Storage Node

0g Labs

/ DRAFT /

HALBORN

Storage Node - 0g Labs

Prepared by: HALBORN

Last Updated 09/12/2024

Date of Engagement by: August 12th, 2024 - September 6th, 2024

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
14	2	1	1	4	6

TABLE OF CONTENTS

1. Introduction

0g Labs engaged Halborn to conduct a security assessment on their storage node beginning on August 12th, 2024 and ending on September 6th, 2024. The security assessment was scoped to the modules provided in the GitHub repository Og-storage-node, commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn assigned one full-time security engineer to check the security of the storage node. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the functionalities of the modules operate as intended
- Identify potential security issues with the modules

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which need to be addressed by the 0g Labs team. The main ones are the following:

- Correct the flawed check that causes every recall position to fall within a mine range.
- Prevent appending null nodes.
- Prevent updating the last leaf node with a null value, or adjust the logic to avoid corrupting the Merkle tree.
- Replace the panic macro calls with appropriate error-handling mechanisms that safely log errors and allow the calling service to continue functioning.
- Verify that the miner ID is correctly stored on-chain, even when the miner ID in the configuration matches the one in the database.

3. Test Approach And Methodology

`Halborn` performed a combination of the manual view of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the storage node assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance the coverage of modules. They can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture, purpose, and use of the storage node.
- Manual code read and walkthrough.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Cross module call controls.
- Architecture related logical controls.
- Fuzz testing (`cargo-fuzz`).
- Scanning of Rust files for vulnerabilities (`cargo audit`).
- Checking the unsafe code usage (`cargo-geiger`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: Og-storage-node

(b) Assessed Commit ID: 53449e1

(c) Items in scope:

- node/miner
- common/zgs_seal
- common/append_merkle

Out-of-Scope: Third party dependencies., Economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	1	1	4	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - RECALL POSITIONS MAY BE FALSELY MARKED AS COVERED WITHIN A MINER'S MINE RANGE	CRITICAL	-
HAL-14 - APPENDING A NULL NODE TEMPORARILY DISABLES PROOF GENERATION AND MAY LEAD TO NODE INSTABILITY	CRITICAL	-
HAL-04 - OVERWRITING THE LAST NON-NULL LEAF WITH NULL CORRUPTS THE MERKLE TREE	HIGH	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-15 - USE OF PANIC MACROS IN ERROR HANDLING	MEDIUM	-
HAL-03 - POSSIBILITY OF UNREGISTERED MINERS PARTICIPATING IN PORA	LOW	-
HAL-06 - DISABLED OVERFLOW CHECKS AND USAGE OF UNCHECKED MATH OPERATIONS	LOW	-
HAL-07 - MISMATCH BETWEEN DOCUMENTATION AND CODE IN PORA HASH CALCULATION	LOW	-
HAL-05 - AN UNKNOWN LEAF NODE CAN BE FILLED WITH NULL	LOW	-
HAL-10 - INACCURATE ERROR MESSAGES	INFORMATIONAL	-
HAL-08 - INEFFECTIVE COMPUTATION	INFORMATIONAL	-
HAL-09 - INEFFICIENT CODE	INFORMATIONAL	-
HAL-11 - PRESENCE OF TODOS IMPLYING UNFINISHED CODE	INFORMATIONAL	-
HAL-12 - PRESENCE OF TYPOS	INFORMATIONAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-13 - MISLEADING DOCUMENTATION AND COMMENTS	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) RECALL POSITIONS MAY BE FALSELY MARKED AS COVERED WITHIN A MINER'S MINE RANGE

// CRITICAL

Description

The `mine::is_covered` function checks whether a given recall position is covered fully or partially within the miner's mine range.
However, the current flawed logic makes it possible that any recall position will be considered covered due to the following implementation:

```
Some(
    self_start_position <= recall_position + SECTORS_PER_LOAD as u64
        || self_end_position > recall_position,
)
```

This can be interpreted in the following way:

- If recall position is smaller than a mine range end position, `self_end_position > recall_position` evaluates to **true**.
- If recall position is bigger or equal to a mine range's end position, which is already bigger than the start position, then the expression `self_start_position <= recall_position + SECTORS_PER_LOAD as u64` evaluates to **true**.

As a result, the function will always return `true`. Consequently, this function check will be consistently bypassed during PoRA iterations. This oversight will lead to the miner attempting to load sealed data from the database at the specified recall position immediately afterward. Consequently, data loading failures will cause a decline in miner performance and potential mining rewards.

Code Location

The `pora::iteration` function verifies if the recall position falls within the miner's mine range and, if so, attempts to load the corresponding sealed data from the database:

```
62  pub async fn iteration(&self, nonce: H256) -> Option<AnswerWithoutProof>
63      inc_counter(&SCRATCH_PAD_ITER_COUNT);
64      let ScratchPad {
65          scratch_pad,
66          recall_seed,
67
```

```
68              pad_seed,
69          } = self.make_scratch_pad(&nonce);
70
71          let recall_position = self.range.load_position(recall_seed)?;
72          if !self.mine_range_config.is_covered(recall_position).unwrap() {
73              trace!(
74                  "recall offset not in range: recall_offset={}",
75                  recall_position,
76              );
77              return None;
78          }
79
80          inc_counter(&LOADING_COUNT);
81          let MineLoadChunk {
82              loaded_chunk,
83              avalibilities,
84          } = self
85              .loader
86              .load_sealed_data(recall_position / SECTORS_PER_LOAD as u64)
87              .await?;
            ...
```

Proof of Concept

PoC Explanation: The PoC confirms that a recall position exceeding the end of a mining range is incorrectly treated as covered. The correct behavior should prevent this.

The following PoC was added to the `node/miner/src/mine.rs` file:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_uncovered_position_considered_covered() {
        let shard_config = ShardConfig::default();
        let start_position = 1024u64;
        let end_position = 2048u64;
        let mine_range = MineRangeConfig { start_position: Some(start_positior
        assert!(mine_range.is_covered(end_position + 1).unwrap()); // recall_p
    }
}
```

Executing the PoC produces the following result:

```
~/Doc/p/r/0g-storage-node @53449e1f !15 ?2 ❯ cargo test -p miner --release test_uncovered_posit
ion_considered_covered -- --nocapture -- --exact
   Compiling contract-interface v0.1.0 (/Users/cyberj0e/Documents/projects/rust/0g-storage-node
/common/contract-interface)
   Compiling miner v0.1.0 (/Users/cyberj0e/Documents/projects/rust/0g-storage-node/node/miner)
    Finished release [optimized] target(s) in 10.21s
     Running unittests src/lib.rs (target/release/deps/miner-938a91429ee983aa)

running 1 test
test mine::tests::test_uncovered_position_considered_covered ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 7 filtered out; finished in 0.00s
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:H/D:N/Y:M (10.0)

Recommendation

It is recommended to use the AND (&&) operator instead of the OR (||) operator to ensure the function aligns with its intended behavior:

```
Some(
    self_start_position <= recall_position + SECTORS_PER_LOAD as u64
        && self_end_position > recall_position,
)
```

7.2 (HAL-14) APPENDING A NULL NODE TEMPORARILY DISABLES PROOF GENERATION AND MAY LEAD TO NODE INSTABILITY

// CRITICAL

Description

The `AppendMerkleTree` tree has multiple functions for appending new nodes:

- `append`: appends a new leaf to the Merkle tree.
- `append_list`: appends a list of new leaves to the Merkle tree.
- `append_subtree`: appends a leaf list by providing their intermediate node hash.
- `append_subtree_list`: appends multiple leaf lists by providing their intermediate node hashes.

However, the appended nodes are not validated to ensure they are non-null. As a result, during tree recomputation, new parents for these null nodes will also be null, leading to a Merkle root that is null. This will have the following impact:

- **Proof generation:** The presence of a null root in the Merkle tree will render proof generation impossible for any of the tree's leaves. This is because the proof's lemma depends on having a non-null Merkle tree root.
- **Node stability:** Processes utilizing the `storage` module, which invokes `AppendMerkleTree::get_subtrees()`, will experience crashes. This instability arises from the panic triggered by the `first_known_root_at` function, which is called by `get_subtrees()`, due to the presence of at least one leaf where the path from the leaf to the Merkle tree root consists solely of null nodes.

The Merkle tree root will remain null until all the inserted null leaves are replaced with their actual values.

Code Location

The `append` function does not verify that the new leaf being appended is non-null:

```
141 pub fn append(&mut self, new_leaf: E) {
142     self.layers[0].push(new_leaf);
143     self.recompute_after_append_leaves(self.leaves() - 1);
144 }
```

The `append_list` function does not verify that the new leaves being appended are non-null:

```
146 pub fn append_list(&mut self, mut leaf_list: Vec<E>) {
147     let start_index = self.leaves();
148     self.layers[0].append(&mut leaf_list);
149
```

```
150     self.recompute_after_append_leaves(start_index);
    }
```

The `append_subtree` function does not verify that the `subtree_root` being appended is non-null:

```
157 pub fn append_subtree(&mut self, subtree_depth: usize, subtree_root: E) -
158     let start_index = self.leaves();
159     self.append_subtree_inner(subtree_depth, subtree_root)?;
160     self.recompute_after_append_subtree(start_index, subtree_depth - 1);
161     Ok(())
162 }
```

The `append_subtree_list` function does not verify that the `subtree_list` elements being appended are non-null:

```
164 pub fn append_subtree_list(&mut self, subtree_list: Vec<(usize, E)>) -> R
165     for (subtree_depth, subtree_root) in subtree_list {
166         let start_index = self.leaves();
167         self.append_subtree_inner(subtree_depth, subtree_root)?;
168         self.recompute_after_append_subtree(start_index, subtree_depth -
169     }
170     Ok(())
171 }
```

The `recompute` function will set a new parent node to null if any of its child nodes are null:

```
403 while let Some([left, right]) = iter.next() {
404     // If either left or right is null (unknown), we cannot compute the p
405     ...
406     let parent = if *left == E::null() || *right == E::null() {
407         E::null()
408     } else {
409         A::parent(left, right)
410     };
411     parent_update.push((next_layer_start_index + i, parent));
412     i += 1;
413 }
```

The `first_known_root_at` function will panic with the `unreachable!` macro if the path from the given leaf to the Merkle tree root contains only null nodes:

```
560 fn first_known_root_at(&self, index: usize) -> (usize, E) {
561     let mut height = 0;
562     let mut index_in_layer = index;
563     while height < self.layers.len() {
564         let node = self.node(height, index_in_layer);
565         if !node.is_null() {
566             return (height + 1, node.clone());
567         }
568         height += 1;
569         index_in_layer /= 2;
570     }
571     unreachable!("root is always available")
572 }
```

The `get_subtrees` function invokes the `first_known_root_at` function for each leaf in the Merkle tree:

```
308 pub fn get_subtrees(&self) -> Vec<(usize, E)> {
309     let mut next_index = 0;
310     let mut subtree_list: Vec<(usize, E)> = Vec::new();
311     while next_index < self.leaves() {
312         let root_tuple = self.first_known_root_at(next_index);
313     ...
```

Proof of Concept

PoC Explanation: The PoC performs the following steps:

1. Construct a balanced Merkle tree consisting of 4 filled leaves
2. Ensure that the current Merkle tree root is not null
3. Append a new leaf that has a null hash
4. Validate that the new Merkle tree root is null
5. Validate that proof generation is not possible for all the leaves
6. Call `get_subtrees()` to trigger a panic with a message that matches the one specified in the `#[should_panic]` attribute

The following test was added to the `common/append_merkle/src/lib.rs` file:

```
#[test]
#[should_panic = "entered unreachable code: root is always available"]
fn test_append_with_null() {
    let nb_non_empty_leaves: usize = 4;
    let mut leaves = Vec::new();
```

```
    for _ in 0..nb_non_empty_leaves {
        leaves.push(H256::random());
    }
    let mut merkle = AppendMerkleTree::<H256, Sha3Algorithm>::new(leaves, 0, N
    assert!(!merkle.root().is_null());
    merkle.append(H256::null());
    assert!(merkle.root().is_null());
    for i in 0..(merkle.leaves() - 1) {
        assert!(merkle.gen_proof(i).err().map(|err| err.to_string().contains('
    }
    assert!(merkle.gen_proof(merkle.leaves() - 1).err().map(|err| err.to_strin
    merkle.get_subtrees();
}
```

Executing the PoC produces the following result:

```
~/Doc/p/r/0g-storage-node @53449e1f !20 ?2 ❯ cargo test -p append_merkle --release test_append_with_null -- --nocapture -- --exact
   Compiling append_merkle v0.1.0 (/Users/cyberj0e/Documents/projects/rust/0g-storage-node/common/append_merkle)
    Finished release [optimized] target(s) in 2.10s
     Running unittests src/lib.rs (target/release/deps/append_merkle-dd2d3f5c9d1e810c)

running 1 test
thread 'tests::test_append_with_null' panicked at common/append_merkle/src/lib.rs:574:9:
internal error: entered unreachable code: root is always available
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
test tests::test_append_with_null - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 20 filtered out; finished in 0.00s
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:H/I:C/D:N/Y:N (9.9)

Recommendation

It is recommended to validate that the nodes being appended are not null.

7.3 (HAL-04) OVERWRITING THE LAST NON-NULL LEAF WITH NULL CORRUPTS THE MERKLE TREE

// HIGH

Description

The `AppendMerkleTree::update_last` function updates the last leaf node with the provided value. If this value is null, recomputing the Merkle tree will only update the leaf node itself, leaving its parent nodes and all nodes in the path to the root unchanged. This is because the recompute function does not overwrite existing non-null parent nodes with null values. As a result, if the last leaf was previously non-null, the Merkle tree will experience the following issues:

1. **Inconsistent parent hashes:** The parent node of the affected leaf will no longer match the hash of its children. When a Merkle tree is recomputed and parents are inserted for the first time, if one child is null, the parent is also set to null. Therefore, if a child node becomes null, its parent should reflect that change.
2. **Proof generation issues:** Generating a proof for the sibling of the updated leaf that became null is not possible. The proof lemma cannot include a null hash, which affects the inclusion of the sibling in the proof.
3. **Root node inconsistencies:** Since the root node remains unchanged, reverting to an older version of the Merkle tree with that specific root node will result in a missing entry for the root in the `root_to_tx_seq_map` map. This necessitates an additional `commit` with the correct transaction sequence to update the map accordingly.

Code Location

The `AppendMerkleTree::update_last` function allows updating the last leaf node with a null hash:

```
175 pub fn update_last(&mut self, updated_leaf: E) {
176     if self.layers[0].is_empty() {
177         // Special case for the first data.
178         self.layers[0].push(updated_leaf);
179     } else {
180         *self.layers[0].last_mut().unwrap() = updated_leaf;
181     }
182     self.recompute_after_append_leaves(self.leaves() - 1);
183 }
```

The `AppendMerkleTree::recompute_after_append_leaves` function calls the `recompute` function:

```
365 fn recompute_after_append_leaves(&mut self, start_index: usize) {
366     self.recompute(start_index, 0, None)
367 }
```

The `AppendMerkleTree::recompute` function does not overwrite a parent node with null:

```
for (parent_index, parent) in parent_update {
    match parent_index.cmp(&self.layers[height + 1].len()) {
        Ordering::Less => {
            // We do not overwrite with null.
            if parent != E::null() {
                ...
            }
        }
```

The `merkle_tree::MerkleTreeRead::gen_proof` function does not allow a null hash within the proof's lemma:

```
if lemma.contains(&Self::E::null()) {
    bail!(
        "Not enough data to generate proof, lemma={:?} path={:?}",
        lemma,
        path
    );
}
```

The `AppendMerkleTree::revert_to` function removes the upper layers not present in the old Merkle tree and then invokes the `AppendMerkleTree::clear_after` function. This function clears all root nodes and delta nodes associated with transaction sequences greater than the specified one:

```
pub fn revert_to(&mut self, tx_seq: u64) -> Result<()> {
    ...
    self.clear_after(tx_seq);
    Ok(())
}
```

The `AppendMerkleTree::clear_after` function does not verify whether the root being removed from `root_to_tx_seq_map` is the current root node:

```
fn clear_after(&mut self, tx_seq: u64) {
    let mut tx_seq = tx_seq + 1;
    while self.delta_nodes_map.contains_key(&tx_seq) {
        if let Some(nodes) = self.delta_nodes_map.remove(&tx_seq) {
            if nodes.height() != 0 {
```

```
                    self.root_to_tx_seq_map.remove(nodes.root());
                }
            }
            tx_seq += 1;
        }
    }
```

Proof of Concept

PoC Explanation: The PoC performs the following steps:

1. Construct a balanced Merkle tree consisting of 8 filled leaves
2. Replace the last leaf with a null hash
3. Verify that the nodes along the path to the root remain unchanged
4. Confirm that generating a proof for either the updated null leaf or its sibling is not possible
5. Restore the Merkle tree to its initial version
6. Validate that the current Merkle tree root has no entry in the `root_to_tx_seq_map`

The following test was added to the `common/append_merkle/src/lib.rs` file:

```
#[test]
fn test_update_last_with_null() {
    let nb_non_empty_leaves: usize = 8;
    let mut leaves = Vec::new();
    for _ in 0..nb_non_empty_leaves {
        leaves.push(H256::random());
    }
    let mut merkle = AppendMerkleTree::<H256, Sha3Algorithm>::new(leaves, 0, N
    merkle.commit(Some(0));

    let layer_1 = merkle.layers[1].clone();
    let layer_2 = merkle.layers[2].clone();
    let layer_3 = merkle.layers[3].clone();

    let proof = merkle.gen_proof(6).unwrap();
    assert!(merkle.validate(&proof, &merkle.layers[0][6], 6).unwrap());

    // store the initial root_to_tx_seq_map
    let initial_merkle_root_to_tx_seq_map = merkle.root_to_tx_seq_map.clone();

    // overwrite the last leaf with null
    merkle.update_last(HashElement::null());
    merkle.commit(Some(1));
```

```rust
    // since the last leaf is null, their parent won't be updated with a null
    assert_eq!(merkle.layers[1].clone(), layer_1);
    assert_eq!(merkle.layers[2].clone(), layer_2);
    assert_eq!(merkle.layers[3].clone(), layer_3);

    // generating a proof for the leaf at index 6 (left sibling of the last le
    assert!(merkle.gen_proof(6).err().map(|err| err.to_string().contains("Not
    assert!(merkle.gen_proof(7).err().map(|err| err.to_string().contains("Not

    // reverting back to the initial merkle tree corresponding to the tx_seq e
    merkle.revert_to(0).unwrap();
    assert_ne!(initial_merkle_root_to_tx_seq_map, merkle.root_to_tx_seq_map.cl
    assert!(!merkle.root_to_tx_seq_map.clone().contains_key(merkle.root()));
}
```

Executing the PoC produces the following result:

```
~/Doc/p/r/0g-storage-node @53449e1f !20 ?2 ❯ cargo test -p append_merkle --release test_update_last_with_null -- --nocapture -- --exact
   Compiling append_merkle v0.1.0 (/Users/cyberj0e/Documents/projects/rust/0g-storage-node/common/append_merkle)
    Finished release [optimized] target(s) in 4.07s
     Running unittests src/lib.rs (target/release/deps/append_merkle-dd2d3f5c9d1e810c)

running 1 test
test tests::test_update_last_with_null ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 20 filtered out; finished in 0.00s
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:H/I:M/D:N/Y:N (7.3)

Recommendation

It is recommended to review the logic of the `update_last` function and either implement input validation to prevent setting the last leaf to null, or adjust the `recompute` function to overwrite parent nodes so they accurately reflect their child nodes and ensure the root node is updated when a leaf node changes. This adjustment will help prevent the issues identified in the `clear_after` function when reverting the Merkle tree.

7.4 (HAL-15) USE OF PANIC MACROS IN ERROR HANDLING

// MEDIUM

Description

When handling errors, it is generally preferable to return an error and handle it gracefully, rather than using panics, which can cause the calling service(s) to crash. However, in the `common/zgs_seal` and `common/append_merkle` modules, a few panic macros were used for error handling. While the likelihood of these assertions failing requires certain uncommon conditions, if they do fail, they could cause crashes in dependent services, particularly those relying on the `storage` module connected to these modules.

Code Location

The `assert!` and `assert_eq!` macros, which trigger a panic on failure, were utilized for error handling in the following locations:

```
common/zgs_seal/src/lib.rs
25: assert!(data.len() % 32 == 0);
34: assert!(data.len() % 32 == 0);

common/append_merkle/src/proof.rs
15:  assert_eq!(hash.len() - 2, path.len());
225: assert_eq!(children_layer.len(), 1);

common/append_merkle/src/lib.rs
352: assert_eq!(root, right_most_nodes.last().unwrap().1);
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:M/I:N/D:N/Y:L (4.7)

Recommendation

It is recommended to replace the panic macro calls with appropriate error-handling mechanisms that safely log errors and allow the calling service to continue functioning.

7.5 (HAL-03) POSSIBILITY OF UNREGISTERED MINERS PARTICIPATING IN PORA

// LOW

Description

When a miner starts, the mine service first checks if they have a miner ID. If not, it requests one from the Mine contract on-chain before initiating their mining-related services. Within the `miner_id::check_and_request_miner_id` function, if the miner's database ID matches the ID stored in their configuration, the ID is considered valid. However, there is still a possibility that the miner ID may not be registered on-chain.

The miner ID in each miner's configuration is manually set through the CLI when starting the miner node, making it easily configurable. In contrast, the miner ID stored in the database can only be set by invoking the `miner_id::set_miner_id` private function, which is called in the `miner_id::check_and_request_miner_id` function in two scenarios:

1. The miner lacks an ID in the database, but the ID in their configuration is registered on-chain.
2. The miner has neither an ID in the database nor in their configuration, prompting a request for a new miner ID on-chain, which is then saved to the database.

As a result, the only situation in which an unregistered miner ID can end up in a miner's database is if a previously registered ID becomes unregistered. This can occur if the miner accidentally transfers ownership of their miner ID to the zero address by calling Mine.transferBeneficial, which is unlikely.

Code Location

The `service::MineService::spawn` function calls the `check_and_request_miner_id` function:

```
30  pub async fn spawn(
31      executor: task_executor::TaskExecutor,
32      _network_send: mpsc::UnboundedSender<NetworkMessage>,
33      config: MinerConfig,
34      store: Arc<Store>,
35  ) -> Result<broadcast::Sender<MinerMessage>, String> {
36      let provider = Arc::new(config.make_provider().await?);
37
38      let (msg_send, msg_recv) = broadcast::channel(1024);
39
40      let miner_id = check_and_request_miner_id(&config, store.as_ref(), &p
41      debug!("miner id setting complete.");
42      ...
```

The `miner_id::check_and_request_miner_id` function does not verify that the miner ID is registered on-chain, even if the ID in the database matches the one in the miner's configuration:

```
21  pub(crate) async fn check_and_request_miner_id(
22      config: &MinerConfig,
23      store: &Store,
24      provider: &Arc<MineServiceMiddleware>,
25  ) -> Result<H256, String> {
26      let db_miner_id = load_miner_id(store)
27          .await
28          .map_err(|e| format!("miner_id on db corrupt: {:?}", e))?;
29
30      let mine_contract = PoraMine::new(config.mine_address, provider.clone
31
32      match (db_miner_id, config.miner_id) {
33          (Some(d_id), Some(c_id)) => {
34              if d_id != c_id {
35                  Err(format!(
36                      "database miner id {} != configuration miner id {}",
37                      d_id, c_id
38                  ))
39              } else {
40                  Ok(d_id)
41              }
42          }
43      ...
```

BVSS

AO:A/AC:L/AX:H/R:N/S:C/C:C/A:N/I:N/D:N/Y:N (4.1)

Recommendation

It is recommended to verify whether the miner ID is registered on-chain, even if there is a matching miner ID in both the database and the miner's configuration.

7.6 (HAL-06) DISABLED OVERFLOW CHECKS AND USAGE OF UNCHECKED MATH OPERATIONS

// LOW

Description

In computer programming, an overflow occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value. This is a good practice, since having `overflow-checks` enabled in the workspace would avoid overflow situations in `release` mode.

In the current implementation, overflow issues from arithmetic operations or downcasting are not caught in release mode and will cause a panic otherwise, leading to potential crashes.

Code Location

Most arithmetic operations have been reviewed and are not prone to overflow. However, the following examples have been identified as potential sources of overflow.

In the `recall_range::load_position` function, the returned recall position is calculated using unchecked arithmetic operations, an overflow would require the start position to be very large and close to `u64::MAX`, where `SECTORS_PER_LOAD` equals `1024`:

```
33 pub fn load_position(&self, seed: [u8; 32]) -> Option<u64> {
34     let (_, origin_recall_offset) = U256::from_big_endian(&seed)
35         .div_mod(U256::from((self.mining_length as usize) / SECTORS_PER_L
36     let origin_recall_offset = origin_recall_offset.as_u64();
37     let recall_offset = (origin_recall_offset & self.shard_mask) | self.s
38 
39     Some(self.start_position + recall_offset * SECTORS_PER_LOAD as u64)
40 }
```

In the `pora::iteration` function, the recall position is first calculated using `recall_range::load_position`, and it is then incremented at the end as follows:

```
recall_position: recall_position + idx as u64 * SECTORS_PER_SEAL as u64,
```

In the `sealer::fetch_context` function, `context.end` is downcast from `u128` to `u64` twice. If `context.end` holds a value exceeding `u64::MAX`, this downcasting will cause an overflow:

```
context.end as u64 / SECTORS_PER_SEAL as u64,
```

It's worth noting that the context is updated between the two downcasts:

```
127 let context = match self
128     .flow_contract
129     .query_context_at_position(last_entry)
130     .call()
131     .await
132 {
133     Ok(context) => context,
134     Err(err) => {
135         info!("Error when fetch entries {:?}", err);
136         return Ok(None);
137     }
138 };
```

The on-chain call sets the context's `end` to `tree.currentLength`, representing the current length of the tree. As the tree grows over time, there remains a possibility that it could eventually become large enough to cause an overflow in the distant future.

BVSS

AO:A/AC:L/AX:H/R:N/S:C/C:N/A:M/I:H/D:N/Y:N (3.6)

Recommendation

It is recommended to enable overflow checks when building the project in release mode and to utilize checked arithmetic operations to handle overflow errors gracefully.

7.7 (HAL-07) MISMATCH BETWEEN DOCUMENTATION AND CODE IN PORA HASH CALCULATION

// LOW

Description

Within a Proof of Random Access (PoRA) iteration, there is a mismatch between the documentation and the implementation of the PoRA hash calculation.

Documentation:
Regard `mixedData` *as an array of* `SEAL_SIZE`*-byte elements. For each segment* `y` *with index* `i`*, compute* `poraHash` *and check if it reaches* `targetQuality`*.*

```
ZERO64 = 64 bytes filled with 0
# The hash function here should be blake2b.
poraHash = Hash(i, minerId, nonce, contextDigest, startPosition, miningRange, ZERO64, y)
```

With the given details, the formula can be interpreted as: `poraHash = Hash(sealIndex, minerId, nonce, contextDigest, startPosition, miningRange, ZERO64, mixedData)`.
Implementation: Could be interpreted as: `poraHash = Hash(ZERO24, seal_index, pad_seed, ZERO32, mixed_data)` given the code below:

```rust
#[inline]
fn pora(
    &self,
    seal_index: usize,
    mixed_data: &[u8; BYTES_PER_SEAL],
    pad_seed: [u8; BLAKE2B_OUTPUT_BYTES],
) -> U256 {
    let mut hasher = Blake2b512::new();
    hasher.update([0u8; 24]);
    hasher.update(seal_index.to_be_bytes());

    hasher.update(pad_seed);
    hasher.update([0u8; 32]);

    hasher.update(mixed_data);

    let digest = hasher.finalize();
```

```
184 |       U256::from_big_endian(&digest[0..32])
    }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:L/D:N/Y:N (3.1)

Recommendation

It is recommended to review the logic of the PoRA hash computation and align either the documentation or the implementation to ensure consistency.

7.8 (HAL-05) AN UNKNOWN LEAF NODE CAN BE FILLED WITH NULL

// LOW

Description

The `AppendMerkleTree::fill_leaf` function's purpose is to fill an unknown `null` leaf with its real value, according to the documentation. However, it's possible to fill an unknown leaf with a null value, which deviates from the intended behavior and leads to unnecessary recomputation of the Merkle tree.

Code Location

The `AppendMerkleTree::fill_leaf` function does not validate that the new leaf node hash is non-null:

```
188 pub fn fill_leaf(&mut self, index: usize, leaf: E) {
189         if self.layers[0][index] == E::null() {
190             self.layers[0][index] = leaf;
191             self.recompute_after_fill_leaves(index, index + 1);
192         } else if self.layers[0][index] != leaf {
193             panic!(
194                 "Fill with invalid leaf, index={} was={:?} get={:?}",
195                 index, self.layers[0][index], leaf
196             );
197         }
198     }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:L (2.1)

Recommendation

It is recommended to validate that the new leaf value is non-null within the `AppendMerkleTree::fill_leaf` function.

7.9 (HAL-10) INACCURATE ERROR MESSAGES

// INFORMATIONAL

Description

The following error messages do not accurately describe the encountered issues:

- `Fail to find minerId in receipt`: This error message is returned when finding empty logs within the transaction receipt for requesting a new miner id.
- `Fail to execute mine answer transaction` and `Mine answer transaction dropped after {} retires`: These error messages are returned after failing to execute the `submit` transaction instead of `answer`.
- `Fetch onchain context failed`: This warning message is returned when failing to update the flow length.
- `Error when fetch entries`: This error message is returned when fetching context at a given position.

Although this is not a security vulnerability, the misleading error messages can confuse the miners.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

It is recommended to update the error messages to accurately reflect the underlying causes of the errors.

7.10 (HAL-08) INEFFECTIVE COMPUTATION

// INFORMATIONAL

Description

During a PoRA iteration, if the miner finds a valid answer, a debug message is printed showing both the quality of the answer and the target quality to surpass. However, an ineffective computation was observed in the `pora::iteration` function:

```
114 | difficulty_scale_x64.as_u128() as f64 / (u64::MAX as f64 + 1.0)
```

For `u64` values greater than `2^53`, `f64` starts to lose precision because it can only store a limited number of significant digits. In our case, adding `1.0` to `u64::MAX as f64` has no impact on the result. Both the original value and the incremented value are represented as `18446744073709552000` in `f64`.

Score

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove the ineffective computation by changing `(u64::MAX as f64 + 1.0)` to `(u64::MAX as f64)`.

7.11 (HAL-09) INEFFICIENT CODE

// INFORMATIONAL

Description

In the codebase, several instances of inefficient code were identified.

Code Location

In the`recall_range::load_position` function, the `a.div_mod(b)` function which returns the pair of `(a div b, a mod b)`was utilized. However, the `div` result is omitted (using `_`) and only the `mod` result is being used. In this case, using the `mod` function is more adequate.

```
33 | pub fn load_position(&self, seed: [u8; 32]) -> Option<u64> {
34 |     let (_, origin_recall_offset) = U256::from_big_endian(&seed)
35 |         .div_mod(U256::from((self.mining_length as usize) / SECTORS_PER_L
```

At the beginning of the `MineContextWatcher::spawn` function, the following variable was declared unnecessarily:

```
49 | let provider = provider;
```

In the `miner::is_covered` function, both the start and end positions of a mine range are of type `Option<u64>`. Initially, the function checks if both positions are `Some` values and, if so, assigns their inner `u64` values to local variables. Instead of comparing the `Option<u64>` types directly, which involves additional overhead due to pattern matching for `Some` or `None`, the function should compare these inner `u64` values directly, which are `self_start_position` and `self_end_position`. This approach avoids the extra step of matching the `Option` enum, thus optimizing the comparison by leveraging the fact that both positions are guaranteed to hold `u64` values.

```
94 | let self_start_position = self.start_position?;
95 | let self_end_position = self.end_position?;
96 |
97 | if self.start_position >= self.end_position {
98 |     return Some(false);
99 | }
```

In the `Proof::proof_nodes_in_tree` function, the returned vector will include the proof root and additional elements from the proof lemma alongside their indexes within their respective layers. This is achieved by pushing the proof root then iterating over the proof path elements as follows:

```rust
pub fn proof_nodes_in_tree(&self) -> Vec<(usize, T)> {
    let mut r = Vec::with_capacity(self.lemma.len());
    let mut pos = 0;
    r.push((0, self.root()));
    for (i, is_left) in self.path.iter().rev().enumerate() {
        pos <<= 1;
        if !*is_left {
            pos += 1;
        }
        let lemma_pos = if *is_left { pos + 1 } else { pos - 1 };
        r.push((lemma_pos, self.lemma[self.lemma.len() - 2 - i].clone()))
    }
    r.reverse();
    r
}
```

It's evident that the returned vector will contain `self.path.len() + 1` elements. This is equivalent to `self.lemma.len() - 1`, as the proof lemma always includes two more elements than the proof path.

In the `Proof::file_proof_nodes_in_tree` function, the returned vector will include elements from the proof lemma alongside their indexes within their respective layers. This is achieved by iterating over the proof path elements as follows:

```rust
pub fn file_proof_nodes_in_tree(
    &self,
    tx_merkle_nodes: Vec<(usize, T)>,
    tx_merkle_nodes_size: usize,
) -> Vec<(usize, T)> {
    let mut r = Vec::with_capacity(self.lemma.len());
    ...
    for (i, is_left) in self.path.iter().rev().enumerate() {
        if !in_subtree {
            ...
        } else {
            ...
            r.push((lemma_pos, self.lemma[self.lemma.len() - 2 - i].clone
        }
    }
    r.reverse();
    r
}
```

It's clear that the length of the returned vector will not exceed `self.path.len()`, although it was created with a larger capacity.

Score

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to:

- Use `mod` instead of `div_mod` when the `div` result is not needed.
- Remove the unnecessary variable declaration.
- Compare the inner `u64` values of the mine range's start and end positions directly using `self_start_position` and `self_end_position`.
- Set the appropriate capacity for the vectors in question.

7.12 (HAL-11) PRESENCE OF TODOS IMPLYING UNFINISHED CODE

// INFORMATIONAL

Description

Open TODOs can point to architecture or programming issues that still need to be resolved.

Code Location

The following TODO comments were identified in the modules in scope:

```
node/miner/src/submitter.rs
181: // TODO: The conversion will be simpler if we optimize range proof struct

common/append_merkle/src/merkle_tree.rs
96:  // TODO: This can be skipped if the tree size is available in validation.
126: // TODO(zz): Optimize range proof.

common/append_merkle/src/lib.rs
59:  // TODO(zz): Check when the roots become available.
94:  // TODO: Delete duplicate nodes from DB.
156: /// TODO: Optimize to avoid storing the `null` nodes?
187: /// TODO: Batch computing intermediate nodes.

common/append_merkle/src/proof.rs
199: // TODO: We can avoid copying the first layer.
```

Score

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to resolve the open TODOs.

7.13 (HAL-12) PRESENCE OF TYPOS

// INFORMATIONAL

Description

Within the modules in scope, several typographical errors were identified.

Code Location

The following typographical errors were identified:

```
node/miner/src/pora.rs
L82: avalibilities,
L95: .zip(avalibilities.into_iter())
L96: .filter_map(|(data, avaliable)| avaliable.then_some(data))
L117: // Undo mix data when find a valid solition

node/miner/src/submitter.rs
L21: const SUBMISSION_RETIES: usize = 15;
L170: "Mine answer transaction dropped after {} retires",

node/miner/src/miner_id.rs
L102: .ok_or("Request miner id transaction dropped after 3 retires")?;

common/append_merkle/src/proof.rs
L61: bail!("Proof item unmatch");
L64: bail!("Proof position unmatch");
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to correct the identified typos:

```
avalibilities -> availabilities
avaliable -> available
solition -> solution
RETIES -> RETRIES
retires -> retries
unmatch -> mismatch
```

7.14 (HAL-13) MISLEADING DOCUMENTATION AND COMMENTS

// INFORMATIONAL

Description

Incorporating documentation and comments into the codebase is essential for elucidating key aspects and facilitating comprehension of the developer's intentions by others. However, the codebase contains misleading documentation and comments.

Code Location

In the `common/append_merkle/src/lib.rs` file, the `AppendMerkleTree::fill_leaf` function is documented as follows: `/// Panics if the leaf changes the Merkle root or the index is out of range.`. However, the function does not actually panic if the Merkle root is changed.

Additionally, `AppendMerkleTree::append` function is documented as follows: `/// Return the new merkle root.`. However, the function's purpose is appending a new leaf to the Merkle tree, without returning the new Merkle root.

The `node/miner/src/mine.rs` file contains the following remark: `// 2^64 ns = 500 years`. However, `2^64` nanoseconds is approximately 584 years.

Score

Impact:

Likelihood:

Recommendation

It is recommended to correct the misleading comments for accuracy.

8. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in `https://crates.io` are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the `cargo audit` output to better know the dependencies affected by unmaintained and vulnerable crates.

ID	PACKAGE	SHORT DESCRIPTION
RUSTSEC-2024-0344	curve25519-dalek 3.2.0	Timing variability in `curve25519-dalek`'s `Scalar29::sub`/`Scalar52::sub`
RUSTSEC-2022-0093	ed25519-dalek 1.0.1	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
RUSTSEC-2022-0040	owning_ref 0.4.1	Multiple soundness issues in `owning_ref`
RUSTSEC-2023-0018	remove_dir_all 0.5.3	Race Condition Enabling Link Following and Time-of-check Time-of-use (TOCTOU)
RUSTSEC-2024-0336	rustls 0.20.9	`rustls::ConnectionCommon::complete_io` could fall into an infinite loop based on network input
RUSTSEC-2024-0370	proc-macro-error 1.0.4	proc-macro-error is unmaintained
RUSTSEC-2018-0017	tempdir 0.3.7	`tempdir` crate has been deprecated; use `tempfile` instead
RUSTSEC-2024-0320	yaml-rust 0.4.5	yaml-rust is unmaintained
RUSTSEC-2021-0145	atty 0.2.14	Potential unaligned read
	bytes 1.6.0	yanked

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.