Orga and Merk

Security Assessment

November 19, 2024

Prepared for:
Matt Bell and Judd Keppel
Turbofish

Prepared by: **Anish Naik and Tjaden Hess**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Project Summary **4**
Executive Summary **5**
Project Goals **7**
Project Targets **8**
Project Coverage **9**
Automated Testing **12**
Codebase Maturity Evaluation **13**
Summary of Findings **15**
Detailed Findings **16**
1. Slashing of re-delegated stake is computed incorrectly 16
2. Malicious state sync peer can cause syncing nodes to crash 18
3. Interrupted snapshots can lead to inconsistent state 19
4. Malicious state sync peer can cause a stack overflow in Merk 21
5. Merk trunk splitting can lead to panics on degenerate trees 22
6. Stored IBC consensus states cannot be pruned 24
7. Merk proofs can be forged to claim arbitrary key/value inclusions 27
A. Vulnerability Categories **29**
B. Code Maturity Categories **31**
C. System Invariants **33**
D. Non-Security-Related Findings **34**
E. Automated Testing **36**
F. Fix Review Results **37**
Detailed Fix Review Results 38
G. Fix Review Status Categories **39**

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 6, 2024	Pre-project kickoff call
September 18, 2024	Status update meeting #1
September 27, 2024	Status update meeting #2
October 4, 2024	Status update meeting #3
October 14, 2024	Delivery of report draft
October 15, 2024	Report readout meeting
November 19, 2024	Delivery of comprehensive report with fix review addendum

Executive Summary

Engagement Overview

Turbofish engaged Trail of Bits to review the security of Orga, a deterministic state machine engine written in Rust, and Merk, a high-performance Merkle key-value store. Additionally, we reviewed dependencies including `merk`, a custom Merkle tree implementation for state management, and `ed`, a library for custom serialization.

This report covers the `orga`, `ed`, `abci2`, and `merk` libraries. We also reviewed the Nomic and `nomic-bitcoin` libraries; those findings are provided in a separate report.

A team of two consultants conducted the review from September 3 to October 11, 2024, for a total of 10 engineer-weeks of effort. Our testing efforts focused on two core areas. First, we investigated whether it was possible to bypass or corrupt the intended state-transition behavior of Orga state machines. Second, we investigated denial-of-service (DoS) attack vectors that could bring the state machine to a halt or prevent new users from joining the network. With full access to source code and documentation, we performed static and dynamic testing of the targets using automated and manual processes.

Observations and Impact

The Turbofish libraries are generally of high quality but also of high complexity. We identified one instance in `merk` that would allow for proof forgery, which could lead to a theft of funds in dependent applications (TOB-TF-7). However, `orga` does not currently use the vulnerable verification routine.

A majority of the issues identified during the audit revolve around the state syncing mechanism and the Merkle tree implementation. As shown by TOB-TF-3, the state syncing flow is not thoroughly tested. Similarly, TOB-TF-2 highlights that the current implementation does not assume that the snapshot provider can be a malicious actor. Although the DoS attack targets a syncing node, testing for this mechanism should assume that the snapshot provider may be malicious. The `merk` implementation also suffers from insufficient edge case testing (TOB-TF-5). It is important to note that `merk` is agnostic to the system that uses it. Thus, even though heavily skewed trees do not appear in current use cases, the implementation must be capable of handling trees that are of arbitrary form.

Additionally, the documentation provided was insufficient, given the complexity of the system. It was difficult to reason through the use case of the various macros, the functionality of various modules, and the end-to-end flow of transactions. Thorough documentation aids in maintaining the codebase while also aiding in future security reviews.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Turbofish take the following steps prior to deployment:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Improve edge case testing of `merk`.** Ensure that the `merk` implementation is well-tested against heavily skewed trees. Add testing for proof verification functions to exercise the implementation in case of degenerate linked-list-like or very skewed Merkle trees.
- **Improve state sync testing.** We identified a variety of edge cases in the state syncing logic. Improving testing to ensure that the ABCI specification is met and that the system can handle the possibility of a malicious snapshot provider would be beneficial.
- **Improve documentation.** The documentation for the system is insufficient. It would be beneficial to invest time into developing high-level walkthroughs, examples, and diagrams to showcase the interactions between components of the system. Additionally, enriching the inline documentation to highlight each module's and function's expected behavior would be beneficial.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	1
Low	5
Informational	0
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	3
Denial of Service	3
Testing	1

Project Goals

The engagement was scoped to provide a security assessment of the Turbofish system. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can malicious parties cause the Orga state machine to misallocate funds?
- Can a minority of stakeholders gain undue control of the state machine?
- Are there DoS attack vectors that could halt the forward progress of the state machine or prevent node synchronization?
- Are third-party protocols such as IBC or Tendermint integrated correctly?
- Can a malicious attack spoof the existence of a node in the Merk tree?
- Can malicious inputs cause serialization/deserialization routines to trigger a panic?
- Do the arithmetic calculations in the system account for potential overflows, precision loss, and/or division by zero?
- Does the state machine manage accounts, balances, signatures, and fees correctly?
- Are the uses of cryptographic signatures, hashes, and commitments vulnerable to collisions or replays?

Project Targets

The engagement involved a review and testing of the targets listed below.

orga

Repository	https://github.com/turbofish-org/orga
Version	`53595ff`
Type	Rust
Platform	Linux, macOS, Windows

merk

Repository	https://github.com/turbofish-org/merk
Version	`cc49630`
Type	Rust
Platform	Linux, macOS, Windows

ed

Repository	https://github.com/turbofish-org/ed
Version	`a657be8`
Type	Rust
Platform	Linux, macOS, Windows

abci2

Repository	https://github.com/turbofish-org/abci2
Version	`27d8d7a`
Type	Rust
Platform	Linux, macOS, Windows

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`orga`.** `orga` is a deterministic state machine that powers the Nomic blockchain. We performed a manual review of the system as follows:
    - We reviewed the integration with the Tendermint and ABCI protocol, assessing whether all ABCI method calls are implemented correctly and provide the necessary information to the underlying application (Nomic). This led to the discovery of TOB-TF-3, which highlights that incoming snapshots do not reset the current sync state.
    - We reviewed the state syncing mechanism, to identify whether a malicious validator can launch a DoS attack on a syncing node. This led to the discovery of TOB-TF-2, which highlights that incoming snapshots could cause the tendermint node to crash or lead to the banning of honest sync nodes.
    - We reviewed the state management data structures and KV store to identify opportunities for key collisions (e.g., via incorrect variable-length encoding) or to trigger a panic, and to assess its general correctness.
    - We reviewed the fee accounting mechanism to determine if an attacker can avoid paying for transactions or launch a DoS attack.
    - We reviewed the integration with the IBC protocol for general correctness (e.g., packet timeouts) and to assess whether the system is robust against light client attacks. This led to the discovery of TOB-TF-6, which highlights that consensus states are not pruned in the client, which may lead to excessive memory and storage consumption over time.
    - We reviewed the integration with `merk` to investigate if read and write operations to the primary and ancillary stores lead to unexpected edge cases.
    - We reviewed the various data structures provided by `orga` out-of-box to assess whether it allows for arbitrary minting of coins, whether the arithmetic of reward distribution/slashing/delegations is valid, and whether there are unexpected edge cases that would allow for the data structures to be misused or cause a panic.
    - We reviewed the signature validation for general correctness and to determine if there is any possibility of hash collisions.

- We reviewed the ICS-23 implementation to check for proof forgeries and path collisions.

- **`merk`.** `merk` is a Merkle tree-based key/value store using an AVL tree based on RocksDB. `merk` provides storage, proofs, verification, snapshotting, and restoration. We performed a manual review of the system as follows:

  - We reviewed the Merkle tree hash structure to check for proper domain separation and encoding injectivity, determining if the Merkle root serves as a collision-resistant commitment to both the content and structure of the tree.

  - We reviewed the batch insertion functionality to assess whether the AVL and BST invariants are preserved under arbitrary, potentially malicious sequences of key insertions and deletions.

  - We reviewed the proof generation logic to check for edge-cases that could prevent the generation of valid proofs due to skewed or maliciously constructed trees.

  - We reviewed the proof verification and tree reconstruction logic to assess whether inclusion and exclusion proofs are sound. This led to the discovery of TOB-TF-7, which highlights that a malicious actor could prove the inclusion of any arbitrary key-value pair in a target tree root. Additionally, we verified that `Map::range` iterator is cryptographically sound and will return an error if it cannot guarantee the inclusion or exclusion of all keys in the query range.

  - We reviewed the snapshot and chunk generation functionality, checking whether all valid trees can be decomposed into chunks without panics.

  - We reviewed the snapshot and chunk restoration functionality. We found several issues (TOB-TF-2, TOB-TF-3, TOB-TF-4, TOB-TF-5) that could lead to panic during snapshot restoration.

- **`ed`.** The `ed` library provides a simple deterministic serialization/deserialization abstraction along with implementations for common types and derive-macros for custom structs and enums.

  - We reviewed the provided implementations to determine if the serialization/deserialization is deterministic and can trigger panics on untrusted inputs.

  - We reviewed the derive-macros to assess whether they preserve termination constraints and thus produce deterministic serialization/deserialization routines.

- We reviewed the codebase for potentially adversarially controlled allocations that could lead to DoS by memory depletion.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code
Cargo Audit	An open-source tool for checking dependencies against the RustSec advisory database

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The arithmetic calculations performed across the various targets are of generally high quality. We did not find any issues related to arithmetic overflows, division by zero, or precision loss. It is important to note that overflow checks are turned on in production, which could result in panics in case of an overflow.	**Strong**
Auditing	There was sufficient logging throughout the system. All ABCI methods emit the expected events, and there is sufficient use of logging to alert on unexpected behavior.	**Satisfactory**
Authentication / Access Controls	We did not identify any attack vectors that could lead to an access control bypass. The critical code paths have sufficient data validation to ensure that only the authorized actors can perform the actions.	**Strong**
Complexity Management	The Orga system has a complex architecture with many components. Within Orga, there are many layers of abstraction (e.g., heavy use of macros and traits) that make it difficult to reason through the system's critical and expected workflows.	**Moderate**
Cryptography and Key Management	We did not identify any instances of misused cryptographic primitives that would cause undefined behavior or trigger an edge case in the state machine.	**Moderate**
Data Handling	We found two issues related to insufficient data validation in `merk`, one of which is of high severity (TOB-TF-7). The other components of the system have sufficient data validation. Function inputs are sufficiently validated, and we identified no edge cases in the serialization and deserialization of custom types.	**Moderate**

Decentralization	The Orga state machine blockchain is fully decentralized. There is no single actor with special permissions unless defined by the higher-level application.	**Strong**
Documentation	The system documentation is of poor quality. Given the system's high level of complexity and layers of abstraction, we recommend creating thorough documentation highlighting the transaction lifecycle, the function and behavior of each module, and the interactions across components. The inline documentation is thorough in some places but is generally sparse.	**Weak**
Memory Safety and Error Handling	The `orga` `Context` type uses unsafe transmutations to implement an inhomogeneous map. The usage of `TypeId`-based keys should prevent any undefined behavior. We found one issue (TOB-TF-2) that resulted from improper handling of errors, specifically raising exceptions rather than returning an `Err` type. Error handling was otherwise good, avoiding `unwrap` and `expect` except in cases where it is clear that a value will be present.	**Satisfactory**
Testing and Verification	`merk` would benefit from additional testing surrounding inclusion proofs (TOB-TF-7) and the testing of heavily skewed or unbalanced trees (TOB-TF-5). Similarly, the state syncing process needs to be better tested to prevent a malicious snapshot provider from crashing a syncing node (TOB-TF-2 and TOB-TF-3). Additionally, TOB-TF-1 showcases that care must be taken when choosing the input parameters for a unit/integration test. Finally, TOB-TF-6 showcases that additional testing of symmetric operations should be performed.	**Moderate**
Transaction Ordering	We did not identify any issues related to front-running or transaction ordering risks.	**Strong**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Slashing of re-delegated stake is computed incorrectly	Testing	**Medium**
2	Malicious state sync peer can cause syncing nodes to crash	Denial of Service	**Low**
3	Interrupted snapshots can lead to inconsistent state	Data Validation	**Low**
4	Malicious state sync peer can cause a stack overflow in Merk	Denial of Service	**Low**
5	Merk trunk splitting can lead to panics on degenerate trees	Denial of Service	**Low**
6	Stored IBC consensus states cannot be pruned	Data Validation	**Low**
7	Merk proofs can be forged to claim arbitrary key/value inclusions	Data Validation	**High**

Detailed Findings

1. Slashing of re-delegated stake is computed incorrectly

Severity: **Medium**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-TF-1
Target: `orga/src/coins/staking`	

Description

When a validator is slashed due to double-signing, any stake delegated to that validator at the time of the misbehavior must also be slashed. The `orga` staking module allows delegators to transfer delegated coins from one validator to another; these coins are tracked so that the proper amount can be deducted if the original validator is slashed.

Due to a miscalculation in the staking module, coins that are slashed while in the process of being re-delegated will forfeit an incorrect percentage of funds. For example, if the `slash_fraction_double_sign` percentage is `1/20`, the re-delegated stake will be slashed by 95% rather than the intended 5%.

Figure 1.1 contains the code responsible for slashing redelegations. Note that the computed value (`multiplier * redelegation_amount`) is equal to the intended final balance after slashing. However, inside `slash_redelegation` (figure 1.2), this parameter is treated as an amount to deduct.

```
let multiplier = (Decimal::one() - self.slash_fraction_double_sign)?;
for entry in redelegations.iter() {
    let del_address = entry.delegator_address;
    for redelegation in entry.outbound_redelegations.iter() {
        let mut validator = self.validators.get_mut(redelegation.address.into())?;
        let mut delegator = validator.get_mut(del_address.into())?;
        delegator.slash_redelegation((multiplier *
redelegation.amount)?.amount()?)?;
    }
}
```

Figure 1.1: Input to `slash_redelegation` pre-deducts the slash amount (`orga/src/coins/staking/mod.rs#503–511`)

```
pub(super) fn slash_redelegation(&mut self, amount: Amount) -> Result<()> {
    let stake_slash = if amount > self.staked.shares.amount()? {
        self.staked.shares.amount()?
    } else {
        amount
    };

    if stake_slash > 0 {
        self.staked.take(stake_slash)?.burn();
    }
```

Figure 1.2: `slash_redelegation` burns the input amount (orga/src/coins/staking/delegator.rs#124–133)

This miscalculation was not detected by the unit test suite because all unit testing is done with a `slash_fraction_double_sign` of ½, which is the unique value x for which `1 - x = x`.

Exploit Scenario

Mallory, a malicious validator, encourages honest users to delegate their staking tokens to her. She wants to manipulate the price of the staking token by artificially burning the supply. Mallory tells her delegators that she will be switching accounts and encourages them to redelegate to another address. During the redelegation period, Mallory intentionally produces a double-sign fault and is slashed. Mallory loses only 5% of her own stake, but all of her delegators lose 95% of theirs, allowing Mallory to manipulate the overall supply at only a minor cost to herself.

Recommendations

Short term, compute the `slash_redelegation` amount by directly multiplying `slash_fraction_double_sign` by the redelegated amount.

Long term, modify the test suite to use a value other than ½ for the slash fraction.

2. Malicious state sync peer can cause syncing nodes to crash

Severity: **Low**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-TF-2
Target: `merk/src/merk/restore.rs`	

Description
A malicious node can cause syncing peers to crash, preventing the syncing peer from using state sync to join the network.

```
// FIXME: this one shouldn't be an assert because it comes from a peer
assert_eq!(self.stated_length, chunks_remaining + 1);
```

Figure 2.1: Malicious peers can cause a panic during state sync. (merk/src/merk/restore.rs#176–177)

This issue was already noted in a `FIXME` comment in the client codebase but was not tracked by a corresponding GitHub issue. The assertion can be triggered by a malicious peer that advertises a snapshot with a valid app hash but provides an incorrect `Snapshot::chunks` value.

Exploit Scenario
A malicious party wants to harm the overall health of an Orga-based network by preventing new nodes from joining. The malicious node advertises snapshots and serves invalid length data. Nodes that attempt to sync from this peer crash, harming the network's reputation and reducing user onboarding success.

Recommendations
Short term, return an `Err` value if the stated length and computed lengths do not match, rather than panicking.

Long term, document any uses of `assert!` with comments explaining why the assertion can never be violated.

3. Interrupted snapshots can lead to inconsistent state

Severity: Low	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TF-3
Target: `orga/src/merk/store.rs`	

Description

During the state sync process, Tendermint may choose to stop synchronizing an in-progress snapshot and begin downloading another.

```
If Tendermint is unable to retrieve the next chunk after some time (e.g. because no suitable
peers are available), it will reject the snapshot and try a different one via OfferSnapshot.
The application should be prepared to reset and accept it or abort as appropriate.
```

Figure 3.1: Excerpt from the ABCI specification

The Orga state sync flow does not properly clear the existing sync data before beginning to download a new snapshot, leading to panics and inconsistent on-disk state.

If an `OfferSnapshot` request is received while a state sync is already in progress, the current implementation will accept the incoming snapshot but fail to reset the `Restorer` state. New chunks from honest peers will then be rejected, as they fail to verify against the partially constructed tree.

Additionally, if a malicious node continues to serve old chunks under the guise of the new snapshot, synchronization will appear to complete correctly, but the recorded block height will not match the app hash. Tendermint will detect this when validating the sync via the `Info` endpoint and panic. The node will not be able to start successfully until the corrupted state directory is deleted.

Exploit Scenario

A node begins to synchronize with an Orga-based blockchain. The node experiences a network fault. When the node synchronization resumes, Tendermint chooses to switch to a new snapshot. However, the inconsistent internal state prevents the new snapshot from applying to the current partial Merk trunk, causing the client to ban all peers and hang indefinitely.

Recommendations

Short term, instantiate and reset the `MerkStore::restorer` instance inside of the `offer_snapshot` method rather than the `apply_snapshot_chunk` method. This will reset the sync state whenever a new snapshot proposal is accepted.

Long term, add unit testing for ABCI state sync request and response flows. Include edge cases such as synchronization restarts.

4. Malicious state sync peer can cause a stack overflow in Merk	
Severity: **Low**	Difficulty: Medium
Type: Denial of Service	Finding ID: TOB-TF-4
Target: `merk/src/proofs/chunk.rs`	

Description
When a node begins processing an incoming Merk snapshot, the node computes the total height of the tree by traversing the reconstructed tree's leftmost path. If the tree is maliciously constructed, this path may be very long, causing a stack overflow in the recursive verification routine.

The `verify_height_proof` and `verify_completeness` functions use recursion to traverse a tree provided by the state sync peer. The `verify_trunk` function (which in turn calls `verify_height_proof and verify_completeness`) is executed before the root hash is verified against the Tendermint-provided `app_hash`, so there is no guarantee that the tree being processed is a valid state tree derived from the blockchain consensus.

Exploit Scenario
A malicious party wants to harm the overall health of an Orga-based network by preventing new nodes from joining. The malicious node advertises snapshots and serves a Merk proof for a degenerate tree consisting of a single path with 32,000 nodes. Nodes that attempt to sync from this peer crash, harming the network's reputation and reducing user onboarding success.

Recommendations
Short term, verify that the trunk's root hash matches the consensus state app hash before further processing the snapshot.

Long term, rewrite all functions that handle externally provided data to be iterative rather than recursive.

5. Merk trunk splitting can lead to panics on degenerate trees

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-TF-5
Target: `merk/src/merk/restore.rs`	

Description

If the Merk AVL tree becomes extremely skewed, all nodes attempting to use state sync to synchronize with the blockchain will panic.

Merk restores its AVL Merkle tree from a snapshot in two steps. The first chunk of the snapshot is the "trunk," comprising the top half of the tree and placeholders for the remaining subtrees to be attached to the trunk's leaves. Each later chunk fills in one of the subtrees on the trunk.

The `merk` library assumes that all trunks will be perfect binary trees, with each leaf representing a nonempty subtree. However, an off-by-one error means that a valid AVL tree may not be complete at the trunk truncation height. If the tree is not complete, syncing nodes will panic at the assertion excerpted in figure 5.1:

```
let chunks_remaining = (2_usize).pow(trunk_height as u32);
assert_eq!(self.remaining_chunks_unchecked(), chunks_remaining);
```

Figure 5.1: Assertion will fail if the trunk of a snapshot is not a perfect BST (`merk/src/merk/restore.rs#167–168`)

Such a tree would be an extreme pathological case, likely reachable only by a maliciously constructed sequence of key insertions. We have supplied a test case generating such a pathological sequence, but due to restrictions on state modifications enforced at the application level, this attack would be difficult to exploit in a real system.

Exploit Scenario

A blockchain application uses `orga` to create a distributed state machine and takes advantage of the state sync functionality. An attacker creates a sequence of transactions that inserts keys into the distributed state tree in a layered breadth-first pattern, systematically skewing the Merk tree to the left. Nodes that attempt to sync from the resulting snapshot state crash, even when syncing from honest nodes.

Recommendations

Short term, document the edge case and track it in a GitHub issue. Add unit tests exercising the edge cases of the Merk tree data structure, including highly skewed AVL trees and trees reconstructed from Merk proofs that do not respect the AVL or BST invariants.

Long term, consider modifying the `LayerIter` implementation to skip empty sub-trees, or include a `Hash(NULL_HASH)` node in the missing leaves of the proof trunk.

6. Stored IBC consensus states cannot be pruned

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TF-6
Target: `orga/src/ibc/client_contexts.rs`	

Description

Over time, due to insufficient pruning of consensus states, the IBC client will consume excessive memory and storage.

The `delete_consensus_state` function is used to delete a consensus state at a given height (figure 6.1).

```
fn delete_consensus_state(
    &mut self,
    consensus_state_path: ibc::core::host::types::path::ClientConsensusStatePath,
) -> Result<(), ContextError> {
    self.clients
        .get_mut(consensus_state_path.client_id.clone().into())
        .map_err(|_| ClientError::ClientSpecific {
            description: "Unable to get consensus state".to_string(),
        })?
        .ok_or(ClientError::ClientStateNotFound {
            client_id: consensus_state_path.client_id.clone(),
        })?
        .consensus_states
        .remove(consensus_state_path.revision_height.to_string().into())
        .map_err(|_| ClientError::ClientSpecific {
            description: "Unable to delete consensus state".to_string(),
        })?;

    Ok(())
}
```

Figure 6.1: The incorrect key is used to prune the `consensus_states` mapping. (`orga/src/ibc/client_contexts.rs#190-209`)

However, the key (`consensus_state_path.revision_height`) used to query the `consensus_states` mapping is incorrect. This is because the key used when adding a consensus state is a formatted string that concatenates the revision number and height (figure 6.2).

```
fn store_consensus_state(
    &mut self,
    consensus_state_path: ClientConsensusStatePath,
    consensus_state: Self::ConsensusStateRef,
) -> Result<(), ContextError> {
    let epoch_height = format!(
        "{}-{}",
        consensus_state_path.revision_number, consensus_state_path.revision_height
    );
    self.clients
        .entry(consensus_state_path.client_id.into())
        .map_err(|_| ClientError::ClientSpecific {
            description: "Failed to store consensus state".to_string(),
        })?
        .or_insert_default()
        .map_err(|_| ClientError::ClientSpecific {
            description: "Failed to store consensus state".to_string(),
        })?
        .consensus_states
        .insert(
            epoch_height.into(),
            WrappedConsensusState {
                inner: ConsensusState::try_from(consensus_state).map_err(|_| {
                    ClientError::ClientSpecific {
                        description: "Failed to store consensus state".to_string(),
                    }
                })?,
            },
        )
        .map_err(|_| ClientError::ClientSpecific {
            description: "Failed to store consensus state".to_string(),
        })?;

    Ok(())
}
```

Figure 6.2: The correct key is the concatenation of the revision number and height (orga/src/ibc/client_contexts.rs#154–188)

Thus, the `delete_consensus_state` function will return `Ok()` and never prune old consensus states.

Exploit Scenario

As the light client attempts to update the state, it is unable to prune the oldest consensus state stored. Over time, this leads to excessive memory and storage consumption and causes the client to crash.

Recommendations

Short term, format the revision height and revision number in `delete_consensus_state` as shown in figure 6.3 and use that as the key into `consensus_states`:

```
    let epoch_height = format!(
        "{}-{}",
        consensus_state_path.revision_number, consensus_state_path.revision_height
    );
```

Figure 6.3: The formatted string to be used to index into the `consensus_states` mapping

Long term, improve unit testing to ensure that create and delete operations are symmetric.

7. Merk proofs can be forged to claim arbitrary key/value inclusions

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-TF-7
Target: `merk/src/proofs`	

Description

A malicious actor can construct a fraudulent Merk proof that demonstrates the inclusion of any arbitrary key-value pair in a target tree root. This could allow attackers to steal funds from bridges and other light clients by fraudulently claiming the existence of transactions that are not included in the genuine consensus state.

The malicious proofs abuse the fact that in a maliciously constructed proof, `KV` nodes may be included as children of `Hash` nodes, which in honest proofs represent omitted subtrees and do not have children. Because the children of `Hash` nodes are not included in the root hash calculation of the proof tree, the addition of these nodes does not modify the overall root hash of the proof tree. However, these nodes are included in the map used for future key/value lookup queries.

```
/// Gets or computes the hash for this tree node.
pub fn hash(&self) -> Result<Hash> {
    fn compute_hash(tree: &Tree, kv_hash: Hash) -> Hash {
        node_hash::<Hasher>(&kv_hash, &tree.child_hash(true),
&tree.child_hash(false))
    }

    match &self.node {
        Node::Hash(hash) => Ok(*hash),
        Node::KVHash(kv_hash) => Ok(compute_hash(self, *kv_hash)),
        Node::KV(key, value) => kv_hash::<Hasher>(key.as_slice(), value.as_slice())
            .map(|kv_hash| compute_hash(self, kv_hash))
            .map_err(Into::into),
    }
}
```

Figure 8.1: Merkle root calculation does not recurse under Hash nodes (merk/src/proofs/tree.rs#51–64)

This issue is high severity because it violates the fundamental guarantees of a Merkle tree implementation. However, in the `orga` library, Merk proofs are not currently used in any security-critical functionality. While this issue is high-severity for `merk` as a standalone library, it is not exploitable in Orga or Nomic.

Exploit Scenario

A blockchain uses `merk` as its backing state storage library. The blockchain developers implement a light client that verifies block headers but does not store the full state of the system. In order to process payments, an exchange maintains a light client but fetches transaction data from full nodes as needed. A malicious full node claims to submit a deposit to the exchange. When the node is queried, it provides a fraudulent Merkle proof claiming that a corresponding transaction has been included in the blockchain. The malicious actor is then credited with coins that they have not deposited and can steal funds from the exchange.

Recommendations

Short term, serialize all commitments using unique prefixes to indicate what type of deposit the commitment is for.

Long term, document the formats of all commitments and message hashing schemes for signatures. Ensure that commitments and message encodings are injective by providing a decoding routine that can be used in unit testing.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.

Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. System Invariants

Below is a non-exhaustive list of invariants that were identified during the course of the audit. Note that these invariants can be written programmatically such that they can be tested using a variety of testing methodologies.

Checkpointing Invariants

- There can be only one checkpoint in the `Building` and/or `Signing` state.
- There can be up to `max_unconfirmed_checkpoints` checkpoints in the `Completed` state.
- The fee pool must always have enough funds to pay for all checkpointing fees (should never underflow).
- All pending deposits, withdrawals, and transfers (regardless of destination) must be added to the `Building` checkpoint.
- The number of inputs in a checkpoint must be less than or equal to the `config.max_inputs`.
- The number of outputs in a checkpoint must be less than or equal to the `config.max_outputs`.
- The first output of a checkpoint transaction is the current reserve balance.
- The second output of a checkpoint transaction is the state commitment.
- The emergency disbursal outputs must account for all funds within the reserve output (excluding fees and escrowed funds on IBC chains).
- No additional inputs/outputs must be added to a checkpoint that is in a `Signing` state.
- Only checkpoints that are in a `Signing` state can be signed by signatories.
- If a checkpoint is in a `Signing` state, there must also be a checkpoint in a `Building` state.

D. Non-Security-Related Findings

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

- **Update the docstring for the `Tree::attach` function.** The docstring claims that the function will panic when attaching a node to an occupied child slot when in fact it returns an error (figure D.1).

```
/// Attaches the child to the `Tree`'s given side. Panics if there is
/// already a child attached to this side.
pub(crate) fn attach(&mut self, left: bool, child: Tree) -> Result<()> {
    if self.child(left).is_some() {
        return Err(Error::Attach(
            "Tried to attach to left child, but it is already Some".into(),
        ));
    }

    self.height = self.height.max(child.height + 1);

    let hash = child.hash()?;
    let tree = Box::new(child);
    *self.child_mut(left) = Some(Child { tree, hash });

    Ok(())
}
```

Figure D.1: merk/src/proofs/tree.rs#119–135

- **Update the `merk` documentation.** The documentation claims to use Blake2b for hashing (figure D.2) when in fact the system uses SHA512/256 (figure D.3).

```
In our implementation, the hash function used is Blake2b (chosen for its performance
characteristics) but this choice is trivially swappable.
```

Figure D.2: merk/docs/algorithms.md#32

```
pub type Hasher = Sha512_256;
```

Figure D.3: merk/src/tree/hash.rs#5

- **Update the docstring for the kv_hash function.** The docstring claims that the function fails when the key is longer than 255 bytes; however, it uses a u32 internally and does not fail on keys up to `u32::MAX` bytes (figure D.4). Because the encoded proof format does not allow for key lengths longer than `u8::MAX`, this never comes up in practice.

```rust
/// Hashes a key/value pair.
///
/// **NOTE:** This will fail if the key is longer than 255 bytes, or the value
/// is longer than 65,535 bytes.
pub fn kv_hash<D: Digest>(key: &[u8], value: &[u8]) -> Result<Hash, TryFromIntError>
{
    let mut hasher = D::new();
    hasher.update([0]);

    u32::try_from(key.len())
        .and_then(|key| u32::try_from(value.len()).map(|value| (key, value)))
        .map(|(key_length, val_length)| {
            hasher.update(key_length.to_le_bytes());
            hasher.update(key);

            hasher.update(val_length.to_le_bytes());
            hasher.update(value);

            let res = hasher.finalize();
            let mut hash: Hash = Default::default();
            hash.copy_from_slice(&res[..]);
            hash
        })
}
```

Figure D.4: merk/src/tree/hash.rs#16-38

- **The `orga` example app does not build** due to missing imports and a missing `abci` feature requirement in `Cargo.toml`.

E. Automated Testing

Semgrep

We performed static analysis on the Turbofish source code using Semgrep to identify common weaknesses. We used several rulesets, some of which are private. To run a publicly available subset of the rules, execute the commands shown in figure D.1. Note that these rulesets will output repeated results, which should be ignored.

```
semgrep --metrics=off --sarif --config="p/rust"
semgrep --metrics=off --sarif --config="p/trailofbits"
semgrep scan --pro --sarif
```

Figure E.1: Commands used to run Semgrep

Clippy

We use `cargo clippy` with "pedantic" lints to check for areas of interest in our investigation. To replicate this set of lints, you can use the command shown in figure E.2.

```
cargo clippy --no-deps --message-format=json -- -W clippy::pedantic | clippy-sarif
> clippy_pedantic.sarif
```

Figure E.2: Commands used to run Clippy

Cargo Audit

We use `cargo audit` to check for known-vulnerable dependencies.

```
cargo install cargo-audit
cargo audit
```

Figure E.3: Commands used to install and run `cargo-audit`

F. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not a comprehensive analysis of the system.

On October 18, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Turbofish team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the seven issues described in this report, Turbofish has resolved six issues and has chosen not to resolve the remaining one issue at this time. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Slashing of re-delegated stake is computed incorrectly	Resolved
2	Malicious state sync peer can cause syncing nodes to crash	Resolved
3	Interrupted snapshots can lead to inconsistent state	Resolved
4	Malicious state sync peer can cause a stack overflow in Merk	Resolved
5	Merk trunk splitting can lead to panics on degenerate trees	Unresolved
6	Stored IBC consensus states cannot be pruned	Resolved
7	Merk proofs can be forged to claim arbitrary key/value inclusions	Resolved

Detailed Fix Review Results

TOB-TF-1: Slashing of re-delegated stake is computed incorrectly
Resolved in commit 0b58db0. The re-delegated stake is now slashed at the same rate as other types of stake. The client modified the unit testing configuration to use ⅓ rather than ½ so that similar bugs can be detected in the future.

TOB-TF-2: Malicious state sync peer can cause syncing nodes to crash
Resolved in commit 921b7eb. The implementation now returns an error rather than panicking when the calculated number of chunks does not match the declared number.

TOB-TF-3: Interrupted snapshots can lead to inconsistent state
Resolved in commit 61d0376. The implementation now sets the restorer state to `None` when accepting a fresh snapshot offer.

TOB-TF-4: Malicious state sync peer can cause a stack overflow in Merk
Resolved in commit 097bc8f1. The `verify_height_proof` function is now iterative. The height of the tree is required to be less than `64`, which serves as a bound on the recursion depth for the `verify_trunk_completeness` function.

TOB-TF-5: Merk trunk splitting can lead to panics on degenerate trees
Unresolved. This issue is unlikely to be exploitable in current systems, and the client indicated that they would like to take time to consider fix options, as the suggested fixes would require substantial modifications to the codebase.

TOB-TF-6: Stored IBC consensus states cannot be pruned
Resolved in commit 462ea84. The key used to delete the consensus state has been corrected to match the key used for insertion.

TOB-TF-7: Merk proofs can be forged to claim arbitrary key/value inclusions
Resolved in commit 6515228. The Merkle tree verifier no longer allows attaching nodes under a `Hash` node.

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.