supposed to validate that the document root is well-formed. If this assumption is not true, a malicious document tree could trigger unexpected behaviors by duplicating elements, or placing them in unexpected locations. Potential risks include double NFT minting.

The dynamic shape of the tree makes location verification difficult. Due to time constraints, we could not find a viable mitigation that would not require extensive modification.

Exploit Scenario
Bob anchors a document that contains all the elements allowing an NFT token to be minted two times, with different token ids. Bob mints two NFT tokens.

Recommendation
Short term, clearly document that the location in the tree is not checked. Developers creating document checks must be aware of it and take the necessary precautions when writing the checks.

Long term, investigate solutions to confirm the leaf location, or an alternative verification format.

15. ABIEncoderV2 is not production-ready

Severity: Undetermined
Difficulty: Low
Type: Patching
Finding ID: TOB-Centrifuge-015
Target: PaymentObligation.sol, UserMintableERC721.sol

Description
The contracts use the new Solidity ABI encoder: ABIEncoderV2. The encoder is still experimental and not ready for production code.

For example, on March 26th, a severe bug was found in the encoder and was introduced in Solidity 0.5.5.

Due to its experimental status, we expect other bugs to be present in the encoder.

Exploit Scenario
Centrifuge deploys its contracts. After the deployment a bug is found in the encoder. As a result, the contracts are broken and can be exploited to steal the token's ownership.

Recommendation
If you plan to deploy contracts in production in the short term, remove the usage of the ABI encoder V2.

Long term, monitor the development and adoption of Solidity encoder to assess its maturity.

16. Lack of fixed-size data cast on binary.Write call leads to incorrect leaves

Severity: Medium
Type: Data Validation
Target: precise-proofs/flatten.go

Difficulty: Medium
Finding ID: TOB-Centrifuge-016

Description

`flatten.toBytesArray` is called during the leaves creation. `toBytesArray` uses `binary.Write` which requires fixed-size data, but is called with non-fixed size data. As a result, incorrect leaves can be created.

`toBytesArray` calls `binary.Write`:

```go
func toBytesArray(data interface{}) []byte {
	buf := new(bytes.Buffer)
	binary.Write(buf, binary.BigEndian, data)
	return buf.Bytes()
}
```

Figure 1: toBytesArray in proofs/flatten.go#L355-L359

The `binary.Write` documentation states:

Data must be a fixed-size value or a slice of fixed-size values, or a pointer to such data

`toBytesArray` is called several times with a reflect.Len() object, to store the length of the mapping or slice in the tree:

```go
f.appendLeaf(lengthProp, toBytesArray(value.Len()), getSalt(lengthProp.CompactName()),
saltsLengthSuffix, []byte{}, false)

...

f.appendLeaf(lengthProp, toBytesArray(value.Len()), getSalt(lengthProp.CompactName()),
saltsLengthSuffix, []byte{}, false)
```

Figure 2: proofs/flatten.go#L137-L150

`reflect.Len()` returns an `int`. This type is not a fixed-size value and is machine dependent. As a result, the length stored will be zero.

Figure 3 shows an example where toBytesArray returns an incorrect value.

```go
package main

import (
    "fmt"
    "encoding/binary"
    "bytes"
    "reflect"
)

func toBytesArray(data interface{}) []byte {
       buf := new(bytes.Buffer)
       binary.Write(buf, binary.BigEndian, data)
       return buf.Bytes()
}

func main() {

    fVal := reflect.ValueOf("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA")

    buf_1 := toBytesArray(int32(fVal.Len()))
    buf_2 := toBytesArray(fVal.Len())

    fmt.Printf("---> %v\n", buf_1) // returns ---> [0 0 0 36]
    fmt.Printf("---> %v\n", buf_2) // returns ---> []
}
```
Figure 3: Proof of concept of toBytesArray misuse

Exploit Scenario
Bob's document contains a mapping. The length of the mapping is incorrectly stored as a zero value. As a result, Bob's document proofs are incorrect.

Recommendation
Cast the call to `Len()` to uint64.

Check the return error of all the calls to `binary.Write`, including
- flatten.go#L357
- property.go#L40
- property.go#L102

Add gosec to the CI.

17. Manually adding leaves can lead to name collisions in the Merkle tree

Severity: Medium	Difficulty: Medium
Type: Data Validation	Finding ID: TOB-Centrifuge-017
Target: tree.go	

Description
Each leaf of the Merkle requires a unique name to allow the correct generation and verification of the Merkle proofs. This assumption can be broken if the leaves are added manually to the tree.

The precise-proof library allows the addition of leaves to a Merkle tree using the following functions:

- AddLeaves (tree.go#L287-L293)
- AddLeaf (tree.go#L299-L305)
- AddLeavesFromDocument (tree.go#L308-L323)

Each leaf is associated with a name, which is used to identify the proof. If a user adds a leaf with a name already in use, the associated proof will be incorrect.

Figure 1 shows an example of misuse.

```
package main

import (
	"crypto/sha256"
	"fmt"

	documentspb "github.com/centrifuge/precise-proofs/examples/documents"
	"github.com/centrifuge/precise-proofs/proofs"
)

func main() {

	document := documentspb.ExampleDocument{}

	doctree := proofs.NewDocumentTree(proofs.TreeOptions{Hash: sha256.New() /*, Salts:
&salts*/})

	checkErr(doctree.AddLeavesFromDocument(&document))
	checkErr(doctree.AddLeavesFromDocument(&document))
	checkErr(doctree.Generate())

	for _, leaf := range doctree.GetLeaves() {
		fmt.Println("#############")
		fmt.Println(leaf.Property.ReadableName())
		fmt.Println(leaf.Property.CompactName())
	}

}

func checkErr(err error) {
        if err != nil {
                panic(err)
        }
}
```

Figure 1 : example program to reproduce this issue.
Note: GetLeaves() was added to return the leaves of a tree

Exploit Scenario
Bob creates a Merkle tree where two leaves share the same name. As a result, the proof of the first leaf can be used to validate the presence of the second leaf.

Recommendation
Prevent a leaf from being added if its name is already present in the tree.

Improve the unit tests coverage of the API usage of the precise proof library.

18. Nil pointer dereferencing can lead the precise proof library to panic when flattened by protobuf

Severity: Low
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-Centrifuge-018
Target: proofs/flatten.go

Description

A lack of nil pointer check can lead the precise proof library to panic when flattened by protobuf.

`FlattenMessage` calls `handleValue` with `reflect.ValueOf(message)` as a parameter. `reflect.ValueOf` can return a nil pointer.

```
func FlattenMessage(message proto.Message, getSalt GetSalt, saltsLengthSuffix
string, hashFn hash.Hash, compact bool, parentProp Property) (leaves []LeafNode,
err error) {
	f := messageFlattener{
		saltsLengthSuffix: saltsLengthSuffix,
		hash:              hashFn,
		compactProperties: compact,
	}


	err = f.handleValue(parentProp, reflect.ValueOf(message), getSalt,
saltsLengthSuffix, nil)
	if err != nil {
		return
	}


	err = f.sortLeaves()
	if err != nil {
		return []LeafNode{}, err
	}
	return f.leaves, nil
}
```

Figure 1: proofs/flatten.go #L249-L266

The parameter is not checked for `nil` value, but is dereferenced by `handleValue`.

```
func (f *messageFlattener) handleValue(prop Property, value reflect.Value,
getSalt GetSalt, saltsLengthSuffix string, outerFieldDescriptor
*go_descriptor.FieldDescriptorProto) (err error) {
        // handle special cases
        switch v := value.Interface().(type) {
        …
}
```

Figure 2: Header of `handleValue` function in proofs/flatten.go#L32-L34

As a result, a nil pointer dereferencing can occur, leading the library to panic.

Exploit Scenario
Alice submits a "`]0000`" as a document protobuf to Bob. The library triggers the nil pointer dereferencing. This causes Bob's node to crash.

Recommendation
Short term, correctly handle the return of 0 in all the uses of `reflect.ValueOf`.

Long term, periodically run the fuzzer provided in Appendix C to identify code panics.

19. The lack of documentation on the protobuf format invites for incorrect document format

Severity: Informational	Difficulty: Low
Type: Auditing and Logging	Finding ID: TOB-Centrifuge-019
Target:	

Description
The lack of documentation on the expected protobuf format for the document is likely to invite errors and might lead to incorrect document formats.

For example, it is unclear what naming convention should be followed when creating a protobuf document schema. The usage of protobuf can lead to name collision if some prefixes related with internal protobuf fields are used (flatten.go#L66-L75) (e.g.if the name starts with XXX_).

Exploit Scenario
Bob creates a protobuf document format. Bob's format has a field named : `XXX___field`. As a result, the field is not included in the tree, and the Merkle root is incorrect. .

Recommendation
Document how to build a correct protobuf document format.

Consider creating a tool to validate the correctness of a protobuf document format.

20. Error messages can be used to leak the collaborators list

Severity: High
Type: Data Validation
Target: p2p/receiver/handler.go

Difficulty: Low
Finding ID: TOB-Centrifuge-020

Description
The collaborators list of a document is meant to be private. An attacker can identify a document's collaborators by checking the node message errors.

To access a document, among others, the node will check for the local presence of the document, and then the caller privilege access. The error message reported for these two checks will not be the same.

```
func (srv *Handler) GetDocument(ctx context.Context, docReq *p2ppb.GetDocumentRequest,
requester identity.DID) (*p2ppb.GetDocumentResponse, error) {
	model, err := srv.docSrv.GetCurrentVersion(ctx, docReq.DocumentIdentifier)
	if err != nil {
		return nil, err
	}


	if srv.validateDocumentAccess(ctx, docReq, model, requester) != nil {
		return nil, err
	}
```

Figure 1: p2p/receiver/handler.go#L215-L222

Only collaborators of the document will have the local copy and will check for the caller privilege access.
As a result, an attacker can determine if a node is a collaborator of a given document by checking the error message when asking the document to the node.

Exploit Scenario
Eve is not a collaborator of a given document, but wants to determine if Bob is. Eve asks the document to Bob's node. The node returns an error message warning that Eve has no access to the document. As a result, Eve knows that Bob is a collaborator.

Recommendation
Short term, return the same error message in case of privilege access error, and missing documents.

Long term, consider that information can be leaked through side channel.

21. Timing attack can be used to leak the collaborators list

Severity: High
Difficulty: Undetermined
Type: Timing
Finding ID: TOB-Centrifuge-021
Target: handler.go

Description

A document's list of collaborators is meant to be private. An attacker can identify the collaborators of a document through a timing attack.

To access a document, among others, the node will check for the local presence of the document, and then the caller privilege access:

```
func (srv *Handler) GetDocument(ctx context.Context, docReq *p2ppb.GetDocumentRequest,
requester identity.DID) (*p2ppb.GetDocumentResponse, error) {
	model, err := srv.docSrv.GetCurrentVersion(ctx, docReq.DocumentIdentifier)
	if err != nil {
		return nil, err
	}


	if srv.validateDocumentAccess(ctx, docReq, model, requester) != nil {
		return nil, err
	}
```

Figure 1: handler.go#L215-L222

Only collaborators of the document will have the local copy and will check for the caller privilege access.

If the document is not present, the node will reply faster than if the node needs to load the document and check the access. As a result, an attacker can determine if a node is a collaborator of a given document by checking the response time when asking for the document.

This issue is similar to TOB-Centrifuge-019. Due to time constraints, we did not evaluate the difficulty to exploit such a scenario in a real-world setup.

Exploit Scenario

Eve wants to determine if Bob has access to a given document. Eve ask the document to Bob's node 100 times. Based on the average response time, Eve deduces that Bob is a collaborator of the document.

Recommendation
Short term, consider adding a random delay when returning an error message.

Long term, consider that information can be leaked through side channel.

22. Centrifuge nodes sign documents without users' consent

Severity: Undetermined Difficulty: Low
Type: Access Controls Finding ID: TOB-Centrifuge-022
Target: `Handler.HandleRequestDocumentSignature`

Description
The lack of manual verification to sign a document can lead a node to sign incorrect or malicious documents.

When a centrifuge node receives a request to sign a document, the node automatically signs the document with no user interaction. Thus, a node could be made to sign a document that the node's owner would never willingly sign.

The severity of the issue depends on the interpretation of the purpose of the signatures, which was unclear from the yellow paper.

Exploit Scenario
Eve creates a document declaring that Alice owes Eve $100 USD. Eve lists Alice as a collaborator on this document. Eve sends the document to Alice, whose node signs it automatically. Eve seeks payment of the "debt" using the signed document as evidence thereof.

Recommendation
Short term, disable automatic signing of documents. Require explicit approval from users using the REST API.

Long term, implement a user interface that allows a user to accept or reject individual document-signing requests. A minimal solution would simply allow the user to indicate "yes" or "no" to each request. A more complicated solution should focus on identifying unsolicited requests, so that the user can discard them.

23. Messages from an identity with no associated P2P keys leads to a node crash

Severity: Medium
Type: Denial of Service
Target: Identity service

Difficulty: Low
Finding ID: TOB-Centrifuge-023

Description

The identity service's `CurrentP2PKey` function does not check whether an identity has any P2P keys. Thus, when trying to fetch an identity's most recent P2P key, an "index out of range" error can result, causing the node to crash.

The code in question is in Figure 1. If no P2P keys were ever registered with the identity corresponding to `did`, then the array `keys` will be empty, and the variable `lastKey` will be assigned `keys[-1]`. This results in an "index out of range" error causing the node to panic. The panic will not be caught, so the node will crash.

```
// CurrentP2PKey returns the latest P2P key
func (i service) CurrentP2PKey(did id.DID) (ret string, err error) {
	keys, err := i.GetKeysByPurpose(did, &(id.KeyPurposeP2PDiscovery.Value))
	if err != nil {
		return ret, err
	}

	lastKey := keys[len(keys)-1]
```

Figure 1: identity/ideth/service.go#L288-L295

Exploit Scenario

Eve creates an identity but does not register any P2P keys with it. Eve sends a message to Alice on the P2P network. Alice extracts Eve's identity from the message and tries to fetch her P2P keys using `CurrentP2PKey`. Alice's node crashes and must be manually restarted.

Recommendation

Short term, `CurrentP2PKey` should check that the identity corresponding to `did` has at least one P2P key. If the identity has none, then `CurrentP2PKey` should return an error.

Long term, implement a fuzzer to generate random requests from other nodes to make sure that untrusted inputs are properly handled.

24. Updated timestamps can decrease

Severity: Low
Type: Data Validation
Target: go-centrifuge/identity/ideth/service.go

Difficulty: High
Finding ID: TOB-Centrifuge-024

Description
Documents have a timestamp to determine when it was updated last. Due to a lack of timestamp increase validation, it is possible to decrease the document's timestamp during an update.

The timestamp field from a document is used to validate the signatures of all the collaborators, as shown in Figure 1.

```
// ValidateKey checks if a given key is valid for the given centrifugeID.
func (i service) ValidateKey(ctx context.Context, did id.DID, key []byte, purpose *big.Int,
validateAt *time.Time) error {
	....
	// if revoked
	if ethKey.RevokedAt > 0 {
		// if a specific time for validation is provided then we validate if a
revoked key was revoked before the provided time
		if validateAt != nil {
			revokedAtBlock, err := i.client.GetEthClient().BlockByNumber(ctx,
big.NewInt(int64(ethKey.RevokedAt)))
			if err != nil {
				return err
			}


			if big.NewInt(validateAt.Unix()).Cmp(revokedAtBlock.Time()) > 0 {
				return errors.New("the given key [%x] for purpose [%s] has been
revoked before provided time %s", key, purpose.String(), validateAt.String())
			}
		} else {
			return errors.New("the given key [%x] for purpose [%s] has been
revoked and not valid anymore", key, purpose.String())
		}
	}

	…
}
```

Figure 1: *go-centrifuge/identity/ideth/service.go#L328-L369*

There is no validation to ensure that the timestamp of the new document is greater than the current version. As a result, it is possible to update a document with a timestamp lower than the version to be updated.

Exploit Scenario

Bob creates a document with a timestamp of June, 10th 2019. Eve updates the document with a timestamp of May, 10th 2019. As a result, the document associated timestamp is incorrect.

Recommendation

Short term, properly validate the updated timestamp to be greater than the previous one.

Investigate what core document fields should be automatically checked.

25. Incorrect message error handling on invalid document access

Severity: Low
Difficulty: High
Type: Error Reporting
Finding ID: TOB-Centrifuge-025
Target: p2p/receiver/handler.go

Description
A lack of error check leads the Centrifuge node to report an incorrect message error in response to invalid document access.

On a document access request, `GetDocument` executes `srv.validateDocumentAccess`:

```
if srv.validateDocumentAccess(ctx, docReq, model, requester) != nil {
        return nil, err
}
```

Figure 1: p2p/receiver/handler.go#L220-L222

If the function fails, the error returns the `nil` value instead of the error returned by `srv.validateDocumentAccess`. As a result, `GetDocument` returns `(nil, nil)`, and will trigger another error in `PrepareP2PEnvelope`.

The error reported to the user will be related to `PrepareP2PEnvelope` instead of the invalid document access, preventing the user from understanding the failure of the call.

Exploit Scenario
Bob has no access to Alice's document. Bob calls Alice's node to get the document. The node returns an error that does not indicate to Bob its lacks of privilege. Bob is confused and loses a few hours before realizing the problem.

Recommendation
Short term, return the validateDocumentAccess error if the function call fails.

Long term, add unit tests to cover all the possible message errors.

26. libp2p dependencies are not up to date

Severity: Informational
Difficulty: Low
Type: Patching
Finding ID: TOB-Centrifuge-026
Target: `go-centrifuge/Gopkg.toml`

Description
go-centrifuge relies on several outdated dependencies.

For example, go-libp2p uses gx/6.0.1 from June 9th 2018, while the current version is 0.0.3 from March 26th, 2019 (note: they changed the release version format). There were more than 20 releases between gx/6.0.1 and the latest version. The situation is similar for several other packages, including go-libp2p-host and go-libp2p-peer.

As a result, Centrifuge does not benefit from potential security fixes of its dependencies.

Exploit Scenario
An attacker learns of an exploitable vulnerability in an old version of a libp2p golang dependency and uses it to gain unauthorized access to a node or to produce a denial of service.

Recommendation
Short term, update build process dependencies to the latest versions wherever possible.

Long term, implement a check in your CI testing in order to detect out-of-date dependencies.

27. Collaboration possibly leaked at the P2P level

Severity: Undetermined
Type: Data Exposure
Target: P2P network

Difficulty: High
Finding ID: TOB-Centrifuge-027

Description
The libp2p's routing algorithm, which is based on Kademlia, could allow peers to learn when two identities on the network are collaborating.

In a Kademlia network, each node has an associated ID. Suppose Alice needs to communicate with Bob on the network. Of the peers that Alice knows about, Alice determines those whose IDs share a long prefix with Bob's ID. Alice then asks those nodes, "Do you know how to communicate with Bob?" Each node either responds with "Yes" and the relevant details, or "No, but here are some nodes that share a longer ID prefix with Bob than I do." Alice continues in this way until Bob is found.

Thus, if Alice asks Eve "Do you know how to communicate with Bob?", then Eve learns that Alice and Bob are potentially collaborating. Eve could use this information to her advantage.

Exploit Scenario
Alice and Bob collaborate on a document, a fact they wish to keep secret. Alice uses the P2P network to send the document to Bob for signature. In determining how to communicate with Bob, Alice asks Eve, "Do you know how to communicate with Bob?" Eve thereby learns that Alice and Bob are likely collaborating, and tries to use this information for financial gain (e.g., blackmail).

Recommendation
Prevent the leak of information. A solution may be to ask to communicate with other users randomly to introduce noise in the network.

Investigate alternative private routing solutions.

28. Documentation should indicate who the consumers of fields are

Severity: Informational
Difficulty: Undetermined
Type: Data Validation
Finding ID: TOB-Centrifuge-028
Target: Documentation

Description
The existing Centrifuge protocol documentation does not indicate the bodies of code that use and maintain the fields in a Centrifuge document, making its review difficult.

A Centrifuge document consists of many fields. The documentation should make clear which bodies of code make use of those fields. The following are some examples.

- Signatures
    - Centrifuge nodes set this field during document signing.
    - The anchor registry expects this field to exist but does not inspect its contents.
    - The PaymentObligation contract inspects just the owner's signature within this field during token minting.
- TokenId
    - Centrifuge nodes set this field during token minting.
    - The PaymentObligation contract checks this field during token minting, e.g., that no other tokens with this ID exist.
- Timestamp
    - Centrifuge nodes set this field and check it during signature verification.
    - No on-chain code uses this field.

Note that the protocol documentation does mention "core document fields". However, this is not sufficient. For example, neither TokenId nor Timestamp is included in "core document fields."

The problem is one of compartmentalization. Many parties interact with Centrifuge documents, e.g., the node that authors a document, the node's collaborators, "core" Centrifuge contracts (e.g., `AnchorRepository.sol`), other contracts, etc. At present, there is no clear delineation as to which parties should be concerned with which fields. As such, when a change is made to *any* field, one must consider the effects of that change on *all* possible parties.

Even if compartmentmentalization cannot be enforced using the source language (e.g., Solidity, Go), it can still be documented. This would involve (1) introducing additional categories like "core document fields" mentioned above, and (2) laying out precisely, for each category, the parties that can read from the fields in that category, and the parties that can write to the fields in that category.

Exploit Scenario
Eve submits a pull request to the Centrifuge repository affecting how some document field is updated. Reviewers are forced to consider how that change might affect all parties that interact with Centrifuge documents. Time and resources are wasted. Such waste would have been avoided had proper compartmentalization been implemented.

Recommendation
Document the interactions and the expected fields of the Centrifuge components.

29. Consider requiring consent to become a collaborator

Severity: Informational	Difficulty: Undetermined
Type: Access Controls	Finding ID: TOB-Centrifuge-029
Target: P2P network	

Description
Currently, anyone on the P2P network can list anyone else as collaborator. This exposes nodes to unwanted and potentially malicious traffic.

A possible solution would be to implement whitelisting. More specifically, the owner of a node could prepare a list of the other nodes with whom they are willing to collaborate. A node utilizing a whitelist would only accept and parse documents from nodes on the whitelist.

Exploit Scenario
Eve discovers a document parsing vulnerability. Eve creates a document to exploit the vulnerability and sends it to Alice. Alice's node accepts the document even though she has never heard of Eve. Alice experiences financial and/or data loss as a result of accepting the document.

Recommendation
Short term, disable automatic collaborator acceptance. Require explicit approval from users using the REST API.

Long term, implement a user interface that allows a user to accept or reject individual collaboration requests.

30. Anchor id update allows for multiple tokens mint for the same document

Severity: Medium
Type: Data Validation
Target: PaymentObligation.sol

Difficulty: High
Finding ID: TOB-Centrifuge-030

Description

A NFT can be minted to represent a document. A document is supposed to have only one NFT token associated. This assumption can be broken if the token id of the document is updated.

The id of the NFT is the token id of the document:

```
function mint(
   address to,
   uint256 tokenId,
   string memory tokenURI,
   uint256 anchorId,
   bytes[] memory properties,
   bytes[] memory values,
   bytes32[] memory salts,
   bytes32[][] memory proofs
 )
 public
 {
   // First check if the tokenId exists
   require(
     !_exists(tokenId),
     "Token exists"
   );
```

Figure 1: PaymentObligation.sol#L116-L132

`PaymentObligation` ensures that a NFT is not minted multiple times by checking the existence of the token id.

However, the document can change its token id during an update. As a result, users can mint multiple tokens for a document by changing the token id value.

Exploit Scenario

Eve creates a payment obligation and mints two NFTs for it, breaking the invariant that only one NFT can exist for a document. Eve takes advantage of the broken invariant by selling the two NFTs to Alice and Bob. Alice and Bob each expect the invariant to hold, i.e., expect their NFT to be the sole NFT associated with the payment obligation. They do not realize that both NFTs correspond to the same “debt”, and that they cannot both collect on it.
It is worth mentioning that this attack can be prevented if the participants use unmodified Centrifuge nodes, or if the trustworthiness of the participants is verified outside of the Centrifuge protocol.

Recommendation
Short term, implement an onchain mapping from document IDs to token IDs. Before minting a token for document, check that no token has already been minted for that document. Prevent the document ID to be changed during update.

Long term, investigate what core document fields should be automatically checked on-chain and off-chain.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client’s reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General recommendations

- **Use the terminology in the implementation as it appears in the yellow paper.** Using the same terminology simplifies the review of the code and is less error-prone. For example `Rcore` of the yellow paper if called `documentRoot` in the contract.

Yellow paper

- **Split the equations into two categories: equations and properties.** Clearly identifying properties helps in their review. For example, equations (7), (13), (14), (15) and (16) are properties that must remain true**.**
- **Add an Appendix that describes how each property is guaranteed or checked.** Identify how the properties are checked helps to ensure their correct behavior.

Centrifuge node

- **Change `len(collaborators) < 0` to `len(collaborators) == 0 in documents/write_acls.go##L218.`** `len(collaborators) < 0` will always return false, as a result, the branch is never taken.
- **Fix the nil pointer dereference in identity/ideth/factory.go#L49-L54.** If the user has insufficient ether, `SubmitTransactionWithRetries` returns a nil pointer and an error. The nil pointer is dereferenced in the log message.
- **Fix Go vet context leaks reports**. To avoid wasting memory, consider adding "`defer cancel()`" immediately after calls to `WithTimeout` or `WithCancel`.
- **Rename the identity variables (did.go#L31-L33, L40-L44) to avoid too-similar names.** `keyPurposeAction, keyPurposeP2PDiscovery, keyPurposeSigning` are too similar to `KeyPurposeAction, KeyPurposeP2PDiscovery, KeyPurposeSigning`. Variables with names that are too similar are error-prone and more difficult to review.
- **Make sure `RandomSlice` (go-centrifuge/utils/tools.go#L96-L105) always returns a list with random bytes**. If the `size` parameter is equal to 0, this function will return an empty list, which could produce undefined behavior in other parts of the code.

Smart contracts

- **Consider having the Identity contract's `execute` method revert when its call fails.** Reverting makes sure the caller does not forget to check for errors.

Deployment

- **Consolidate uses of address 0x89b0a86583c4444acfd71b463e0d3c55ae1412a5.** Naming this address with a constant and using that constant consistently will make it easier to deploy in new settings (e.g., using Ganache).
- **Fix the deploy procedure of the Centrifuge Ethereum Contracts to work reliably.** Deploying the centrifuge-ethereum-contracts to the Docker container will fail sporadically. This costs development time and raises doubts about the correctness of tests involving those deployed contracts.

C. Fuzzer-based Test Cases for Centrifuge

Trail of Bits has added test cases for go-fuzz and libFuzzer to the go-centrifuge repository.

Test Cases for Centrifuge

Trail of Bits is working to include a collection of fuzzing tests using go-fuzz and libFuzzer, two high-performance, coverage-guided and evolutionary fuzzing engines. These tests cover a variety of parsing and processing functions, as well as functions that handle untrusted inputs. We integrated them into the build process in order to provide improved testing of the go-centrifuge code. For instance, Figure C.1 shows the tests created to ensure the robustness of the `ResolveDataEnvelope` implementation. Any panic will be reported by the fuzzers.

```
func FuzzResolveDataEnvelope(data []byte) int {
    p := &protocolpb.P2PEnvelope{}
    err := proto.Unmarshal(data, p)
    if err != nil {
            return 0
    }
    p2pcommon.ResolveDataEnvelope(p)
    return 1
}
```

FIgure C.1, a test for unexpected behavior in `ResolveDataEnvelope`

Our suite of fuzzer supports both go-fuzz and libfuzzer fuzzing. The two fuzzers may provide different levels of coverage and discover different bugs. Trail of Bits recommends running the fuzzers with both go-fuzz and libFuzzer to maximize coverage.

Our current tests cover the following functionality:

- Parsing, processing and generation of proofs from protobuf inputs.
- Extraction and parsing of field tags from protobufs.
- Parsing, processing and handling of Ethereum addresses.
- Serialization and deserialization of bytes as hex numbers (using `hexutil`)
- Use of the `SliceOfByteSlicesToHexStringSlice` function.
- Serialization and deserialization of `BigInts`.
- Parsing, processing and handling versions strings.
- Extraction, processing and handling of DID
- Use of `ResolveDataEnvelope` function.

These test cases will be built via the `Makefile.fuzzing`. In order to perform the building of the libFuzzer tests, Clang++ 6.0 or newer is required. For instance, to compile the hexutil fuzz tests:

```
$ make -f Makefile.fuzz build TARGET=Hexutil
```

After the build finishes, a fuzzing campaign could be started using one of these commands:

```
$ make -f Makefile.fuzz fuzz-go TARGET=Hexutil
$ make -f Makefile.fuzz fuzz-libfuzzer TARGET=Hexutil
```

The results will be stored in the corresponding `fuzzing/$TARGET` directory.

Measuring coverage

Regardless of how inputs are generated, an important task after running a fuzzing campaign is to measure its coverage. To perform this measure, we used the support provided by Go's source-based code coverage feature (https://golang.org/cmd/cover/). This feature runs only with go-fuzz.

After running this fuzzer for a while, we can generate the coverage report with the following command:

```
$ cd fuzzing/$TARGET
$ sed '/0.0,1.1/d' coverprofile > coverprofile.fixed
$ go tool cover -html=coverprofile.fixed
```

The cover tool will produce an html with the exact lines covered during the fuzzing campaign as well as some statistics on the number of lines.

Integrating fuzzing and coverage measurement into the development cycle

Once the fuzzing procedure has been tuned for speed and efficiency, it should be properly integrated in the development cycle to catch bugs. We recommend adopting the following procedure to integrate fuzzing using a CI system:

1. After the initial fuzzing campaign, save the corpus of every test generated. We provide the initial corpora.
2. For every internal milestone, new feature or public release, re-run the fuzzing campaign for each test for at least 24 hours, starting with the current corpora.[1]
3. Update the corpora with the new inputs generated.

[1] For more on fuzz-driven development, see this CppCon 2017 talk by Kostya Serebryany of Google

Note that over time, the corpora will come to represent thousands of CPU hours of refinement, and be very valuable for guiding efficient code coverage during fuzz-testing. However, an attacker could also use them to quickly identify vulnerable code. We recommend avoiding this additional risk by keeping fuzzing corpora in an access-controlled storage location rather than in a public repository. Some CI systems allow maintainers to keep a cache to accelerate building and testing. The corpora could be included in such a cache, if they are not very large.

D. Fix Log

Trail of Bits performed a retest of the Centrifuge system on July 8th and 9th, 2019. Centrifuge provided fixes and supporting documentation for the findings outlined in the most recent security assessment. Trail of Bits performed verification of each fix provided for the findings detailed in the report, using a best-effort methodology.

Centrifuge introduced the required protections to improve the security of their smart contracts and the off-chain code executed by the nodes. They also fixed issues and potentially problematic corner cases in their precise proof library (findings 16-19) to avoid users from generating incorrect or invalid proofs. A detailed log of their responses to discovered issues follows below.

Fix Log Summary

#	Title	Severity	Status
1	Revoked Management and Action keys can still be used	High	Fixed
2	A race condition on commit can break document updates	Medium	Issue documented
3	User can commit anchor with incorrect Merkle root proof	Undetermined	Fixed
4	Lack of contract existence check may lead to unexpected behavior	Medium	Fixed
5	An anchor can be committed multiple times if its Merkle root is zero	Low	Fixed
6	Merkle root verification can be done on empty proofs	Undetermined	Fixed
7	REST API is exposed on all interfaces	High	Fixed
8	Centrifuge “createconfig” requires password to be passed on command line	High	Fixed
9	Centrifuge “createconfig” stores a plaintext password in a configuration file	High	Fixed

10	Private keys are world-readable by default	High	Fixed
11	Smart contract build dependencies are not up to date	Informational	Fixed
12	Solidity compiler optimizations can be dangerous	Undetermined	Partially fixed
13	User can commit anchor without requiring collaborators' signatures	Undetermined	Issue documented
14	Lack of location verification in the Merkle Tree is error prone	Undetermined	Partially fixed
15	ABIEncoderV2 is not production-ready	Undetermined	Won't fix
16	Lack of fixed-size data cast on binary.Write call leads to incorrect leaves	Medium	Fixed
17	Manually adding leaves can lead to name collisions in the Merkle tree	Medium	Fixed
18	Nil pointer dereferencing can lead the precise proof library to panic when flattened by protobuf	Low	Fixed
19	The lack of documentation on the protobuf format invites for incorrect document format	Informational	Fixed
20	Error messages can be used to leak the collaborators list	High	Fixed
21	Timing attack can be used to leak the collaborators list	High	Fixed
22	Centrifuge nodes sign documents without users' consent	Undetermined	Issue documented
23	Messages from an identity with no associated P2P keys leads to a node crash	Medium	Fixed

24	Updated timestamps can decrease	Low	Fixed
25	Incorrect message error handling on invalid document access	Low	Fixed
26	libp2p dependencies are not up to date	Informational	Fixed
27	Collaboration possibly leaked at the P2P level	Undetermined	In progress
28	Documentation should indicate who the consumers of fields are	Informational	Won't fix
29	Consider requiring consent to become a collaborator	Informational	Won't fix
30	Anchor id update allows for multiple tokens mint for the same document	Medium	Won't fix

Detailed Fix Log

Finding 1: Revoked Management and Action keys can still be used
Resolved by verifying the `revokedAt` field every time a key is used.

Finding 2: A race condition on commit can break document updates
Properly documented as a limitation of the current implementation:

"Two or more collaborators could try to update a document at the same time. The "first" update that goes through (the first version being anchored) essentially blocks the other from updating the desired document version.

Mitigation is to always have "pre-commit" enabled. Mid-term this is also possible to be mitigated by supporting document forking/merging."

Finding 3: User can commit anchor with incorrect Merkle root proof
Resolved by checking that that proof provided follows the structure required in the protocol specification.

Finding 4: Lack of contract existence check may lead to unexpected behavior
Resolved by checking the contract's existence prior to the call, with the assembly opcode extcodesize.

Finding 5: An anchor can be committed multiple times if its Merkle root is zero
Resolved by preventing anchors with a Merkle tree equal to zero from being committed.

Finding 6: Merkle root verification can be done on empty proofs
Resolved by invalidating empty proofs.

Finding 7: REST API is exposed on all interfaces
Resolved by adding a configuration option to specify the interface to expose the REST API that defaults to the localhost (127.0.0.1).

Finding 8: Centrifuge "createconfig" requires password to be passed on command line
Resolved by asking the user to securely type the password to unlock the node wallet.

Finding 9: Centrifuge "createconfig" stores a plaintext password in a configuration file
Resolved by removing the password field from the configuration file.

Finding 10: Private keys are world-readable by default
Resolved by writing the configuration file readable only by the user who created them (Unix file permissions 600).

Finding 11: Smart contract build dependencies are not up to date
Resolved by updating build process dependencies to the latest version wherever possible.

Finding 12: Solidity compiler optimizations can be dangerous
Partially resolved by using less aggressive optimizations and downgrading the compiler to 0.5.3. The Centrifuge team responded:

"We chose to downgrade to version 0.5.3 as the contracts become undeployable without the gas optimizations. Version 0.5.3 of the Solidity compiler is a reasonable compromise of having the new features available and a compiler version that has been in use for enough time to give us confidence of its security."

Finding 13: User can commit anchor without requiring collaborators' signatures
Properly documented as a limitation of the current implementation:

"It is possible for any collaborator to anchor a new document version at any time. Previous collaborator's signatures are not required to anchor/publish a new document version. This is less of a limitation and more of a feature to prevent malicious collaborators from blocking documents by withholding signatures.

Mid-term a feature could be added that requires an x of n signature scheme where a certain threshold of collaborator signatures is required to anchor a new state. For now, anybody can publish a new version of a document."

Finding 14: Lack of location verification in the Merkle Tree is error prone
Partially resolved in some specific cases, by checking that that proof provided follows the structure required in the protocol specification (See Finding 3). The Centrifuge team responded:

"In Centrifuge OS, the trust in the individual collaborators of a document is an important piece of the whole system. Multiple collaborators, who sign off on a document, increase the security of document validity. A single, malicious/buggy, user can construct and sign any document they want and publish the root hash on Ethereum without any further validations happening. As soon as multiple Centrifuge users collaborate on a document, the benign users will withhold their signatures when they detect a wrongly created document. Hence the trust in an NFT being minted lies either in the single entity that created & signed the document and minted the NFT, or in the group of collaborators who signed off on the document. The correct validation of the overall tree with all its leaves will always happen off-chain and rely on the agreement of multiple collaborators signing off. A buyer of an NFT will have to rely either on trusting the single entity

who minted the NFT in the first place, or the group of collaborators who signed off. This is a design decision and a buyer of an NFT will always have to trust (or verify) the underlying data of the private document or the author of the document."

Finding 15: ABIEncoderV2 is not production-ready
Not resolved. The Centrifuge team responded:

"We chose the ABIEncoderV2 with considerations as we need the ability to pass nested lists into contract calls. Even though the feature is marked "experimental" we deem it an appropriate risk to use it. Other prominent projects, like MakerDAO, for example in https://github.com/makerdao/dss/blob/master/src/jug.sol, are using ABIEncoderV2 as well. We deem the risk adequate compared to the benefit provided."

Finding 16: Lack of fixed-size data cast on binary.Write call leads to incorrect leaves
Resolved by properly implementing the `toBytesArray` function using a cast to `int64`.

Finding 17: Manually adding leaves can lead to name collisions in the Merkle tree
Resolved by properly implementing the addToLeave function to check that no leaves can be added if its name is already present in the tree.

Finding 18: Nil pointer dereferencing can lead the precise proof library to panic when flattened by protobuf
Resolved by checking the validity of the value to flatten before using reflection, which triggered the null pointer dereference.

Finding 19: The lack of documentation on the protobuf format invites for incorrect document format
Resolved by adding proper documentation to warn about adding protobuf fields with invalid fields that can produce invalid proofs.

Finding 20: Error messages can be used to leak the collaborators list
Resolved by masking the error messages that outputted when a document is not found.

Finding 21: Timing attack can be used to leak the collaborators list
Resolved by adding a minimal amount of time to respond to any document request.

Finding 22: Centrifuge nodes sign documents without users' consent
Properly documented as a limitation of the current implementation:

"A Centrifuge node is a technical client to Centrifuge OS. This client exchanges and signs data in well-known formats. It does not validate document data authenticity.

Data authenticity and correctness are always validated by the upstream system. E.g. the accounting system interacting with a Centrifuge node.

A signature of a collaborator on a Centrifuge document signifies the technical receipt and validation of a message. It does not signify the agreement that a document itself is valid, e.g. if an invoice amount is matching the underlying purchase order.

It is possible to attach additional signatures to a document (e.g., with custom attributes) to indicate "business agreement" of a document. However, this is not part of the protocol specifications and is the responsibility of an upstream system."

Finding 23: Messages from an identity with no associated P2P keys leads to a node crash
Resolved by properly checking for empty lists of keys, when a particular P2P key is required.

Finding 24: Updated timestamps can decrease
Resolved by validating the timestamp every time a collaborator updates it.

Finding 25: Incorrect message error handling on invalid document access
Resolved by properly implementing the missing error handling routine.

Finding 26: libp2p dependencies are not up to date
Resolved by updating all libp2p node dependencies to latest stable versions.

Finding 27: Collaboration possibly leaked at the P2P level
Not resolved yet, but in progress. The Centrifuge team responded:

"This is a common problem with Kademlia DHT. Longterm it will be addressed by replacing our DHT with one that can be queried anonymously. We expect this to be implemented as a libp2p module. HOPR is one project that could be used for node discovery or expose DHT nodes on TOR."

Finding 28: Documentation should indicate who the consumers of fields are
Not resolved. The Centrifuge team responded:

"Documentation is always good and can always be improved. The definition of Centrifuge messages is separated out into an individual repository (https://github.com/centrifuge/centrifuge-protobufs) and there is, in our opinion, a reasonable separation from Core Document to individual business document structures (Invoice, Purchase Order, etc.). Additionally the Centrifuge Protocol Paper (https://github.com/centrifuge/protocol/releases) outlines the use of each of the key protocol-specific fields."

Finding 29: Consider requiring consent to become a collaborator
Not resolved. The Centrifuge team responded:

"We considered this design decision and decided to allow message sending/receiving by any participant in the network in order to create the most open network possible. There are possibilities to add white listing and rate limiting of messages at a later point, which we will consider."

Finding 30: Anchor id update allows for multiple tokens mint for the same document
Not resolved. The Centrifuge team responded:

"In Centrifuge OS, the trust in the individual collaborators of a document is an important piece of the whole system. Multiple collaborators, who sign off on a document, increase the security of document validity. A single, malicious or buggy user can construct and sign any document they want and publish the root hash on Ethereum without any further validations happening. As soon as multiple Centrifuge users collaborate on a document, the benign users will withhold their signatures when they detect a wrongly created document. Hence the trust in an NFT being minted lies either in the single entity that created & signed the document and minted the NFT, or in the group of collaborators who signed off on the document. A buyer of an NFT will have to rely either on trusting the single entity who created the document and minted the NFT in the first place, or the group of collaborators who signed off. This is a design decision and a buyer of an NFT will always have to trust (or verify) the underlying data of the private document or the author of the document."