Security Audit Report

Zenrock Q1 2026 - Hush privacy protocol

Authors:
Carlos Rodriguez,
Karolos Antoniadis,
Ranadeep Biswas,
Simon Noetzlin,
Luca Joss,
Aleksandar Stojanovic,
Vukašin Dokmanović

Last Revised:
03.02.2026

Contents

Audit Overview

The Project

In January 2026, Zenrock engaged Informal Systems to perform a security audit of the Hush Privacy protocol.

Scope of this report

The audit evaluated the correctness and security properties of the `hush-wasm` library, the `x/hush` module, Miden ZK circuits, the `miden-merkle` contract, and the `miden-verifier` contract within the Hush Privacy protocol.

Audit plan

The audit was conducted between January 14th, 2026 and January 23rd, 2026, by the following personnel:

- Carlos Rodriguez
- Karolos Antoniadis
- Ranadeep Biswas
- Simon Noetzlin
- Luca Joss
- Aleksandar Stojanović
- Vukašin Dokmanović

Conclusions

The Zenrock development team has built an ambitious and technically sophisticated privacy protocol that leverages Miden STARKs to provide shielded transactions for wrapped assets on Solana. The codebase demonstrates strong engineering practices with comprehensive documentation, thoughtful architecture decisions and proactive security considerations. The team's responsiveness to identified issues and willingness to address design concerns speaks highly of their commitment to security. However, our audit uncovered several critical vulnerabilities that required immediate attention before mainnet deployment.

We identified a design flaw where the circuit fails to cryptographically bind `note_secret` to `spending_key`, allowing anyone who knows a commitment's pre-image to generate unlimited valid proofs using different spending keys—each producing a unique nullifier that bypasses double-spend protection. Additionally, the protocol suffers from other critical issues. Some of them are: asset type is not included in the proof's cryptographic binding (enabling cross-asset theft where attackers can withdraw one asset using a proof for another), values throughout the system are not validated to be less than the Goldilocks field modulus before field arithmetic operations (enabling wraparound attacks), and neither asset type nor sender address are bound to proofs (enabling mempool replay attacks where valid transactions can be intercepted and replayed with modified fields). These vulnerabilities stem from missing cryptographic bindings between proof components and inadequate input validation at system boundaries.

Additionally, other findings around input validation, duplicate detection, commitment ordering inconsistencies, and mempool replay attacks highlight gaps in defense-in-depth. A critical overarching issue is that circuit inputs on both the operand stack and advice tape are frequently not sanitized before being treated as field elements—all numeric values (amounts, fees, assets, sequences, leaf indices) should be explicitly reduced modulo p and validated against reasonable protocol-specific bounds (e.g., maximum amount per note of 2^62 to prevent accumulation overflow). Beyond these specific vulnerabilities, we recommend to deploy zrchain to a dedicated security testnet with adversarial testing scenarios (cross-asset proofs, nullifier replays, field overflow edge cases, mempool front-running) to validate that on-chain validation and ZK circuit constraints properly mitigate these attacks. This complements static code review by verifying runtime behavior under real attack conditions and ensures the circuit-client-chain integration behaves correctly under Byzantine inputs.

Once the initial audit ended, the development team addressed all findings shortly after and we reviewed the fixes and updated the status of the findings.

System Overview

Hush privacy protocol

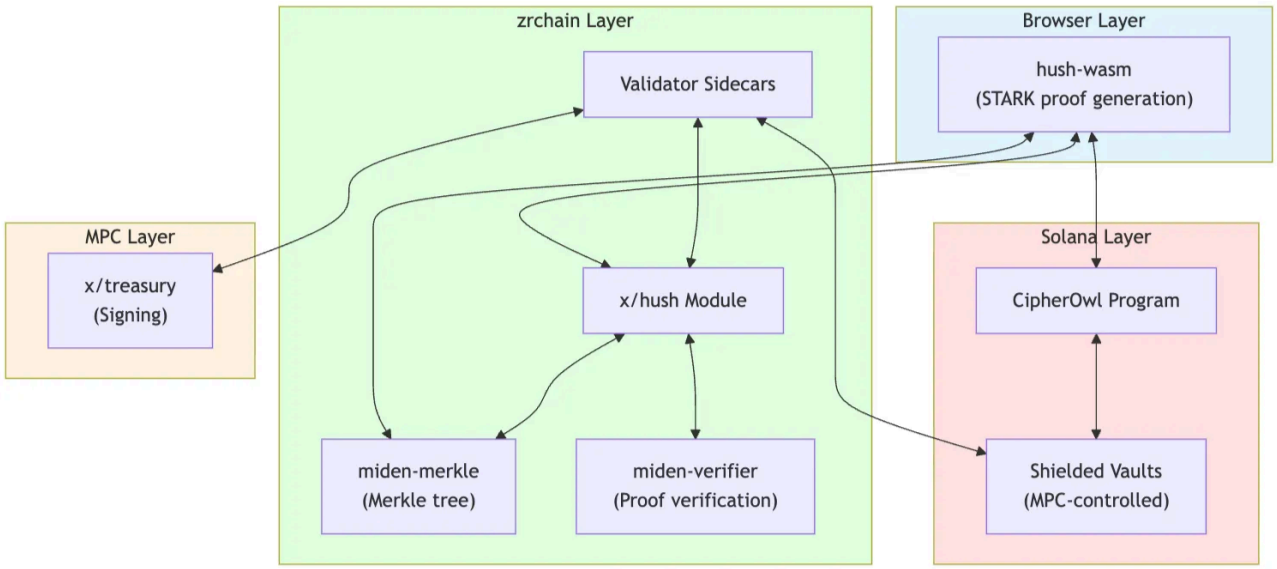
The Hush privacy protocol provides privacy for wrapped assets (zenBTC, jitoSOL) through a two-chain architecture combining Solana's settlement layer with zrchain's privacy layer. Users shield tokens on Solana by transferring them to a vault, receiving cryptographic commitments on zrchain that obscure amounts and ownership. They can then perform untraceable shielded transfers to other users or unshield to a new Solana address with no on-chain link to the original shield transaction. The protocol operates across three layers:

- Solana (token custody via vaults),
- zrchain (shielded state management via `x/hush` module, `miden-merkle` contract for Merkle tree operations, and `miden-verifier` contract for STARK proof verification),
- and browser (`hush-wasm` library for client-side proof generation).

Validator sidecars coordinate between chains by monitoring Solana events and submitting transactions to both the module and MPC keyrings for signature generation. The privacy guarantees are enforced through ZK circuits written in Miden assembly (`hush.masm`) that define the proving logic for unshield and shielded transfer operations.

The protocol achieves transaction unlinkability through zero-knowledge proofs that allow users to prove they own shielded tokens without revealing which specific tokens they're spending. Each shielded balance is represented by a cryptographic commitment that hides both the amount and the owner, and when spending, users reveal a nullifier that prevents double-spending but cannot be linked back to the original commitment. The system supports different levels of access through tiered viewing keys: spending keys provide full control, full viewing keys allow auditing of all transactions, and incoming viewing keys only permit viewing received amounts. All commitments are stored in a Merkle tree that maintains historical snapshots, allowing users to generate proofs against any past state of the system.

System architecture



Components

`hush-wasm` library

The `hush-wasm` library serves as the client-side cryptographic engine for the privacy protocol, providing browser-based implementations of all cryptographic operations required for shielded transactions. It acts as the bridge between user wallets and the on-chain privacy system, compiling Rust cryptographic code to WebAssembly for execution in web browsers.

The library computes commitments from secret values that represent shielded notes in the Merkle tree implemented in `miden-merkle` CW contract and derives nullifiers that mark notes as spent. For shielded transfers, it handles ephemeral key generation and performs key exchange to establish shared secrets, then encrypts transfer amounts so only the intended recipient can decrypt them. The library also derives the hierarchical key structure from the wallet signature, producing nullifier keys for spending operations and viewing keys that enable audit or receive-only capabilities without exposure to spending authority.

When configured with the prover feature, the library generates full STARK proofs directly in the browser. It assembles the complete execution program from the Miden assembly circuit, constructs the advice stack with all private inputs, including note secrets and Merkle authentication paths, builds the public stack inputs with the outputs commitment hash and Merkle root, and executes the Miden VM to generate a proof that can be verified on-chain.

The library also implements deterministic wallet recovery that allows users to reconstruct their entire transaction history from a wallet signature. By signing standardized messages with sequential indices, users derive the same note secrets they originally generated for each voucher, then query the blockchain for vouchers with matching commitments to identify their own. Voucher amounts are encrypted on-chain using keys derived from the spending key (for own balance notes) or ECDH shared secrets (for incoming transfers), allowing the library to decrypt and recover complete voucher data even if browser storage is cleared.

`x/hush` module

The `x/hush` Cosmos SDK module serves as the privacy layer within the zrchain blockchain, orchestrating confidential transactions using zero-knowledge proofs. It coordinates between user operations, blockchain state, and cross-chain token movements, allowing users to deposit tokens from Solana into a shared privacy pool, transfer funds within the pool with hidden amounts, and withdraw to any address without observers being able to link deposits to withdrawals. By design, the module operates on a model where each shielded voucher can be spent exactly once, with nullifiers serving as spent markers to prevent double-spending while maintaining unlinkability between commitments and their spent state.

The module handles three core operations:

- Shield deposits occur when validators detect incoming transfers on Solana, screen the sender for compliance, and if approved, add the commitment to the privacy pool and issue a new voucher to the depositor.
- Unshielding allows users to withdraw tokens by submitting a cryptographic proof of ownership.
- Shielded transfers move value between users within the privacy pool by accepting ownership proofs, creating new encrypted vouchers for recipients and change, and collecting fees.

The module maintains several key data structures to enable privacy operations. It tracks which vouchers have been spent through permanent nullifier records, manages the lifecycle of withdrawal requests from submission through signing and broadcasting to completion with automatic retry logic for failed attempts, and coordinates with the `miden-merkle` contract to maintain a rolling window of valid historical roots that users can prove against. For each operation, the module independently reconstructs a cryptographic summary of all public transaction data and combines it with the user's proof inputs to ensure verification matches what the proof actually commits to, preventing any tampering with transaction details after proof generation.

Miden ZK circuits

The `hush.masm` circuit functions as the cryptographic core of the privacy protocol, implementing a zero-knowledge STARK program that proves the validity of unshielded and shielded transfers. The circuit receives private inputs through Miden's advice stack mechanism, including the user's spending key, existing balance note data (secret values, randomness, amount), and up to 24 incoming notes with their authentication paths. It derives the nullifier key from the spending key, recomputes commitments for all input notes to verify they match the claimed values, generates nullifiers to mark these notes as spent, and verifies each note's membership in the Merkle tree by checking authentication paths against the public root. The circuit then accumulates all input amounts from both the existing balance and incoming notes, ensuring accounting across note claims.

`miden-merkle` contract

The `miden-merkle` CosmWasm contract maintains a depth-configurable sparse Merkle tree to store commitments. When a commitment is added through the `SudoMsg::AddCommitment` interface (only callable by the chain), the contract computes its position in the tree based on a sequential leaf index, updates all parent hashes along the Merkle path using the RPO256 hash function, and stores the new root. To support proof verification against historical states, the contract maintains a rolling window of recent roots via the `ROOT_HISTORY` map, with a configurable `HISTORY_SIZE` that defaults to 1000 blocks.

`miden-verifier` contract

The `miden-verifier` CosmWasm contract serves as the zero-knowledge proof verification endpoint. It acts as a thin wrapper around the Miden VM's native STARK verifier, translating data structures into the formats required by Miden's proof system. By design, the verifier is exclusively callable through the `SudoMsg` interface, ensuring that only the `x/hush` module keeper can submit proofs for verification.

The contract's primary responsibility is to verify Miden STARK proofs against a specified program hash and public inputs. When the `x/hush` Keeper submits an unshielded or shielded transfer transaction, it calls `SudoMsg::Verify` with four components: the program hash (identifying which circuit was executed), the stack inputs (public values available to the verifier), the stack outputs (expected final stack state), and the base64-encoded proof bytes. The contract deserializes the program hash into a `RpoDigest`, constructs a `ProgramInfo` with the default Miden kernel, converts stack inputs and outputs into the appropriate `Felt` representations, and invokes Miden VM's `verify()` function.

Audit Dashboard

Target Summary

- **Type:** Protocol and implementation
- **Platform:** Cosmos SDK, Go, Rust, CosmWasm, MASM
- **Artifacts:** At commit hash `6f555b9`:
  - `hush-wasm` library
  - `x/hush` module
  - Miden ZK circuits
  - `miden-merkle` and `miden-verifier` contracts

Engagement Summary

- **Dates**: January 14th, 2026 ⟶ January 23rd, 2026
- **Method**: Threat modeling, manual code review, testing

Severity Summary

Finding Severity	Number
Critical	6
High	5
Medium	11
Low	3
Informational	8
Total	**33**

Table 1: Identified Security Findings

Threat Model

Assumptions:

- For zrchain:
  1. The blockchain state is persistent and not subject to corruption or data loss.
  2. Consensus mechanism properly replicates state across validators.
  3. No chain reorganizations after finality.
  4. zrchain sidecars parse shield events that originate from actual Solana on-chain events.
  5. Shield events require ≥2/3 validator consensus and are not compromised.
  6. Shielded token assets have maximum supplies below `uint64` maximum value ($\sim 1.84 \times 10^{19}$).

Property HUSH-01: Each voucher (commitment) can only be spent (transferred or withdrawn) exactly once

- Threat a: Same commitment produces multiple distinct valid nullifiers, allowing the commitment to be spent more than once (e.g., non-deterministic nullifier computation, circuits allowing arbitrary nullifiers, etc)

  The threat holds.

  - `hush.masm`: The same commitment can produce multiple distinct valid nullifiers because the `spending_key` is an unconstrained private input, allowing any prover to generate valid proofs for the same commitment with different spending keys, resulting in different nullifiers. The circuit does to verify that the `spending_key` used to compute the nullifier is the same `spending_key` that was used to create the commitment's `note_secret`. Since `nullifier_A ≠ nullifier_B`, both can be marked as spent separately, allowing the same commitment to be double-spent. See finding "*Note secret and randomness not cryptographically bound to spending key*".

- Threat b: Nullifier spent status not persisted correctly or can be reset, allowing previously spent nullifiers to be reused

  The threat does not hold.

  - `x/hush`: Nullifier spent status is stored in the `NullifiersStore`. When a nullifier is marked as spent via `MarkNullifierSpent`, it is written to state **forever** (there is no method to remove a nullifier from `NullifierStore`). Before processing any nullifier, we call `IsNullifierSpent` check in `Unshield` for the balance nullifier and the incoming nullifiers and hence verify that each nullifier has not been spent. We have the same checks in `ShieldedTransfer` for balance and incoming nullifiers. Then, in `Unshield` and in `ShieldedTransfer` the nullifiers are marked here and here, respectively. Because a transaction is executed atomically, all the unused nullifiers are marked and they cannot be used ever again; note that `Unshield` or `ShieldedTransfer` cannot return successfully at any point before the nullifiers are marked.

- Threat c: Nullifier spent status not checked before processing, allowing the same proof to be submitted and accepted multiple times

  The threat holds.

- `x/hush`: `Unshield` and `ShieldedTransfer` check if there are duplicates in the incoming nullifiers. However, there is no check to prevent the balance nullifier (i.e., `msg.Nullifier`) from being used as an incoming nullifier. As a result, a user could submit the balance nullifier as an incoming nullifier and double spend the amount of tokens he has.

Property HUSH-02: If a user deposits amount Q of asset type A, then a spendable commitment is created that cryptographically binds the correct amount Q and asset type A, and only the depositor (holding the corresponding spending key) can spend it

- Threat a: User deposits tokens of asset type A, but receives commitment for asset type B

  The threat does not hold under the assumptions 4 and 5. The `processShieldEvents` ABCI handler processes shield events from oracle data, which originates from Solana on-chain events. The `shieldEvent.Asset` is determined by which Solana `EventStore` program emitted the event, ensuring the voucher's asset type matches what was actually deposited. The commitment stored on-chain correctly encodes asset type A. Note that a separate vulnerability exists where a user can unshield their correctly-created commitment as a different asset type. We have reported this issue in finding *"Cross-asset theft vulnerability"*.

- Threat b: User deposits Q tokens, but the total quantity of commitment created is Q', where Q' ≠ Q

  Similar to Threat a, the `shieldEvent.Amount` is sourced from Solana on-chain events via the oracle. The amount is correctly propagated from the `EventStore` to the voucher creation, ensuring the commitment binds the deposited quantity Q.

- Threat c: User deposits Q tokens, but created commitment not spendable (cannot generate valid proofs or nullifiers)

  The threat does not hold. The commitment is added to the Merkle tree atomically via `AddCommitment`. Spendability depends on the client possessing the correct spending key to generate valid nullifiers and ZK proofs.

- Threat d: User deposits Q tokens, but created commitment controlled by a non-intended user (wrong `spending_key` derivation).

  The threat holds. When a user shields tokens, they generate a balance note by deriving `note_secret` and `randomness` from their `spending_key` and sequence number (code ref). The commitment is then computed as `hash(hash(note_secret, randomness), [amount, seq, 0, asset])` (code ref). This commitment is cryptographically bound to the specific `spending_key` used during generation. However, if the pre-image of the commitment is compromised, an attacker can spend the voucher using their own `spending_key` because the circuit verifies that the `spending_key` used to compute the nullifier is the same `spending_key` that was used to create the commitment's `note_secret` and `randomness`. This is the same issue as the one reported in finding *"Note secret and randomness not cryptographically bound to spending key"*.

Property HUSH-03: If a user submits a valid unshield proof for amount Q of asset type A with fee F, then after state transitions are confirmed, the user receives Q tokens of asset type A, the pool balance decreases by Q+F, and the fee F is collected

- Threat a: User unshields commitment of asset type A, but receives tokens of asset type B

  The threat holds.

  - `hush.masm`: While asset type is cryptographically bound within individual commitments, it is not included in the proof's public outputs (`outputs_commitment`), allowing an attacker to spend commitments of one asset type while claiming to spend a different asset type, effectively converting between assets without authorization. The circuit loads asset from advice (code ref) and correctly includes it in all commitment computations, ensuring `commitment = hash(hash(note_secret, randomness), [amount, seq, 0, asset])`. However, the `asset` value is never included in the `outputs_commitment` that binds the proof's public outputs. The circuit uses one asset value to compute commitments and verify Merkle proofs, while the chain uses `msg.Asset` from the transaction message to create new vouchers and determine the asset type for unshielding. We have reported this issue in finding *"Cross-asset theft vulnerability"*.

- Threat b: User unshields commitment for amount Q, but receives amount Q' on Solana where Q' ≠ Q

  The threat does not hold.

  - `hush.masm`: The circuit correctly includes the recipient amount in the `outputs_commitment` for unshields (as opposed to transfers where the amount is kept private by using 0—code ref).

  - `hush-wasm`: The library correctly passes the `recipient_amount` to the proof generation and includes it in the outputs commitment computation.

  - `x/hush`: The module recomputes the outputs commitment using `msg.Amount` from the message and verifies the ZK proof against this commitment. If a user attempts to generate a proof with amount Q but submits a message claiming amount Q', the proof verification will fail because the computed outputs commitments will not match. Only after successful proof verification does the chain create an `UnshieldRequest` with the validated amount, which is then used to construct the Solana transfer instruction.

- Threat c: Fee calculation contains arithmetic errors (overflow/underflow/rounding)

  The threat does not hold for rounding/overflow/underflow during arithmetic operations (code ref), however, there is an edge case that can prevent legitimate users to transact (although the likelihood is low because the user's balance would need to be very large).

  - `hush.masm`: The circuit performs all arithmetic in the Goldilocks field (modulo `p = 2^64 - 2^32 + 1 = 18446744069414584321`), where operations automatically reduce results exceeding the modulus. However, the user-controlled amounts (`new_balance_amount`, `recipient_amount`) are pushed directly to the advice tape as raw `u64` integers without field validation. When constructing the advice inputs, these values are converted to `Felt` using `Felt::new()` (code ref), which does not check if the value exceeds the Goldilocks modulus `p = 2^64 - 2^32 + 1`.

- `x/hush`: The chain-side validation uses native Go `uint64` arithmetic without field reduction, only checking for `uint64` overflow (values exceeding `2^64 - 1`). This creates a gap: amounts between the Goldilocks modulus and `2^64 - 1` will pass the chain's overflow check. When individual values (`msg.Amount` or `msg.Fee`) are `>= GOLDILOCKS_MODULUS` these pass the chain's overflow check but create invalid field elements in the CosmWasm contract.

- `miden-merkle`: The `x/hush` module queries `miden-merkle` contract on endpoint `query_rpo_hash_circuit` passing the public outputs. The contract uses `Felt::new()` which does not reduce values >= modulus (ref) resulting in invalid `Felt` elements with values outside the field.

When both sides operate on invalid field elements (values >= `18446744069414584321`), the resulting behavior may produce incorrect results or undefined behavior. We have documented this issue in finding "*Unsanitized `u64` as field element*".

- Threat d: Fee deduction skipped or bypassable, allowing unshield to proceed without required fees

The threat does not hold.

- `hush.masm`: The circuit enforces conservation of value where the fee must be included in the balance equation: `total_input = new_balance + recipient_amount + fee`. The fee is cryptographically bound in the `outputs_commitment`, preventing users from generating a proof with one fee value but claiming a different fee in the message

- `x/hush`: The module enforces for unshields that `supply.TotalShielded` is decremented by `totalDeduction = amount + fee` (code ref), ensuring fees are properly accounted for in supply tracking.

Property HUSH-04: If a user submits a valid shielded transfer proof sending amount Q of asset type A with fee F to recipient R, then after state transitions are confirmed, the sender's commitment is nullified, the recipient receives a spendable commitment for Q tokens of asset type A, and the fee F is collected

- Threat a: Shielded transfer creates output commitment with an asset type different from the input

The threat does not hold.

- `hush.masm`: The circuit loads a single `asset` value (code ref) and uses this same asset for all three commitments: the balance note being spent (code ref), the new balance note (code ref), and the recipient note (code ref). The input commitment's asset type is cryptographically enforced through Merkle path verification, and all output commitments are guaranteed to contain the same asset. Therefore, the actual commitment hashes never have mismatched asset types.

  However, a related vulnerability exists: mismatched voucher metadata creates permanently unspendable funds**.** While the commitments themselves are internally consistent, the circuit does not bind the `asset` to the `outputs_commitment` (code ref). A user can generate a valid proof with `asset_A`, but submit `MsgShieldedTransfer` with `asset_B`. The proof verifies successfully, and the chain creates vouchers tagged with `asset_B` metadata. These vouchers store commitment hashes that contain `asset_A` (crypto-

graphically), but the `Asset` field claims `asset_B`. When attempting to spend later, the wallet uses `asset_B` from metadata to compute the commitment preimage, producing a different hash that doesn't exist in the Merkle tree. The funds become permanently unspendable. This issue's root cause is the same as the one reported for finding *"Mempool proof replay attack"*.

- Threat b: Sender transfers amount Q, but recipient receives commitment with total amount Q' where Q' ≠ Q

  The threat does not hold.

  - `hush.masm`: The circuit enforces that the `recipient_amount` value is loaded exactly once from the advice tape (code ref) and stored in memory. This single value is then used both in the recipient commitment computation (code ref) and in the conservation of value equation (code ref). The commitment cryptographically binds the amount as `hash(hash(note_secret, randomness), [amount, 0, 0, asset])`, and the conservation check enforces `total_input = new_balance_amount + recipient_amount + fee`.

- Threat c: Sender transfers amount Q, but created commitment not spendable by the recipient (cannot generate valid proofs or nullifiers)

  The threat does not hold.

  - `hush.masm`: The design requires the sender to deterministically derive `note_secret` and `randomness` from ECDH shared secrets with the recipient's pubkeys (code ref). However, please note that the circuit does not enforce this derivation. The circuit accepts arbitrary `note_secret` and `randomness` values from the advice tape and uses them to compute the commitment (code ref). If the sender uses incorrect `note_secret` or `randomness` values (whether through malice, software bugs, or incorrect recipient pubkeys), the resulting commitment gets inserted into the Merkle tree and marked as the recipient's voucher. However, the recipient cannot decrypt the encrypted amount to learn the commitment preimage. The funds become permanently locked.

- Threat d: Sender transfers to recipient R, but created commitment controlled by a non-intended user

  The threat does not hold.

  - `hush.masm`: The design requires the sender to deterministically derive `note_secret` and `randomness` from ECDH shared secrets with the recipient's pubkeys (code ref). Unless the sender uses the incorrect `recipient_viewing_pubkey` (whether through software bugs, or human error) the created commitment would be spendable by the intended recipient.

- Threat e: Fee calculation contains arithmetic errors (overflow/underflow/rounding)

  The threat does not hold for rounding/overflow/underflow during arithmetic operations, however, there is an edge case that can prevent legitimate users from transacting (although the likelihood is low because the user's balance would need to be very large).

  Similarly as for Property HUSH-03, Threat c, while the transfer fee is fixed (code ref), user-controlled amounts (`new_balance_amount`, `recipient_amount`) are pushed directly to the advice tape as raw `u64` integers without field validation. We have documented this issue in finding *"Unsanitized `u64` as field element"*.

- Threat f: Fee deduction skipped or bypassable, allowing transfer to proceed without required fees

  The threat holds.

  - `x/hush`: If a user does not provide a `FeeCommitment` then `hasFeeCommitment` is not set and as a result the fee-voucher-related code is not even called. Even worse, even if a `FeeCommitment` is provided, there is no check that this commitment can be spent by the fee collector in any way, so the created fee voucher remains unspendable or the user can send it back to himself.

Property HUSH-05: Commitments are created only with deposits or as outputs from valid transfers, and nullifiers are added only with withdrawals or as outputs from valid transfers

- Threat a: Transfer creates output commitment but fails to nullify input commitment

  The threat does not hold. In `ShieldedTransfer`, nullifiers are marked spent (code ref), then output vouchers are created (code ref). All operations execute within the same Cosmos SDK transaction context, ensuring atomicity.

- Threat b: Input commitment is nullified but output commitment is not created

  The threat does not hold. In both `Unshield` and `ShieldTransfer` handler, input and output commitments are respectively nullified and created within the same atomic transaction. If the voucher creation fails in `Unshield` (code ref) an error is returned, causing the entire transaction to revert. Similarly in `ShieldedTransfer`, any failure in `CreateChangeVoucherPrivate` calls (code ref) reverts all state changes, including nullifier marks.

- Threat c: Commitments created without corresponding shield event or valid transfer proof (including cross-asset replay attacks where proof is valid but no actual deposit occurred on the target asset)

  The threat does not hold under assumptions 4 and 5. Commitments can only be created through three code paths:

  1. **Shield events:** `processShieldEvents` creates vouchers only from consensus-verified oracle data (code ref). Under assumptions 4 and 5, these events represent actual Solana deposits.
  2. **Unshield change vouchers:** Created in `Unshield` handler only after ZK proof verification succeeds (code ref).
  3. **Transfer output vouchers:** Created in `ShieldedTransfer` handler only after ZK proof verification succeeds (code ref).

- Threat d: Nullifiers marked as spent without valid proof verification

  The threat does not hold. In both `Unshield` and `ShieldTransfer` handlers, nullifiers are marked spent only after the ZK proof verification succeeded.

Property HUSH-06: The global supply accounting invariant TotalShielded + PendingUnshields + TotalUnshielded + TotalFeesBurned = Total Ever Shielded always holds after any state transition

- Threat a: Integer overflow in `TotalShielded` counter

  The threat does not hold under assumptions 4, 5, and 6. The `TotalShielded` counter is incremented in `processShieldEvents` → `createVoucherInternal` (code ref) using the `shieldEvent.amount` originating from the oracle data. Note that we have reported a related recommendation to add an integer overflow check in the finding "*Missing integer overflow check in x/hush module*".

- Threat b: Integer underflow in `TotalShielded` when processing unshields with fees

  The threat does not hold. The `TotalShielded` counter is only decremented in the `Unshield` handler (code ref), which explicitly checks for overflow before `amount + fee` addition and underflow before `TotalShielded -= totalDeduction`.

- Threat c: Non-atomic updates to supply counters

  The threat does not hold. All supply updates are executed atomically within the same transaction context using Cosmos SDK's transactional state management.

- Threat d: Fee calculation overflow or rounding errors cause the calculated fee to differ from the deducted amount

  The threat does not hold under assumptions 4, 5 and 6. All fee computations use integer arithmetic only and are checked for overflow and underflow in the `Unshield` handler (code ref).

- Threat e: Fee deduction from `TotalShielded` not synchronized with `TotalFeesBurned` increment

  The threat does not hold. The `TotalShielded` decrement and `TotalFeesBurned` increment are performed atomically within the same transaction in the `Unshield` handler (code ref).

- Threat f: State migration or upgrade fails to preserve supply accounting correctly

  The threat does not hold. The migration system uses two mechanisms that preserve supply accounting:

  1. Devnet-only state clears: Migrations that clear state (v5→v6 through v12→v13) are guarded by a chain ID prefix check that only allows execution on "amber" (devnet) chains (code ref). Non-amber chains (including mainnet) skip these migrations entirely and preserve the existing supply.

  2. Genesis export/import: The `ExportGenesis` function properly exports the `Supply` struct (code ref), and `InitGenesis` properly imports it (code ref), ensuring supply accounting is preserved during chain upgrades via genesis state transfer.

  3. Versioned migration system: All migrations are registered via the Cosmos SDK's `cfg.RegisterMigration` (code ref), which ensures ordered, idempotent execution and prevents duplicate or out-of-order migrations that could corrupt state.

Property HUSH-07: If a voucher record exists with commitment C, then commitment C exists in the Merkle tree at some leaf position, and sum of all spendable voucher amounts equals (total shielded) - (total unshielded) - (total fees collected)

- Threat a: Voucher record created in state but corresponding commitment not inserted into Merkle tree

  The threat does not hold.

  - `x/hush`: The voucher record is created in `createVoucherInternal` and stored in the state through `SetVoucher`. Note that before `SetVoucher`, the corresponding commitment is added through `AddCommitment` that calls the Merkle contract. If `AddCommitment` fails to be added, `createVoucherInternal` returns immediately without creating the voucher in the state. Therefore, in normal conditions, this threat holds.

- Threat b: Commitment inserted into the Merkle tree, but no corresponding voucher record created

  The threat does not hold.

  - `x/hush`: From the `x/hush` module perspective, the same applies as in *Threat a*. There is a small difference, in case a voucher is also created during shield events that takes place in `PreBlocker`, potentially having a successful `AddCommitment` but then failing to add the voucher in the store, and hence this shield event is omitted even though a commitment was added due to it. This can only happen due to storage corruption which we assume is not the case or if the `len(commitment) != 32` but if this was the case `AddCommitment` would also fail.

- Threat c: Same commitment inserted multiple times at different leaf positions, allowing multiple spends via different Merkle paths

  The threat does not hold but can lead to issues.

  - `x/hush`: Although the same commitment can be inserted multiple times at different leaf positions, it does not allow for multiple spends.

    The same commitment can be inserted multiple times if a user initiates two shielding events with the exact same commitment that are both in `processShieldEvents`. During the first processing of the shield event, a voucher would be created for this commitment. Then, when the second shield event is being processed (this can happen because it has a different `shieldEvent.TxId` and `shieldEvent.LogIndex` than the first shield event), it will also create a voucher for the exact same commitment, and furthermore *overwrite* `CommitmentToVoucherStore` the `nextID` of this commitment. Nevertheless, double spending is **not** possible because the nullifier for this commitment can only be used once. When the user spends one voucher, the nullifier is marked as spent and the user cannot use the other voucher.

    The same issue can potentially appear in `Unshield` and `ShieldedTransfer` but would need extra effort from the user to create `NewBalanceCommitment`.

    Due to the above we can have erroneous supply stats and bad UX (see finding "*Erroneous supply stats*").

- Threat d: Commitment insertion reports success, but the internal tree state is not updated correctly

The threat does not hold.

- `miden-merkle`: The commitment insertion process in the `miden-merkle` contract is protected by CosmWasm's transaction atomicity guarantees, which ensure that tree state cannot become inconsistent with reported success.

  When the `sudo_add_commitment()` function executes, all storage operations write to a cached state rather than directly to persistent storage. If any operation in this sequence fails, the entire transaction is rolled back atomically, discarding all cached changes and returning an error to the caller.

  Furthermore, the tree update algorithm itself is mathematically correct, implementing proper parent hash computation with correct sibling selection and node position encoding that prevents collisions.

  Since the keeper receives an error if anything fails and the contract only returns success after all state has been committed atomically, there is no scenario where commitment insertion can report success with incorrectly updated internal tree state.

- Threat e: State migration, upgrade, or error recovery causes permanent desynchronization between voucher records and the tree

  At the current state the threat does not hold.

  - `x/hush`: In case of an exported genesis file, the Merkle-tree state is exported and is taken care of by `x/wasm`. For the `x/hush` module, the genesis exports and imports the *whole* state as expected. Note that `GenesisState` does not include `CommitmentToVoucherStore` but this state is re-constructed during `InitGenesis`.

Property HUSH-08: Only a user who knows the complete commitment pre-image (note secret, randomness, amount, asset) and holds the spending key from which the nullifier key is derived can spend that commitment

- Threat a: Circuit fails to cryptographically bind `nullifier_key` and `note_secret` to the same root `spending_key`

  The threat holds. The circuit derives `nullifier_key` from `spending_key` (code ref) but loads all `note_secret` (an `randomness`) values for balance note (code ref) and incoming notes (code ref) directly from the advice stack, without verifying they were derived from the same `spending_key`. Since nullifiers are computed as `hash(nullifier_key, commitment)` where commitment depends on `note_secret` (and `randomness`), an attacker can use the same commitment with different spending keys to generate different nullifiers, bypassing double-spend protection. See finding *"Note secret not cryptographically bound to spending key"* for more details.

- Threat b: Circuit allows the prover to use arbitrary `nullifier_key` not derived from the `spending_key` that created the commitment

  The threat does not hold. The circuit correctly enforces that `nullifier_key` is derived from `spending_key`. The circuit loads `spending_key` from advice (code ref),

and it executes `exec.derive_nullifier_key` (code ref) which deterministically computes `nullifier_key = hash(spending_key || NULLIFIER_KEY_DOMAIN)` (code ref). The derived `nullifier_key` is then stored in memory and used for all nullifier computations via `exec.compute_nullifier`. However, as explained in Threat a, the circuit does allow arbitrary `spending_keys` to be provided, which, combined with the fact that `note_secret` is not verified to be derived from that same `spending_key`, enables the double-spend vulnerability previously identified.

- Threat c: Circuit allows spending with knowledge of the pre-image but without proving ownership of the corresponding `spending_key`

  The threat holds. The circuit only verifies that the prover knows the commitment pre-image (`note_secret`, `randomness`, `amount`, `seq`, `asset`) and that this commitment exists in the Merkle tree. It never verifies that the provided `spending_key` has any relationship to the commitment being spent. This is precisely the double-spend vulnerability identified earlier: knowing the commitment pre-image is sufficient to spend, and there's no cryptographic proof of `spending_key` ownership over that specific commitment.

- Threat d: `nullifier_key` derivation uses weak domain separation (e.g., missing or non-unique separators, separators that allow collisions)

  The threat does not hold. The nullifier key derivation uses strong, collision-resistant domain separation. The circuit implementation uses a unique constant `NULLIFIER_KEY_DOMAIN = 7310582938571023456` (code ref).

- Threat e: Note secret derivation uses weak domain separation (e.g., missing or non-unique separators, separators that allow collisions)

  The threat does not hold. The note secret derivation uses strong, collision-resistant domain separation with unique separators for each purpose. The implementation uses two distinct derivation paths: (1) Balance notes derive `note_secret = hash(spending_key || "zenrock.hush.balance_note.v1" || seq)` (code ref), and (2) Stealth transfers derive `note_secret = hash(ECDH_shared_secret || "zenrock.hush.note_secret.v1")` (code ref).

- Threat f: Message sender validation insufficient, allowing user A to submit valid proof generated by user B and spend B's commitment

  The threat holds. The `recipient_address` (Solana destination) of `MsgUnshield` (code ref) is not cryptographically bound to the ZK proof, allowing proof interception and fund redirection attacks. For unshield operations, the circuit sets `recipient_commitment = zeros` (code ref), and the `recipient_address` string field is never included in the `outputs_commitment` (code ref). The chain only validates the recipient address for base58 format, not cryptographic binding to the proof.

  If Alice generates a valid ZK proof to unshield funds to her Solana address and Bob intercepts the proof bytes, then Bob can construct a new `MsgUnshield` transaction with: (1) his own address as `creator`, (2) Alice's ZK proof, (3) Alice's nullifiers/commitments/amounts (unchanged), (4) Bob's Solana address as `recipient_address`. The proof verification succeeds because it only validates nullifiers, commitments, and amounts (the recipient address is completely unconstrained). The unshield proceeds and funds are sent to Bob's Solana address instead of Alice's. We have reported this issue in finding "*Unshield recipient address not cryptographically bound to ZK proof*".

- Threat g: Nullifier extracted from proof outputs does not match the nullifier being marked as spent in state, allowing commitment to remain spendable

  The threat does not hold. The nullifier marked as spent in the state is cryptographically bound to the ZK proof and cannot differ from the nullifier proven in the circuit.

  The circuit computes the balance nullifier as `hash(nullifier_key, commitment)` and stores it in memory (code ref). This nullifier is then included in the `outputs_commitment` computation (code ref), which becomes a public input to the verifier. When the chain processes the transaction, it recomputes the `outputs_commitment` using `msg.Nullifier` from the message (code ref `MsgUnshield`, code ref for `MsgShieldedTransfer`) and verifies the proof against this commitment. If an attacker attempts to provide a different nullifier in `msg.Nullifier` than what the circuit computed, the chain's recomputed `outputs_commitment` will not match the one in the proof, causing verification to fail. Only after successful verification does the chain mark `msg.Nullifier` as spent, ensuring that the nullifier marked in the state is exactly the one that was cryptographically proven.

  Similarly, for incoming notes, their nullifiers are included in the public outputs (code ref), and if proof verification succeeds, the nullifiers are marked as spent.

Property HUSH-09: If a user has spending key SK, then only holders of keys derived from SK can decrypt voucher amounts: spending key holder (full access), full viewing key holder (decrypt only, no spend), incoming viewing key holder (decrypt received only), no key holder (see only encrypted data)

- Threat a: Encryption key derivation vulnerable to collision attacks, related-key attacks, or length extension attacks due to weak key derivation function construction (e.g., single SHA256 instead of HKDF, missing salt/context, insufficient iterations)

  The threat does not hold. Currently, all inputs to `rpo_hash_internal` operate on fixed-length inputs (e.g., 32-byte shared secrets) or on concatenations where any variable-length component is followed by a fixed, non-zero-terminated domain separator. As a result, the zero-padding used in the byte-to-field encoding does not introduce ambiguity or collision risk under the existing design assumptions.

- Threat b: ECDH shared secret vulnerable to small subgroup attacks or brute-force due to: (1) received public keys not validated (low-order points, curve membership), or (2) generated private keys weak (insecure RNG, insufficient entropy)

  The threat holds, see finding "*Missing shared secret validation*".

  When generating the shared secret with the public key, if it is all-zeros, then the encrypted value can then be decrypted using any private key combined with the all-zeros shared secret.

- Threat c: Authenticated encryption (ChaCha20-Poly1305) fails to verify tags, uses predictable or non-unique nonces, or leaks timing information

  The threat does not hold. Decryption errors are handled without leaking information, and there are no fast/slow paths depending on the result. Nonces and keys combinations are unique or generated randomly.

- Threat d: Key hierarchy vulnerable to privilege escalation (deriving a higher privilege key from a lower privilege key) due to reversible key derivation (e.g., XOR with constants, weak hash, algebraic relations) instead of one-way cryptographic functions

  The threat does not hold. The key derivations follow the key hierarchy described where the `wallet_signature` is the highest privilege key, the spending key is derived from the `wallet_signer` and the other keys can be derived from the `spending_key`.

- Threat e: Circuit public inputs expose private information (amounts, note secrets, randomness, or other sensitive data) that should remain encrypted

  The threat does not hold. The protocol uses only 8 field elements as public inputs: `merkle_root` (4 felts) and `outputs_commitment` (4 felts) (code ref). All sensitive data (amounts, note secrets, randomness, spending keys, asset type) are passed as private advice inputs and never exposed in public inputs. The `outputs_commitment` is an RPO hash binding public outputs (nullifiers, commitments, fee, and recipient amount) but does not reveal the underlying values. For shielded transfers, privacy is maximized by setting `recipient_amount = 0` in the `outputs_commitment` (code ref), with the actual amount encrypted in `encrypted_recipient_amount`. For unshields, the amount and asset are intentionally public since the unshield recipient address is visible on Solana anyway. No note secrets, randomness, or spending keys appear in public inputs.

Property HUSH-10: If a user performs a shielded transfer of amount Q from commitment C1 to recipient R, observers cannot determine recipient's identity, amount Q being transferred, or which commitment C1 is being spent

- Threat a: Recipient identity revealed through on-chain data (recipient address, recipient public key, non-unique ephemeral keys)

  - `x/hush`: The threat does not hold. Assuming the user performs shielded transfers carefully (i.e., `sender_randomness` is random and not re-used across transfers and the user correctly derives the ephemeral key), then this threat does not hold. In such a scenario, there is no way for an adversary to look at `EmphemeralPubKey` to infer something meaningful (e.g., transfers stem from the same sender). Additionally, the `RecipientCommitment` is just as hash, as well as the recipient, asset, etc.

- Threat b: Transfer amount revealed through on-chain data (plaintext amounts, predictable encryption)

  - `x/hush`: The threat does not hold. As with Threat a, if we make the same assumption, the transfer amount is encrypted in `EncryptedRecipientAmount`.

- Threat c: Input commitment (sender) revealed through on-chain data

  - `x/hush`: The threat does not hold as long as `MsgShieldedTransfer.Creator` is not re-used. Additionally, the sender reveals their `Nullifier` but it is unlinkable to the commitment and even if someone could correlate a nullifier to its original voucher, they would not learn who actually sent this amount.

Property HUSH-11: All protocol components correctly integrate with Miden VM and produce cryptographic results (commitments, nullifiers, hashes, Merkle paths) that are consistent with Miden VM circuit behavior

- Threat a: Hash computations (commitments, nullifiers, Merkle nodes) in off-circuit implementations produce different values than the circuit's `hmerge` / `hperm` operations for identical inputs, causing commitment/nullifier mismatches

  The threat does not hold. For both the implementation of `vm_hmerge` (`hush-wasm`) (code ref) and `hmerge_circuit` (`miden-merkle`) (code ref), when performing the hash of two words `hash(A, B)` both Miden VM's `hmerge` and the off-circuit implementation produce a stack-ordered result: `[result[3], result[2], result[1], result[0]]` (this represents `hash(A, B)` in stack order). The following table summarizes the steps of both operations for hashing `A` and `B` (with `B` on the top of the stack).

  - Step 0: Input format
    - Miden VM `hmerge`:
      - Stack: `[B[3], B[2], B[1], B[0], A[3], A[2], A[1], A[0]]`
      - Words in stack order
    - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
      - Function receives Word structs:
        - `B = [B[3], B[2], B[1], B[0]]` (stack order)
        - `A = [A[3], A[2], A[1], A[0]]` (stack order)
  - Step 1: Convert to logical order
    - Miden VM `hmerge`:
      - Automatically reverses during stack→state mapping:
        - `B_stack → B_logical = [B[0], B[1], B[2], B[3]]`
        - `A_stack → A_logical = [A[0], A[1], A[2], A[3]]`
    - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
      - Explicitly reverses each word:
        - `b_logical = [B[0], B[1], B[2], B[3]]`
        - `a_logical = [A[0], A[1], A[2], A[3]]`

        Note:

`a_on_top` (param) = B

`b_on_bottom` (param) = A

- Step 2: Build RPO state
  - Miden VM `hmerge`:
    - Constructs 12-element state: `[A_logical, B_logical, zeros]`
    - Applies RPO permutation
  - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
    - Calls `Rpo256::merge([A_logical, B_logical])`
    - Constructs same state `[A_logical, B_logical, zeros]`
- Step 3: Extract result
  - Miden VM `hmerge`:
    - Result in logical order: `[result[0], result[1], result[2], result[3]]`
  - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
    - `Rpo256::merge` returns result in logical order: `[result[0], result[1], result[2], result[3]]`
- Step 4: Convert to stack order
  - Miden VM `hmerge`:
    - Automatically reverses during state→stack mapping: `result_logical → result_stack = [result[3], result[2], result[1], result[0]]`
  - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
    - Explicitly reverses result `[result[3], result[2], result[1], result[0]]`
- Output format
  - Miden VM `hmerge`:
    - Stack: `[result[3], result[2], result[1], result[0]]`
    - Result in stack order
  - `hmerge_circuit(B, A)` / `vm_hmerge(B, A)`:
    - Returns Word: `[result[3], result[2], result[1], result[0]]`
    - Result in stack order

- Threat b: Merkle tree implementation in `miden-merkle` contract uses a different hash function, node ordering, or path format than the circuit's `verify_merkle_path`, causing valid proofs to be rejected

  The threat does not hold. The `miden-merkle` contract uses the function `hash_nodes` to hash intermediate nodes and the root. This function internally uses `hmerge_circuit`, which produces the same result as `hmerge`. The Merkle path returned by the query `query_merkle_path` (code ref) consists of the leaf value and a vector of siblings (starting from the sibling of the leaf) that are iteratively hashed to calculate the root. This is consistent with the implementation of `verify_merkle_path` (code ref). The siblings are encoded as 4 field elements in stack order (big endian).

- Threat c: Public input ordering or encoding for proof generation/verification does not match circuits' expectations

  The threat does not hold. The ordering of public inputs is the same in `hush-wasm` (code ref) and in `x/hush` (code ref).

- Threat d: Commitment computation in `hush-wasm` differs from circuit computation

  The threat does not hold. Commitment computation in functions `compute_commitment_internal` (code ref) and `compute_balance_commitment_internal` (code ref) of `hush-wasm` (code ref) produce the same result as the circuit's commitment computation (code ref). The commitment is logically computed as `hash(hash(note_secret, randomness), [asset, 0, sequence, amount])` and the implementations in `hush-wasm` performs the same steps as in the circuit:

  - Calculate `step1` by applying `hmerge` on a stack with `[randomness, note_secret]` with field elements of both words in stack order (big endian).
  - Calculate commitment by applying `hmerge` on a stack with `[amount, sequence, 0, asset, step1]`.

- Threat e: Commitment computation in `miden-merkle` differs from circuit computation

  The threat holds. Commitment computation in query `query_compute_commitment` of `miden-merkle` (code ref) does not produce the same result as the circuit's commitment computation (code ref). The commitment should be logically computed as `hash(hash(note_secret, randomness), [asset, 0, sequence, amount])` but the implementation in `miden-merkle` performs the following steps:

  - Calculate `step1` by applying `hmerge` on a stack with `[randomness, note_secret]` with field elements of both words in stack order (big endian).
  - Calculate commitment by applying `hmerge` on a stack with `[0, 0, asset, amount, step1]` instead of `[amount, sequence, 0, asset, step1]`.

  We have documented this issue in finding "*Commitment field ordering inconsistency*".

- Threat f: Nullifier computation in `hush-wasm` differs from circuit computation

  The threat does not hold. Nullifier computation in function `compute_nullifier_v2_internal` in `hush-wasm` (code ref) produces the same result as the circuit's nullifier computation (code ref). The nullifier is logically computed as `hash(nullifier_key, commitment)` and the implementation in `hush-wasm` performs the

same step as in the circuit: Applies `hmerge` on a stack with `[commitment, nullifier_key]` with field elements of both words in stack order (big endian).

- Threat g: Nullifier computation in `miden-merkle` differs from circuit computation

  The threat does not hold. Nullifier computation in query `query_compute_nullifier` in `miden-merkle` (code ref) produces the same result as the circuit's nullifier computation (code ref). The nullifier is logically computed as `hash(nullifier_key, commitment)` and the implementation in `miden-merkle` performs the same step as in the circuit: Applies `hmerge` on a stack with `[commitment, nullifier_key]` with field elements of both words in stack order (big endian).

- Threat h: Field element reduction or modular arithmetic differs between components (Goldilocks field modulus 2^64 - 2^32 + 1 not consistently applied)

  The threat holds. Not all locations in `hush-wasm`, `miden-merkle` and `miden-verifier` that create a field element from a `u64` (with `Felt::new()`) reduce the input value modulo Goldilocks by using the function `to_field_safe` (code ref). For example, the implementation of `bytes_to_word` in `miden-merkle` doesn't call `to_field_safe` (code ref) while `bytes_to_word_internal` in `hush-wasm` does (code ref). See finding "*Unsanitized `u64` as field element*".

- Threat i: Domain separation constants (NULLIFIER_KEY_DOMAIN, NOTE_SECRET_DOMAIN, etc.) not consistent across Rust and assembly

  The threat does not hold. The function `derive_nullifier_key_circuit_internal` in `hush-wasm` (code ref) uses the same domain separator constant (code ref) as the circuit (code ref). The circuit does not compute the note secret and thus does not use its domain separator.

Property HUSH-12: The Merkle tree implementation provides sound membership proofs: valid proofs are accepted for leaves in the tree, and no valid proof exists for leaves not in the tree; the tree preserves insertion order and historical roots

- Threat a: Hash function used in the Merkle tree is not collision-resistant, allowing an attacker to find two different commitments with the same hash

  The threat does not hold. The Merkle tree uses RPO256 (Rescue Prime Optimized) as its hash function, which provides strong collision resistance guarantees.

- Threat b: Merkle tree function is not second preimage resistant

  The threat holds, but it is mitigated. While the Merkle tree implementation is theoretically vulnerable to second preimage attacks—intermediate node hashes are observable and could be targeted to find alternate paths—this threat is fully mitigated by the ZK circuit constraints and does not pose a practical security risk.

- Threat c: Given a commitment in the tree, the Merkle path generation fails or produces an invalid path that does not verify against the root

The threat does not hold. The Merkle path generation implementation cannot produce invalid paths under normal operation. All commitments are validated to be exactly 32 bytes before storage (code ref), all nodes are stored via `word_to_bytes()` which always produces 32 bytes (code ref), and missing entries in sparse tree positions correctly return computed empty subtree hashes rather than errors (code ref 1, code ref 2). The only theoretical error path in `bytes_to_word` (code ref) cannot occur given these validation guarantees. The threat does not hold and would only manifest in cases of storage corruption or catastrophic database failure affecting the entire blockchain node.

- Threat d: Given a commitment not in the tree, a Merkle path can be constructed that falsely verifies against the root

  The threat does not hold. An attacker cannot provide fake proofs on fake roots because the root validation mechanism at `x/hush` requires all Merkle roots to be either the current root or present in `ROOT_HISTORY` (code ref). Roots can only be added to history through the Sudo-protected `AddCommitment` endpoint (code ref), which is exclusively callable by the `x/hush` module during legitimate shield event processing. Even if an attacker could mathematically generate a valid ZK proof against an arbitrary root value, that root would fail the `IsValidMerkleRoot` check and the transaction would be rejected.

- Threat e: Tree does not handle edge cases correctly (empty tree, single leaf, full tree, tree depth limits)

  The threat does not hold. The Merkle tree implementation correctly handles all edge cases through explicit validation and proper empty subtree computation. Empty trees are properly initialized by computing `empty_leaf() = hash(0)` and hashing it with itself depth times (code ref). Sparse tree positions use the same recursive formula to compute empty subtree hashes at any level (code ref), ensuring consistency between initialization and path generation. Full tree conditions are explicitly prevented with capacity checks that return `TreeFull` error when `next_leaf_index >= 2^depth` (code ref).

- Threat f: Insertion order is not preserved or deterministic, causing different nodes to compute different roots for the same commitment set

  The threat does not hold. The Merkle tree insertion is fully deterministic and order-preserving. Each commitment is inserted at the position `next_leaf_index` (code ref), which is a monotonically increasing counter stored in the tree state and incremented after each insertion (code ref). The insertion algorithm is deterministic: given a leaf index, it computes the parent hash using a fixed formula based on whether the index is odd (right child) or even (left child) (code ref), stores all intermediate nodes at deterministic positions (code ref), and updates the root.

- Threat g: Historical root preservation fails, causing valid proofs with recent historical roots to be incorrectly rejected

  The threat holds. The contract stores historical roots using `block.height` as the key in the `ROOT_HISTORY` map (code ref). When multiple commitments are added within the same block (e.g., multiple `MsgShieldedTransfer` or `MsgUnshield` transactions in a single block), each call to `sudo_add_commitment` updates the Merkle tree root and attempts to save it to `ROOT_HISTORY` with the same `block.height` key. This causes intermediate roots to be overwritten (only the final root from the last commitment addition in that block is preserved in history). Users who generate proofs against intermediate roots (created earlier in the same block) will have their proofs rejected during `IsValidRoot` checks, even though those roots were valid at the time the commitment was added. We have reported this issue in finding "*AddCommitment overwrites*".

- Threat h: Root history retention policy not enforced, allowing roots to be dropped before the configured period or kept beyond limits

  The threat does not hold. The contract intentionally separates the validation window (controlled by a configurable `HISTORY_SIZE`) from storage retention (unlimited). The `execute_update_history_size` admin function allows runtime tuning of the proof validity window without requiring contract migration. While this causes unbounded storage growth, it doesn't affect query costs. A recommended root history cleanup mechanism is detailed in the "*Miscellaneous code improvements*" section.

Property HUSH-13: All message fields are validated before processing

- Threat a: `MsgUnshield` message inputs are not properly validated before processing

  The threat holds. `MsgUnshield` message inputs are not properly validated before processing. A finding has been reported "*Missing message validation in Unshield handler*".

- Threat b: `MsgShieldedTransfer` message inputs are not properly validated before processing

  The threat holds. `MsgShieldedTransfer` message inputs are not properly validated before processing. A finding has been reported "*Missing inputs validation in Unshield handler*".

- Threat c: `MsgUpdateParams` message inputs not properly validated before processing

  The threat does not hold. The `UpdateParams` handler validates authority via `CheckAuthority` (code ref), then calls `SetParams` which is expected to perform internal validation. Authority validation prevents unauthorized parameter modifications.

- Threat d: `SudoMsg::AddCommitment` message inputs not properly validated before processing

  The threat holds. The `sudo_add_commitment()` function (code ref) implements basic structural validation by checking that commitments are exactly 32 bytes and that tree capacity is not exceeded, preventing length-based attacks and overflow conditions. However, the function lacks field element validation for the commitment bytes. When converting the 32-byte commitment to four field elements via the `bytes_to_word()` function (code ref), the code uses `Felt::new(u64::from_le_bytes(arr))` (code ref) which does not validate that the resulting `u64` values are less than the Goldilocks field modulus. While `Felt::new()` may internally reduce values modulo p, accepting out-of-range inputs creates a trust boundary issue where the contract accepts commitments that were not properly validated during their off-chain generation, potentially leading to inconsistencies between on-chain and off-chain hash computations if the client's commitment generation uses different validation logic. This issue has been reported in the "*Unsanitized `u64` as field element*" finding.

  Another validation gap is the absence of an explicit check rejecting commitments that equal the `empty_leaf()` value (RPO hash of [0,0,0,0]), which could theoretically cause confusion in Merkle proof verification if such a collision occurred, though the cryptographic improbability makes this purely a defense-in-depth consideration rather than a practical security concern. A recommendation for this issue has been explained in the "*Miscellaneous findings on CW contracts*" section.

- Threat e: `ExecuteMsg::UpdateHistorySize` message inputs are not properly validated before processing

The threat does not hold. The only input parameter that is provided to this function is the `new_size: u32` parameter (code ref), and the only present validation is a check that this parameter does not equal zero. This validation is sufficient. Since this function is only callable by the contract admin, the admin can increase or decrease the `HISTORY_SIZE` storage variable.

By doing so, the admin modifies how many of the last `ROOT_HISTORY` entries can be queried by the user. The `ROOT_HISTORY` storage variable stores the historical root hashes, while the `HISTORY_SIZE` determines how many of them can be queried and used to submit proofs for.

- Threat f: `ExecuteMsg::UpdateAdmin` message inputs are not properly validated before processing

  The threat holds. From the validation standpoint, the only input parameter that is provided is the `new_admin: Option<String>` parameter (code ref), which is properly validated. It follows the best practices by being of `String` type, and by being validated via the `deps.api.validate()` function. There are no other validations necessary for this input parameter.

  Even though an invalid `new_admin` parameter cannot be provided, the whole admin update process is implemented in the `execute_update_admin()` does not follow the best practices. The process is implemented in a single step, where the admin only overwrites its address by adding the new admin, which does not follow the best practices regarding the admin update process.

  The admin is assumed to be a trusted and reliable source, but operational errors can still happen. To mitigate the risk of administrative lockout and to ensure that admin updates are both valid and explicitly acknowledged, the contract should use the two-step ownership transfer pattern provided by the `cw-ownable` library.

  The library enforces a proposal → acceptance workflow before ownership changes are finalized. It ensures that no contract can become ownerless or accidentally locked due to operational errors.

  This issue has been reported in the "*Lack of confirmation during admin updates*" finding.

- Threat g: `SudoMsg::Verify` message inputs are not properly validated before processing

  The threat holds. The `hash` parameter is adequately protected by `RpoDigest::read_from_bytes()`, which enforces the exact 32-byte size requirement and rejects malformed inputs, making additional validation redundant.

  The `inputs` parameter is constructed deterministically by the keeper to always contain exactly 8 elements, and while Miden's `StackInputs::try_from_ints()` (code ref) silently pads any input size up to 16 elements with zeros, this behavior cannot be exploited maliciously. However, this silent padding mechanism can mask programming bugs in the Go layer, leading to cryptic verification failures that are difficult to debug. A recommendation for this issue has been explained in the "*Miscellaneous findings on CW contracts*" section.

  The `proof` parameter accepts arbitrarily large base64-encoded strings without size bounds, representing a fail-slow inefficiency where oversized proofs waste gas during decode and deserialization before eventually failing verification. While this creates user friction for accidental misuse (paying unnecessary gas fees for malformed proofs), it does not constitute a practical DoS vector since attackers must pay for all consumed gas, making the attack economically self-defeating.

The `outputs` parameter suffers from a confusing API design using a nested `Vec<Vec<u64>>` structure when only a single `Vec<u64>` is needed when calling the `StackOutputs::new()` function (code ref), and it only utilizes the first element while silently discarding the rest, though this too is limited to causing developer confusion rather than security issues. A recommendation for this issue has been explained in the "*Miscellaneous findings on CW contracts*" section.

Adding explicit size validation for inputs (expecting exactly 8 elements), enforcing an upper bound on proof size, and either flattening the outputs structure or validating that only one array is provided would significantly improve code robustness and debugging experience.

- Threat h: `InstantiateMsg` message inputs are not properly validated before processing

  The threat holds. The `instantiate()` function (code ref) does not validate the `tree_depth` input parameter, which can lead to the parameter being set to a value ≥64. This can then cause overflows in the downstream execution when calling `max_tree_capacity()` (code ref) and `node_position()` (code ref) functions, which prevents any commitments from being added. This issue has been reported in the "*Missing tree depth validation makes the contract unusable*" finding.

  Another problem is that the `instantiate()` function does not verify that the `history_size` is not zero. This check is explicitly implemented in the `execute_update_history_size()` function.

Property HUSH-14: Queries are properly constructed with valid parameters, execution errors are handled correctly, and responses are correctly interpreted by the caller

- Threat a: `QueryMsg::RpoHashCircuit` constructed with invalid parameters, execution errors not handled, or response misinterpreted by the caller

  The threat does not hold. `RpoHashCircuit` correctly sets its parameters and returns the response from the Merkel contract. Specifically, `RpoHashCircuit` is called from `ComputeOutputsCommitmentV7` that generates an `inputs` slice with 112 `uint64` that is passed to `RpoHashCircuit` and eventually to the contract that uses those values without returning an error.

- Threat b: `QueryMsg::GetMerklePath` constructed with invalid parameters, execution errors not handled, or response misinterpreted by the caller

  The threat does not hold. The `GetMerklePath` query gets a `LeafIndex` as a parameter. This leaf index stems from the voucher retrieved in `MerklePath` and this voucher was created here so the leaf index is correctly set and the leaf exists (so we are not in this case). The response from the query is correctly converted and used by the `MerklePath` caller.

- Threat c: `QueryMsg::IsValidRoot` constructed with invalid parameters, execution errors not handled, or response misinterpreted by the caller

  The threat does not hold under specific cases. `IsValidRoot` is called from `IsValidMerkleRootFromContract` and gets as a parameter a `Root` and returns whether it is valid or not. The provided parameter is checked that it has the right length in `Unshield` and in `ShieldedTransfer`.

❗ Note that `IsValidRoot` might miss some roots if those are overwritten due to `AddCommitment` (we describe this issue in more detail in finding "*AddCommitment overwrites*").

- Threat d: `QueryMsg::GetRoot` execution errors not handled or response misinterpreted by the caller

  The threat does not hold. `GetRoot` does not take any parameters and loads the tree state storage and extracts from that state the root, so it operates like `GetTreeState` with the addition that the state (that is instantiated) of the root is also returned. `GetMerkleRoot` that queries `GetRoot` correctly retrieves the response and returns the root.

- Threat e: `QueryMsg::GetTreeState` execution errors not handled or response misinterpreted by the caller

  The threat does not hold. `GetTreeState` does not take any parameters and loads the tree state from storage and the state is initialized so loading of the tree state succeeds. `GetMerkleTreeStateFromContract` that queries the tree state correctly retrieves the response and sets the corresponding `MerkleTreeState` fields and in case of errors, those are handled (and here).

- Threat f: `QueryMsg::GetVerifResult` execution errors not handled or response misinterpreted by the caller

  The threat does not hold. `GetVerifResult` does not take any parameters and simply returns the result stored in `deps.storage`. `GetVerifResult` is called from `VerifyZKProof` after the proof has been validated and the result has already been stored. Note that in case of a successful verification `"Execution verified!"` is returned, which is exactly the same string against which `VerifyZKProof` performs the check.

Property HUSH-15: Wallet signatures used for key hierarchy derivation are treated as secrets and never persisted, logged, or transmitted

- Threat a: Wallet signatures persisted to browser storage allowing extraction after the session ends

  The threat does not hold**.** The current implementation of `hush-wasm` doesn't use any persistent storage. The functions in `hush-wasm` are pure functions without states. The wallet related relevant code is in `web/` directory — which is out of scope.

- Threat b: Wallet signatures logged to console, debug logs, or error messages allowing exposure through developer tools or log aggregation

  The threat does not hold. The current implementation of `hush-wasm` does not log or output wallet signatures, but errors should still be improved as mentioned in "*Miscellaneous findings on hush-wasm*" to ensure no sensible data is exposed.

- Threat c: Wallet signatures transmitted over network (analytics, telemetry, API calls) allowing interception or server-side logging

  The threat does not hold**.** The current implementation of `hush-wasm` only uses pure functions without any side effects or states. So, there is no network access. The related relevant code is in `web/` directory — which is out of scope.

- Threat d: Wallet signatures not cleared from memory after `spending_key` derivation, leaving them accessible to memory inspection

  The threat holds. The variable `padded` in `fn rpo_hash_internal` can hold the signature but `padded.zeroize()` is not called.

- Threat e: Wallet signatures included in the serialized state for wallet recovery or session restoration

  The threat does not hold. The wallet signatures are not included in serialized states.

Property HUSH-16: Private keys (spending, nullifier, viewing) are never leaked through explicit channels (logging, network transmission, plaintext storage) or side-channels (timing, cache behavior)

- Threat a: Private keys logged to console, debug logs, or error messages allowing exposure through developer tools or log aggregation

  The threat does not hold. The current implementation of `hush-wasm` does not log or output private keys, but errors should still be improved as mentioned in "*Miscellaneous findings on hush-wasm*" to insure no sensible data is exposed.

- Threat b: Private keys transmitted over network (analytics, telemetry, API calls, WebSocket connections) allowing interception or server-side logging

  The threat does not hold. `hush-wasm` doesn't use any network access. Looking at the errors logs we don't find any private data leakage.

- Threat c: Private keys stored in plaintext in browser storage without encryption

  The threat does not hold. The `hush-wasm` uses pure functions without persistent states.

- Threat d: Private keys exposed through JavaScript error stack traces or exception messages

  The threat does not hold. The error strings are free of private data.

- Threat e: Private keys passed to third-party libraries or browser extensions with excessive permissions

  The threat does not hold. `hush-wasm` is self contained without any unreasonable cargo dependencies. The wasm blob will be compiled by code authors, whose responsibility is to make sure the cargo dependencies are not compromised.

- Threat f: Private keys not cleared from memory after cryptographic operations, leaving them accessible to memory inspection

  The threat holds. As `zeroize()` is not called on all variables holding private key information when they are not used anymore.

- Threat g: Secret-dependent comparisons (key verification, nullifier checks) not using constant-time algorithms, leaking information through timing differences

The threat does not hold. There are few usages of `ct_eq` but there are still some (in)equalities that use Rust native operations on `Vec` types. They will not execute in constant-time. But these are on public on-chain data, including the existing `eq_ct` calls which return the matched data.

- Threat h: Cryptographic operations have variable execution time or secret-dependent branches, enabling timing or cache-based side-channel attacks.

  The threat does not hold. There are few cases where the loop terminated if some condition is met. But the data are on-chain public data.

Findings

Name	Type	Severity	Status
Unshield recipient address not cryptographically bound to ZK proof	Design	4 - Critical	Resolved
Note secret and randomness not cryptographically bound to spending key	Design	4 - Critical	Resolved
Cross-asset theft vulnerability	Design	4 - Critical	Resolved
Circumventing fees	Design	4 - Critical	Resolved
Reusing balance nullifier	Implementation	4 - Critical	Resolved
Hardcoded note sequence limit	Implementation	4 - Critical	Resolved
Integer overflow in balance accumulation	Implementation	3 - High	Resolved
Unsanitized u64 as field element	Implementation	3 - High	Resolved
Missing shared secret validation	Implementation	3 - High	Resolved
Stealth recovery mismatch	Implementation	3 - High	Resolved
Missing message validation in x/hush handlers	Implementation	3 - High	Resolved
Cross-chain linkability	Design	2 - Medium	Resolved
Mempool proof replay attack	Implementation	2 - Medium	Resolved
Missing tree depth validation makes the contract unusable	Implementation	2 - Medium	Resolved

Name	Type	Severity	Status
Lack of confirmation during admin updates	Implementation	2 - Medium	Resolved
Nullifier key derivation mismatch in account recovery	Implementation	2 - Medium	Resolved
Erroneous supply stats	Implementation	2 - Medium	Resolved
Unbounded Merkle Depth (DoS vector)	Implementation	2 - Medium	Resolved
Unbounded JSON string (DoS vector)	Implementation	2 - Medium	Resolved
Missing host-side new balance validation (DoS vector)	Implementation	2 - Medium	Resolved
Silent recipient string truncation	Implementation	2 - Medium	Resolved
AddCommitment overwrites	Design	2 - Medium	Resolved
Missing check for leaf_index	Implementation	1 - Low	Resolved
Commitment field ordering inconsistency	Implementation	1 - Low	Resolved
Duplicate vouchers compute wrong balance	Implementation	1 - Low	Resolved
Missing integer overflow check in x/hush module	Implementation	0 - Informational	Resolved
Note secret derivation inconsistency between balance notes and vouchers	Implementation	0 - Informational	Resolved
Duplicate incoming notes not validated	Implementation	0 - Informational	Resolved

Name	Type	Severity	Status
Viewing key lifetime leak	Design	0 - Informational	Resolved
Miscellaneous findings on hush-wasm	Implementation	0 - Informational	Reported
Miscellaneous findings in hush.masm	Implementation	0 - Informational	Resolved
Miscellaneous comments on x/hush	Implementation	0 - Informational	Reported
Miscellaneous findings on CW contracts	Implementation	0 - Informational	Resolved

Table 2: Identified Security Findings

Unshield recipient address not cryptographically bound to ZK proof

Severity Critical **Exploitability** High **Status** Resolved

Type Design **Impact** High

Involved artifacts

- `zrchain/contracts/miden-circuits/hush.masm`

Description

The protocol fails to cryptographically bind the `recipient_address` (Solana destination) to the ZK proof for unshield operations. The circuit's `outputs_commitment` includes the nullifiers, commitments, amounts, and fees, but sets `recipient_commitment = zeros` for unshields (code ref). The `recipient_address` is never included in the cryptographic proof: it's only validated for base58 format after proof verification. This allows anyone who obtains a valid proof to redirect the unshielded funds to an arbitrary Solana address by constructing a new transaction with the same proof but a different recipient address.

Problem scenarios

- **Mempool front-running**: Alice broadcasts an unshield transaction with her proof and Solana address. Bob monitors the mempool, extracts the proof bytes, and submits his own transaction with the same proof but his Solana address, using higher gas to front-run Alice. The proof verifies successfully (all cryptographically-bound fields match), and funds are sent to Bob's address instead.
- **Proof interception**: Alice generates a proof on a compromised client or transmits it over an insecure channel. An attacker intercepts the serialized proof before Alice broadcasts the transaction. The attacker constructs their own `MsgUnshield` with the stolen proof, their own Cosmos signing key, and their own Solana recipient address. The chain accepts this transaction since the proof is mathematically valid and the recipient address is unconstrained.

Recommendation

Include the `recipient_address` in the circuit's `outputs_commitment` by hashing it into the commitment structure. The circuit should accept the recipient address (or its hash) as part of the public inputs and include it in the outputs commitment calculation. This ensures that any modification to the recipient address invalidates the proof. This requires circuit changes to accept and validate the recipient binding, and client changes to include the recipient address when generating proofs.

Resolution

The development team has addressed this finding in PR #839.

Note secret and randomness not cryptographically bound to spending key

Severity Critical **Exploitability** High **Status** Resolved

Type Design **Impact** High

Involved artifacts

- `zrchain/contracts/miden-circuits/hush.masm`

Description

The circuit in `hush.masm` does not cryptographically bind `note_secret` and `randomness` values to the `spending_key`, allowing complete bypass of double-spend protection. The circuit derives `nullifier_key` from `spending_key` but loads all `note_secret` and `randomness` values directly from the advice stack without verification (code ref).

- The circuit derives `nullifier_key = hash(spending_key || NULLIFIER_KEY_DOMAIN)` (code ref)
- Nullifiers are computed as `nullifier = hash(nullifier_key, commitment)` (code ref)
- Commitments depend on `note_secret` and `randomness`. However, `note_secret` and `randomness` values are loaded via `adv_push.4` for balance note (code ref) and incoming notes (code ref) with no verification that they were derived from the same `spending_key`.

An attacker can use the same commitment with multiple different spending keys to generate unique nullifiers, spending the same funds multiple times. Since the chain only tracks nullifiers (not commitments), each spend appears valid.

Problem scenarios

Double-spend balance note

Prerequisites:

- Attacker controls a valid balance note with commitment `C` and amount `A`.

Attack steps:

1. Create spending key `spending_key_1` → derive `nullifier_key_1` → generate `nullifier_1 = hash(hash(nullifier_key_1, C)`.
2. Submit transaction, spend the funds.
3. Create another spending key `spending_key_2` → derive `nullifier_key_2` → generate `nullifier_2 = hash(hash(nullifier_key_2, C)`.
4. Submit second transaction with the same commitment `C` but different nullifier.
5. Chain accepts both transactions because `nullifier_1 ≠ nullifier_2`.

Impact: Attacker spends the same balance note twice (or more), creating unbacked tokens.

Double-spend of stealth transfer incoming notes

Prerequisites:

- Legitimate recipient receives a stealth transfer (commitment `C` is public on-chain)
- Recipient scans the transfer and learns the pre-image: `note_secret`, `randomness`, `amount`, `asset`, `seq`.

Attack steps:

1. Attacker (who knows the pre-image) generates `spending_key_1` → derives `nullifier_key_1` → computes `nullifier_1 = hash(nullifier_key_1, C)`.
2. Attacker creates a valid proof with the known pre-image (`note_secret`, `randomness`, `amount`, `asset`, `seq`) and Merkle path.
3. Chain accepts the proof and marks `nullifier_1` as spent.
4. Attacker generates `spending_key_2` → derives `nullifier_key_2` → computes `nullifier_2 = hash(nullifier_key_2, C)`.
5. Attacker creates another proof for the same commitment `C` using the same pre-image but different spending key.
6. Chain accepts because `nullifier_2` is different from `nullifier_1`
7. Repeat with `spending_key_3`, `spending_key_4`, … unlimited times

Impact: Legitimate recipient can double-spend their own incoming notes unlimited times

Recommendation

The circuit must enforce that all `note_secret` (and ideally also `randomness`) values are cryptographically derived from the `spending_key` rather than accepting them as arbitrary advice inputs. For balance notes and new balance notes, the circuit should derive `note_secret` (and `randomness` if possible) internally instead of loading from advice. For incoming notes from stealth transfers, the protocol might require a redesign: for example, the sender could derive the `note_secret` (and `randomness` if possible) using the ECDH shared secret and the recipient `spending_key`.

Resolution

The development team has addressed this finding in PR #846.

Cross-asset theft vulnerability

Severity Critical **Exploitability** High **Status** Resolved

Type Design **Impact** High

Involved artifacts

- `zrchain/x/hush/keeper/merkle.go`
- `zrchain/x/hush/keeper/msg_server.go`

Description

The protocol's circuit architecture contains a security vulnerability that allows an attacker to unshield tokens as a different asset type than originally shielded. This enables cross-asset theft attacks where one asset's vault can be drained using a proof generated for a completely different asset.

The root cause is that the asset type is neither verified either in the circuit or the `Unshield` message handler.

Root causes:

1. **Asset not bound to proof:** The asset type was included in V6's public inputs but was **dropped in V7** (likely due to the 8-element stack limit). The `msg.Asset` field is only used after proof verification completes, meaning the verifier never checks which asset the proof was actually generated for.
2. **Global supply tracking:** The `TotalShielded` supply counter is tracked globally across all assets, not per-asset. This means a valid unshield of "100 units" passes the supply check regardless of which asset those units belong to—allowing an attacker to drain any vault as long as the total global supply is sufficient.

Combined, these issues allow an attacker to generate a proof for Asset A but claim the withdrawal as Asset B, with proof verification succeeding because the verifier is asset-agnostic.

Problem scenarios

Given two vaults containing Asset A (e.g., jitoSOL) and Asset B (e.g., zenBTC), an attacker can achieve cross-asset theft through the following steps:

1. Shield tokens of Asset A:
   - The attacker shields 100 jitoSOL, creating a commitment `C = hash(note_secret, [100, 0, 0, assetA])`.
2. Generate a valid ZK proof for Asset A:
   - Ownership of the commitment in the Merkle tree
   - Correct balance conservation (balance ⟶ new_balance + amount + fee)
   - The circuit uses the private input asset=A internally to match the commitment
3. Submit Unshield with `msg.Asset` = B:
   - `msg.Asset` = zenBTC ⟵ **Exploits vulnerability**
   - `msg.Amount` = 100 ⟵ Matches the proof

- `msg.RecipientAddress` = attackerWallet
- `msg.ZkProof` = validProofForAssetA

4. Proof verification succeeds:
   - `ComputeOutputsCommitmentV7()` hashes the message commitments without including the asset
   - The circuit's internal hash matches the on-chain computed hash
   - `msg.Asset` is only used after verification for chain routing, not validation
   - `UnshieldRequest` is created with both Asset=zenBTC and `Caip2ChainId` mapping to zenBTC and not jitoSOL
5. Global supply check passes:
   - `TotalShielded >= msg.Amount + msg.Fee` checks against the total shielded (cross-assets) counter
   - The system doesn't verify that Asset B specifically has sufficient shielded funds
6. ABCI routes to wrong vault:
   1. The `UnshieldRequest` is created with Asset=zenBTC thus is picked up by the ABCI handler
   - The ABCI handler looks up zenBTC's Solana program and transfers 100 zenBTC from the zenBTC vault to the attacker

Result: The attacker receives 100 zenBTC (from another user's deposit) while their jitoSOL commitment is nullified. The jitoSOL vault retains its 100 tokens (orphaned), and the zenBTC vault is drained.

Recommendation

- **Add asset to outputs commitment:** Include `msg.Asset` in `ComputeOutputsCommitmentV7()` so the proof is cryptographically bound to a specific asset. Requires circuit and client library updates.
- **Per-asset supply tracking**: Replace: global `TotalShielded` with per-asset counters (`map[ShieldAsset]*AssetSupply`). Prevents cross-asset balance exploitation even if other checks fail.

Resolution

The development team has addressed this finding in PR #846.

Circumventing fees

Severity Critical **Exploitability** High **Status** Resolved

Type Design **Impact** High

Involved artifacts

- `zenrock/zrchain/x/hush/keeper/msg_server.go`

Description

There is no guarantee that fees are paid when a user submits a shielded transfer due to missing logic and constraints in `ShieldedTransfer`.

Problem scenarios

A user can simply skip from providing a `FeeCommitment`. As a result, `hasFeeCommitment` is not set and as a result the fee-voucher-related code is not even called. Even worse, even if a `FeeCommitment` is provided, there is no check that this commitment can be spent by the fee collector in any way, so the created fee voucher remains unspendable. Or the user could just set a `FeeCommitment` in such a way as to get the fees back.

Recommendation

The solution presented PR #846 seems to be going towards the right approach. Just update *fee stats* during shield transfers and then based on those stats, allow governance to claim those protocol fees.

Resolution

The development team addressed this finding in commit `83d5620`.

Reusing balance nullifier

Severity Critical **Exploitability** High **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zenrock/zrchain/x/hush/keeper/msg_server.go`

Description

There is no check on whether a nullifier is used both as a *balance* and an *incoming* nullifier.

Problem scenarios

User submits a `MsgUnshield` or a `MsgShieldedTransfer` message with a balance nullifier `B` and includes this exact same nullifier `B` in the incoming nullifiers. Although both `Unshield` and `ShieldedTransfer` check for duplicate incoming nullifiers, there is no check that a balance nullifier is not reused as an incoming nullifier. As a result, a user can shield 1 token, submit an `MsgUnshield` for that token and reuse the balance nullifier in the incoming nullifier, ending up getting 2 tokens back (i.e., one from the balance and one due to the incoming nullifier). Note that this is possible because the balance nullifier is unspent at the beginning of the `IsNullifierSpent` check.

Recommendation

PR #839 already added a check that the balance nullifier does not appear in the incoming nullifiers. But what might be even better is to immediately **mark** a nullifier (`MarkNullifierSpent`) after seeing that it is not spent (i.e., `IsNullifierSpent`). Because there is no reason to only mark the nullifier much later on because in any case the `Unshield` and `MsgUnshield` operate on a cached context so if the operation fails, the *markings* will be reversed.

Resolution

The development team has addressed this finding in PR #839.

Hardcoded note sequence limit

Severity Critical **Exploitability** High **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

Uses only `[0..1000)` range to find `balance_note`.

Problem scenarios

If `balance_note` is created with a higher sequence, it's not recovered.

Recommendation

Use a configurable or guessed sequence set as input to iterate over.

Resolution

The development team has addressed this finding in PR #846.

Integer overflow in balance accumulation

Severity High **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/contracts/miden-circuits/hush.masm`

Description

The circuit accumulates input and output amounts through multiple field addition operations without overflow detection. Since Miden VM uses Goldilocks field arithmetic (modulus `p = 2^64 - 2^32 + 1`), when accumulated values exceed this modulus, they automatically wrap around via modular reduction. This wraparound breaks the conservation of value invariant, allowing attackers to craft transactions where the balance equation `total_input == total_output` passes despite having vastly different actual amounts.

The vulnerable accumulation operations occur at:

- Accumulate balance note amount into `total_input` (code ref).
- Accumulate incoming note amounts into `total_input` (up to 24 notes) (code ref).
- Accumulate new balance amount into `total_output` (mem_load.68 mem_load.34 add mem_store.34).
- Add `recipient_amount` to `total_output` (code ref).
- Add fee to `total_output` (code ref).

And then the balance is verified (code ref).

Neither the circuit nor `hush-wasm` validate that individual amounts or their sum remain below the field modulus. Amounts are pushed to the advice tape without sanitization at lines, and the circuit performs no bounds checking before accumulation.

Problem scenarios

When a user accumulates multiple large shielded transfers over time the sum of their balance note and incoming notes can legitimately exceed the Goldilocks field modulus during normal protocol usage. Since the circuit performs field additions without overflow detection, this sum wraps around modulo `p` to a much smaller value. The balance verification then compares this wrapped input total against the output total, which the attacker can craft to match the wrapped value rather than the true value. This allows transactions where the actual inputs far exceed the outputs (causing token loss) or where outputs exceed inputs (creating unbacked tokens), yet the circuit's balance check passes because both sides wrap to the same incorrect value.

Recommendation

- **Input sanitization in `hush-wasm`**: Before pushing amounts to the advice tape, apply `to_field_safe()` function to reduce any values >= `GOLDILOCKS_MODULUS`. Additionally, validate that each individual amount

(balance, incoming notes, new balance, recipient, fee) does not exceed a reasonable maximum to ensure even with 24 incoming notes plus balance, the sum cannot overflow. Also perform pre-flight validation by computing total inputs and total outputs and rejecting if either sum would exceed `GOLDILOCKS_MODULUS`.

- **Circuit-level bounds enforcement**: At each `adv_push.1` operation that loads an amount add an assertion to verify the loaded value is within the safe maximum (e.g., `<= 2^62`). This creates a hard constraint that malicious clients cannot bypass and guarantee that accumulation cannot overflow (i.e., exceed `GOLDILOCKS_MODULUS`) regardless of how amounts are combined.

Resolution

The development team has implemented `felt_from_u64` and introduced checked addition across `hush-wasm` in PR #903. Additionally in PR #913 development team introduced validation for individual amount bounds `< 2^59` and sum bounds (`total_input` and `total_output`) to be `< GOLDILOCKS_MODULUS` (check added both in `hush-wasm` and in `hush.masm`).

Unsanitized u64 as field element

Severity High **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs` (code ref 1, code ref 2, code ref 3, etc)
- `zrchain/contracts/miden-merkle/src/contract.rs` (code ref)

Description

`u64` is directly as field elements without validating they are less than `GOLDILOCKS_MODULOUS`.

Problem scenarios

A high value `u64` maybe equivalent to low value `u64` in the field operation, resulting attack vectors.

Recommendation

Validate the balances to be less then `GOLDILOCKS_MODULUS`.

Resolution

The development team addressed this finding in PR #851.

Missing shared secret validation

Severity High **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The shared secret generated with ECDH should be rejected if it's all-zeros to avoid encrypted values being decrypted by other keys.

Problem scenarios

If the shared secret is not validated and is all-zeros, values encrypted with that shared secret can be decrypted by other parties.

```rust
#[test]
fn x25519_all_zero_shared_secret_with_zero_public_key() {
    use x25519_dalek::{PublicKey, StaticSecret};

    let amount = 123456789u64;

    // Generate random sender randomness for nonce derivation
    let mut sender_randomness = [0u8; 12];
    getrandom::getrandom(&mut sender_randomness).map_err(|e| e.to_string()).unwrap();

    let mut nonce_input = Vec::new();
    nonce_input.extend_from_slice(&sender_randomness);
    nonce_input.extend_from_slice(ENCRYPTION_NONCE_DOMAIN);
    let nonce_hash = rpo_hash_internal(&nonce_input).unwrap();
    let mut amount_nonce = [0u8; 12];
    amount_nonce.copy_from_slice(&nonce_hash[..12]);

    // Two different private keys for testing
    let my_secret = StaticSecret::from([42u8; 32]);
    let my_secret2 = StaticSecret::from([51u8; 32]);

    // Attacker-provided peer public key = all zeros (a small-order point encoding).
    let their_pub = PublicKey::from([0u8; 32]);

    let shared = my_secret.diffie_hellman(&their_pub);
```

```
26    let shared2 = my_secret2.diffie_hellman(&their_pub);
27
28    assert_eq!(shared.as_bytes(), &[0u8; 32], "hared secret should be all-zero");
29    assert_eq!(shared.as_bytes(), &[0u8; 32], "shared secret should be all-zero");
30
31    let derived_key_1 = derive_amount_key_internal(shared.as_bytes()).unwrap();
32    let derived_key_2 = derive_amount_key_internal(shared2.as_bytes()).unwrap();
33
34    assert_eq!(derived_key_1, derived_key_2, "amount keys should match for same all-
      zero shared secret");
35
36    // Encrypt with derived key 1
37    let encrypted_amount = encrypt_amount_internal(&derived_key_1, amount,
      &amount_nonce).unwrap();
38    // Decrypt with derived key 2
39    let decrypted_amount = decrypt_amount_internal(&derived_key_2,
      encrypted_amount.as_slice()).unwrap();
40
41    assert_eq!(amount, decrypted_amount, "decrypted amount should match original");
42  }
```

Recommendation

Add validation of shared secrets `shared_secret != [0u8; 32]` (e.g., here).

Resoultion

The development team has addressed this finding in PR #869.

Stealth recovery mismatch

Severity High **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

Stealth transfer creation builds `note_secret` (code ref) and `randomness` (code ref) from the shared secret with a fixed label. Recovery builds them from shared secret and `leaf_index` (code ref 1, code ref 2). So the recovery path makes different secrets than what the transfer used. The commitment check fails and the note gets missed.

Problem scenarios

Alice sends a stealth transfer to Bob. Then Bob runs recovery later. But the note doesn't show up.

Recommendation

Use consistent derivation rule for note secret and randomness.

Resolution

The development team has addressed this finding in PR #903.

Missing message validation in x/hush handlers

Severity High **Exploitability** High **Status** Resolved

Type Implementation **Impact** Medium

Involved artifacts

- `zrchain/x/hush/keeper/msg_server.go`

Description

The `MsgUnshield` and `MsgShieldedTransfer` handlers do not validate the size of `EncryptedNewBalanceIndex` and `EncryptedRecipientIndex` fields. While other encrypted fields like `EncryptedNewBalanceAmount` are validated to be exactly 36 bytes, these index fields accept arbitrary-length byte arrays that get stored permanently in chain state.

Problem scenarios

An attacker submits a valid `MsgUnshield` transaction with a multi-megabyte `EncryptedNewBalanceIndex` field filled with garbage data. The transaction passes all validation checks (ZK proof, nullifiers, commitments) and succeeds. The garbage data is stored in both the `UnshieldRequest` and Voucher records, bloating validator state. While bounded by Cosmos SDK's ~5MB transaction limit and gas costs, repeated attacks could accumulate gigabytes of unnecessary storage.

Recommendation

Add length validation for the missing fields following the Cosmos SDK `ValidateBasic` pattern, where stateless message validation should reject malformed inputs before any state access occurs:

```go
// In Unshield and ShieldedTransfer:
if len(msg.EncryptedNewBalanceIndex) > 0 && len(msg.EncryptedNewBalanceIndex) != 36 {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidRequest, "encrypted_new_balance_index must be 36 bytes")
}

// Additionally in ShieldedTransfer:
if len(msg.EncryptedRecipientIndex) > 0 && len(msg.EncryptedRecipientIndex) != 36 {
    return nil, errorsmod.Wrap(sdkerrors.ErrInvalidRequest, "encrypted_recipient_index must be 36 bytes")
}
```

Resolution

The development team addressed this finding in commit `83d5620`.

Cross-chain linkability

Severity Medium **Exploitability** Medium **Status** Resolved

Type Design **Impact** Medium

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`
- `zrchain/contracts/miden-circuits/hush.masm`

Description

Right now the key derivations (`spending_key` → `note_secret` / `randomness` / `nullifier_key`) and the commitment hash don't mix in a `CHAIN_ID`. That means the same signature and inputs can produce the same keys and commitments on multiple networks. So a wallet can accidentally reuse secrets across chains, and the commitments can be linked across networks.

Problem scenarios

- You use the same wallet signature on mainnet + testnet, and the `spending_key` / `nullifier_key` end up identical on both.
- A note with the same amount/asset/randomness can produce the same commitment on different networks, so someone can match it across chains.
- An auditor watching multiple networks can link activity just by matching commitments or nullifiers.

Recommendation

Include `CHAIN_ID` in key management and the commitment input:

- Mix `CHAIN_ID` into `derive_spending_key_internal` (or into the domain separators) so keys are chain-specific.
- Carry that through the rest of the hierarchy (`note_secret`, `randomness`, `nullifier_key`).
- Add `CHAIN_ID` into the commitment formula in both Rust and the MASM circuit so commitments are chain-specific too.

Resolution

The development team has addressed this finding in commit `75f6e80`.

Mempool proof replay attack

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/x/hush/keeper/merkle.go`
- `zrchain/x/hush/keeper/msg_server.go`

Description

The `x/hush` module's `ShieldedTransfer` function is vulnerable to a mempool proof replay attack that allows an attacker to copy a valid transaction from the mempool, modify the `msg.Asset` and `msg.Creator` fields, and front-run the original sender.

Root causes:

1. **Asset not bound to proof:** The `msg.Asset` field is not included in V7's outputs commitment. The only value that changes with asset is the fee parameter, meaning proofs are interchangeable between assets if their `ShieldedTransferFee` values are identical.
2. **Proof validation is asset-agnostic:** The ZK proof verifies that commitments are correctly formed and conservation holds, but doesn't validate which asset those commitments belong to—that check happens implicitly via fee matching.

Current protection (weak)**:** The attack only succeeds when two assets have the same `ShieldedTransferFee`. Current fee differences (jitoSOL: 10000000, zenBTC: 10000) prevent exploitation, but this is security-by-coincidence, not security-by-design.

Problem scenarios

Given two assets with identical `ShieldedTransferFee` values, an attacker can hijack a shielded transfer:

1. User A submits a `MsgShieldedTransfer` to transfer jitoSOL within the shielded pool:
   - `msg.Creator` = "zen1user…"
   - `msg.Asset` = jitoSOL
   - `msg.ZkProof` = P (valid proof)
   - `msg.RecipientCommitment`, `msg.NewBalanceCommitment`, etc.
2. Attacker observes mempool and front-runs user A's transaction with:
   - `msg.Creator` = "zen1attacker…" ⟵ changed to attacker's address
   - `msg.Asset` = zenBTC ⟵ changed
   - `msg.ZkProof` = P and all other fields copied from the original
3. Proof verification succeeds:
   - `ComputeOutputsCommitmentV7()` computes the same hash (asset not included)
   - The fee value matches (by assumption)
   - The circuit internally verified conservation using the original private inputs

- Neither `msg.Creator` nor `msg.Asset` affect the hash

4. Attacker's transaction succeeds and corrupts the states :
   - Nullifiers are marked as spent
   - Vouchers are created with **wrong asset tags** (zenBTC instead of jitoSOL)
   - `ShieldedTransfer` record stored with the attacker as `msg.Creator`
5. User A's transaction is rejected:
   - "nullifier already spent" error
   - User A cannot retry (nullifiers are consumed)

Result:

- User A's transfer fails (DoS)
- Vouchers are tagged as zenBTC instead of jitoSOL (corruption)
- Recipient voucher routes to wrong Solana program at unshield time
- No funds are stolen (commitments remain cryptographically bound to original owners)

Recommendation

- **Add asset to outputs commitment**: Include `msg.Asset` in the metadata word. Prevents cross-asset proof reuse regardless of fee values.
- **Enforce unique fees per asset:** Governance constraint preventing identical `ShieldedTransferFee` values.

Resolution

The development team has addressed this finding in PR #846.

Missing tree depth validation makes the contract unusable

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/contracts/miden-merkle/src/contract.rs`

Description

The `instantiate()` function in the `miden-merkle` contract accepts a `tree_depth` parameter from `InstantiateMsg` without validating its upper bound.

The vulnerability manifests in two locations. First, in the `max_tree_capacity()` helper function (code ref), the expression `1u64 << depth` overflows when `depth >= 64`. Second, in the `node_position()` function (code ref), the expression `((level as u64) << depth) + index` can overflow when depth values approach 64.

Since the tree depth is set during initialization and cannot be changed afterward, an invalid depth value permanently breaks the contract instance, requiring redeployment with correct parameters.

These overflows prevent the contract from storing any commitments, making it basically unusable.

Test

This is a test showcasing this issue. We instantiate the contract with a `three_depth` of 64. After that, we try to add a commitment and the contract panics with: "*attempt to shift left with overflow*".

```rust
#[test]
#[should_panic(expected = "attempt to shift left with overflow")]
fn test_instantiate_invalid_tree_depth() {
    let mut deps = mock_dependencies();
    let msg = InstantiateMsg { tree_depth: 64, history_size: None };
    let info = mock_info("creator", &[]);
    let res = instantiate(deps.as_mut(), mock_env(), info, msg).unwrap();
    assert_eq!(res.attributes.len(), 3);
    assert!(res.attributes.iter().any(|a| a.key == "tree_depth" && a.value == "64"));

    // Now we try to add a commitment
    let commitment = vec![1u8; 32];
    let sudo_msg = SudoMsg::AddCommitment { commitment };
    sudo(deps.as_mut(), mock_env(), sudo_msg).unwrap();
}
```

Problem scenarios

The deployer of the contract mistakenly sets the `tree_depth` parameter to be ≥ 64. The contract is instantiated correctly, but on the first `sudo_add_commitment()` call, the overflow occurs, making it impossible to add any commitments to the tree.

Recommendation

Add explicit validation in the `instantiate()` function to reject tree depths that could cause arithmetic overflow. The maximum safe depth for a u64-based tree is 63, as `1u64 << 63` produces a valid result while `1u64 << 64` overflows.

Alternatively, if the tree depth is predetermined for the deployment environment, consider defining it as a compile-time constant rather than accepting it as a runtime parameter, eliminating the validation requirement entirely.

Resolution

The development team has addressed this finding in PR #869.

Lack of confirmation during admin updates

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/contracts/miden-merkle/src/contract.rs`

Description

The admin update process implemented in the `execute_update_admin()` function does not follow the best practices. The process is implemented in a single step, where the admin only overwrites its address by adding the new admin.

As a result, it is possible to assign control of the contract to an incorrect address. Once such an update occurs, the contract's administrative functions become permanently inaccessible, effectively locking the contract's configuration and administrator capabilities.

Problem scenarios

An admin unintentionally sets another address as the admin without that entity's knowledge or ability to interact. The system remains operational, but future administrative actions are impossible.

Recommendation

To mitigate the risk of administrative lockout and to ensure that admin updates are both valid and explicitly acknowledged, the contract should use the two-step ownership transfer pattern provided by the `cw-ownable` library.

The library enforces a proposal → acceptance workflow before ownership changes are finalized. It ensures that no contract can become ownerless or accidentally locked due to invalid admin updates.

Resolution

The development team has addressed this finding in PR #846.

Nullifier key derivation mismatch in account recovery

Severity Medium **Exploitability** High **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The `recover_account_state` function uses `derive_nullifier_key_internal` (code ref) to derive the nullifier key, while all other functions like `derive_voucher_v2` (code ref), `create_balance_note_internal` (code ref), and `derive_full_viewing_key` (code ref) use `derive_nullifier_key_circuit_internal`.

These two functions use different hashing approaches:

- `derive_nullifier_key_internal` (code ref): concatenates spending_key with domain bytes and uses `Rpo256::hash_elements`
- `derive_nullifier_key_circuit_internal`: structures inputs to match the circuit's `hmerge` instruction then calls `vm_hmerge` which uses `Rpo256::merge` with the exact state layout that the VM uses internally

They produce completely different outputs for the same spending_key.

Test

Rust

```rust
#[test]
fn test_nullifier_key_derivation_mismatch() {
  let spending_key = [0x41u8; 32];

  let result_circuit = derive_nullifier_key_circuit_internal(&spending_key).unwrap();
   let result_internal = derive_nullifier_key_internal(&spending_key).unwrap();

   println!("Circuit:  {}", hex::encode(&result_circuit));
   println!("Internal: {}", hex::encode(&result_internal));

   // This assertion will FAIL, demonstrating the bug
   assert_eq!(
      result_circuit, result_internal,
      "Circuit and internal derivation outputs are not the same!"
   );
}
```

Test fails with output:

```
Circuit:  943645f1bbd5dcd2bb30e78cd972326dc3e5ab3277634cf0cb0457f5343ddf12
Internal: 7da21cb21e6bd68e7bc80d9ff244413e02cb23ce3cd696c07f45c05f91ce2de1
```

Problem scenarios

When a user calls `recover_account_state` to scan their vouchers and determine which are spent:

1. The function derives the wrong nullifier_key using `derive_nullifier_key_internal`
2. For each voucher, it computes nullifiers using the wrong key
3. It compares against on-chain spent nullifiers (which were computed using the circuit-compatible derivation)
4. No matches are found because the nullifiers are completely different
5. All vouchers appear unspent, even if they were already spent

If the user attempts to spend a voucher that appears unspent but is actually spent, the ZK proof generation will use the correct circuit-compatible derivation, produce the correct nullifier, and the chain will reject the transaction because that nullifier is already marked as spent.

Recommendation

Replace `derive_nullifier_key_internal` with `derive_nullifier_key_circuit_internal` in `recover_account_state`. Additionally, consider deprecating or removing `derive_nullifier_key_internal` to prevent future misuse.

Resolution

The development team has addressed this finding in commit `069c281`.

Erroneous supply stats

Severity Medium **Exploitability** High **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `zenrock/zrchain/x/validation/keeper/abci_hush.go`
- `zenrock/zrchain/x/hush/keeper/keeper.go`

Description

If a user (by accident or maliciously) performs two shielding events with the exact **same** commitment it can lead to erroneous accounting statistics and a lingering voucher that the user cannot use.

Problem scenarios

Specifically, a can user initiate two shielding events with the exact same commitment that are both processed in `processShieldEvents`. During the first processing of the shield event, a voucher would be created for this commitment. Then, when the second shield event is being processed (this can happen because it has a different `shieldEvent.TxId` and `shieldEvent.LogIndex` than the first shield event), it also creates a voucher for the exact same commitment and furthermore *overwrite* `CommitmentToVoucherStore` with the `nextID` of this commitment. Now, when the user spends one voucher, the nullifier is marked as spent and the user cannot use the other voucher.

This leads to:

- Accounting of supply is wrong (due to `updateSupply` being `true`).
- Due to the overwrite of `CommitmentToVoucherStore` the user is not able to generate a proof to retrieve funds from the first voucher. This is good because it does **not** allow for double spending but the UX might be bad in the sense that the user sees a voucher that they **cannot** use.

Recommendation

Introduce a check before the `AddCommitment` call and in case the commitment already exists (e.g., by checking `CommitmentToVoucherStore`) return an `error`.

Resolution

The development team has addressed this finding in PR #869.

Unbounded Merkle Depth (DoS vector)

Severity Medium **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** Medium

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

`depth` (which is `siblings.len() / 32`) is never checked for bounds (code ref).

Problem scenarios

An attacker can pass a `siblings` array of arbitrary size (e.g., 10MB = 327,680 siblings = depth 327,680), causing the loop to execute millions of times with expensive hash operations per iteration.

Recommendation

```rust
const MAX_MERKLE_DEPTH: usize = 64; // Reasonable upper bound
let depth = siblings.len() / 32;
if depth > MAX_MERKLE_DEPTH {
  return Err(JsValue::from_str("Merkle depth exceeds maximum allowed"));
}
```

Resolution

The development team has addressed this finding in commit `3edd36f`.

Unbounded JSON string (DoS vector)

Severity Medium **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** Medium

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The `serde_json::from_str` calls are on `&str` with unbounded length.

Problem scenarios

An attacker (requires highjacked wallet or RPC endpoint) can submit a really large JSON which is processed by wasm functions leading to high memory and CPU usage.

Recommendation

Validate lengths of all unbounded input bytes and strings with a reasonable upper bound.

Resolution

The development team has addressed this finding in commit `3edd36f`.

Missing host-side new balance validation (DoS vector)

Severity Medium **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** Medium

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The `generate_account_proof_internal()` function accepts `new_balance_amount` as a parameter but performs no validation that:

1. `new_balance_amount = old_balance + incoming_total - recipient_amount - fee`
2. Amounts don't overflow
3. Fee is correctly applied

Problem scenarios

While the proof will correctly fail if amounts don't balance, the Rust host code should validate inputs before expensive proof computation (~1-8 seconds). An attacker (requires browser wallet or RPC endpoint highjacking) could:

- Submit `new_balance_amount = u64::MAX` (invalid)
- Trigger expensive STARK proof generation (wasting CPU)
- Proof fails at circuit assertion (but attacker already wasted resources)

This is a DoS vector, not an infinite money bug (circuit prevents actual exploit).

Recommendation

```rust
// Validate amount conservation
let incoming_sum: u64 = incoming_notes.iter().map(|n| n.amount).sum();
let old_balance = balance_note.as_ref().map(|n| n.amount).unwrap_or(0);
let expected_new_balance = old_balance.checked_add(incoming_sum)
  .and_then(|sum| sum.checked_sub(recipient_amount))
  .and_then(|sum| sum.checked_sub(fee))
  .ok_or("Amount overflow or underflow")?;
if new_balance_amount != expected_new_balance {
  return Err("new_balance_amount does not match expected value".to_string());
}
```

Resolution

The development team has addressed this finding in commit `3edd36f`.

Silent recipient string truncation

Severity Medium **Exploitability** Low **Status** Resolved

Type Implementation **Impact** High

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

Only first 32 bytes are copied from `recipient_bytes`.

Problem scenarios

Potential fund loss or transaction errors.

Recommendation

```rust
if recipient_bytes.len() > 32 {
  return Err(JsValue::from_str("Recipient address exceeds 32 bytes"));
}
```

Resolution

The development team has addressed this finding in commit `3edd36f`.

AddCommitment overwrites

Severity Medium **Exploitability** High **Status** Resolved

Type Design **Impact** Low

Involved artifacts

- `zrchain/x/hush/keeper/keeper.go`
- `zrchain/x/hush/keeper/merkle.go`

Description

We can have multiple `AddCommitment` calls in the same block for a specific height `H` and as a result the root for `H` will overwrite the previously written root for height `H`. This means that a user might have to generate a proof again because their proof is based on an invalid root (i.e., cannot be found in `ROOT_HISTORY`) even though the user just generated the proof, leading to bad user experience.

Problem scenarios

If we have multiple `AddCommitment` calls in a block at height `H` (e.g., multiple `MsgShieldedTransfer` transactions in the same block), then each of those `AddCommitments` updates the root for this specific height `H`, overwriting the previous entry for height `H`. This can lead to cases where a user generates a proof based on some Merkle root `R` but when submitting the message with this proof, root `R` cannot be found because it has been overwritten.

Recommendation

Change the key of `ROOT_HISTORY` to not be block height but something globally unique such as the leaf index as done in PR #839.

Resolution

The development team has addressed this finding in PR #839.

Missing check for leaf_index

Severity Low **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The `leaf_index` is not validated that it is smaller than `2 ^ depth` (code ref).

Problem scenarios

Invalid `leaf_index` can be used to successfully call the function `compute_merkle_root`.

Recommendation

```rust
if leaf_index >= (1 << depth) { return Err(...); }
```

Resolution

The development team has addressed this finding in commit `16480dc`.

Commitment field ordering inconsistency

Severity Low **Exploitability** Low **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `zrchain/contracts/miden-merkle/src/contract.rs`

Description

The `miden-merkle` contract uses incorrect field ordering when computing commitments, resulting in a mismatch with the circuit and client implementations. In `query_compute_commitment` uses order `[0, 0, asset, amount]` (code ref) instead of stack order `[amount, 0, 0, asset]` (as done in `hush-wasm`'s `compute_commitment_internal` (code ref)).

Problem scenarios

When external tools query the miden-merkle contract to verify a commitment computed by `hush-wasm`, the hashes will not match.

Recommendation

Fix the implementation of `query_compute_commitment` to use stack order instead of logical order. Change from `[Word::new([ZERO, ZERO, Felt::new(asset), Felt::new(amount)])]` to `[Word::new([Felt::new(amount), ZERO, ZERO, Felt::new(asset)])]` so that after reversal before hashing in `hash_nodes`, the logical order is `[asset, 0, 0, amount]`.

Additionally, make sure values are converted to safe field elements before calling `Felt::new()`.

Resolution

The development team has addressed this finding in PR #855.

Duplicate vouchers compute wrong balance

Severity Low **Exploitability** Medium **Status** Resolved

Type Implementation **Impact** Low

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

The function `recover_account_state` takes `vouchers_json` from outside and just adds matching notes to a list without checking for duplicates. When calculating `total_balance`, it sums up all entries including the duplicates. If someone passes the same voucher 5 times, the balance shows 5x the real amount.

Problem scenarios

A malicious RPC or frontend could return the same voucher multiple times. User sees a fake inflated balance and might try to spend money they don't have. They waste CPU time generating a proof and gas submitting a tx that the chain will reject anyway.

Could also be used for scams by showing fake "proof" of wealth.

Recommendation

Use `HashSet` instead of `Vec` for `incoming_notes` (code ref).

A similar fix should go into `generate_account_proof` (code ref).

Resolution

The development team has addressed this finding in commit `16480dc`.

Missing integer overflow check in x/hush module

Severity Informational **Exploitability** None **Status** Resolved

Type Implementation **Impact** None

Involved artifacts

- `zrchain/x/validation/keeper/abci_hush.go`

Description

In the `processUnveilBroadcasting` function, the `TotalShielded` supply gets incremented by the shield events amount without checking for integer overflow.

Problem scenarios

If a new `ShieldAsset` is added with no economic cap, like Ethereum, repeated large deposits could accumulate to overflow.

Recommendation

Add integer overflow check in `createVoucherInternal`:

```JavaScript
if updateSupply && amount > 0 {
  supply, err := k.GetSupply(ctx)
  if err != nil {
      return 0, err
  }

  // Check for overflow before addition
  if amount > math.MaxUint64 - supply.TotalShielded {
      return 0, fmt.Errorf("supply overflow: adding %d to %d would exceed uint64",
      amount, supply.TotalShielded)
  }

  supply.TotalShielded += amount
  if err := k.SetSupply(ctx, supply); err != nil {
      return 0, err
  }

  // rest of function
```

Resolution

The development team addressed this finding in commit `30f9ae0`.

Note secret derivation inconsistency between balance notes and vouchers

Severity Informational **Exploitability** None **Status** Resolved

Type Implementation **Impact** None

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

Description

Balance notes and vouchers use different hashing methods for `note_secret` derivation:

- `derive_balance_note_secret_internal` (balance notes): uses `rpo_hash_internal`
- `derive_note_secret_circuit_internal` (vouchers): uses `vm_hmerge`

Currently the circuit takes `note_secret` as an input and does not verify its derivation from `spending_key`. However, if the circuit is ever updated to validate note_secret derivation in-circuit, balance notes using `derive_balance_note_secret_internal` would fail verification because `rpo_hash` produces different outputs than `vm_hmerge`.

Recommendation

Consider using circuit-compatible derivation (`vm_hmerge`) for all `note_secret` derivations to ensure forward compatibility if in-circuit validation is added in the future.

Note

Currently this finding does not pose any threat but it should be kept in mind if "*Note secret not cryptographically bound to spending key*" was patched severity of this finding would increase and require patching.

Resolution

The development team has addressed this finding in commit `069c281`.

Duplicate incoming notes not validated

Severity Informational **Exploitability** None **Status** Resolved

Type Implementation **Impact** None

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`
- `zrchain/contracts/miden-circuits/hush.masm`

Description

The `generate_account_proof_internal` function does not validate that incoming notes are unique before generating a proof. While the chain-side validation in `x/hush` for unshields (code ref) and for transfers (code ref) correctly rejects transactions with duplicate incoming nullifiers using a `seenNullifiers` map, the client does not perform this check early. This means users who accidentally include duplicate incoming notes will waste computational resources generating an expensive STARK proof that will be rejected by the chain. Additionally, the circuit at `hush.masm` does not cryptographically enforce nullifier uniqueness, relying entirely on chain-side validation rather than circuit constraints.

Problem scenarios

A user's wallet experiences a synchronization bug or database corruption that causes the same incoming note to appear multiple times in their local state. When attempting to spend their funds, the wallet constructs a transaction including this note multiple times in the `incoming_notes_json` array. The `hush-wasm` client proceeds to generate a proof without detecting the duplication, which involves expensive cryptographic operations (STARK proving for the circuit). The proof generation succeeds because the circuit accumulates the duplicate amounts without checking uniqueness. However, when the transaction reaches the chain, the `x/hush` message handler detects the duplicate nullifiers and immediately rejects the transaction. The user has wasted time and computational resources generating a proof that was doomed to fail, and receives a cryptic error message without understanding what went wrong with their wallet state.

Recommendation

- In `generate_account_proof_internal`, after parsing the incoming notes array, add early validation to detect duplicates by checking that all commitment values are unique.
- While the chain already validates uniqueness, the circuit could enforce this cryptographically by adding pairwise nullifier comparison checks after the incoming notes loop at `hush.masm`. However, given the chain already validates this, the cost-benefit tradeoff may favor keeping this validation chain-side only.

Resolution

The development team has addressed this finding in commit `16480dc`.

Viewing key lifetime leak

Severity Informational **Exploitability** None **Status** Resolved

Type Design **Impact** None

Involved artifacts

- `docs/guides/hush-protocol-overview.md`

Description

If you share a viewing key once, the auditor can keep scanning your incoming notes forever. That's basically a permanent visibility grant, and if their data gets leaked you can be doxxed. There's also no clean way to prove if a leak happened.

Problem scenarios

- Auditor keeps old keys and monitors you indefinitely.
- Key leaks years later ⟶ retroactive exposure.
- No easy "proof of leak" or revocation other than moving funds.

Recommendation

Call this out as a privacy tradeoff in the docs. Suggest rotating to a new wallet for a clean break, and consider short-lived viewing keys or scoped keys (time/amount limits) if we want better privacy ergonomics.

Resolution

The development team acknowledged this finding and explicitly documented recommendations for secure usage of the viewing key (ref). Users are informed of the persistence nature of the key, and they may use it only specifically for compliance and auditing purposes. This design mirrors the well-established approach used by Zcash and other privacy-preserving systems.

Miscellaneous findings on hush-wasm

Severity Informational **Exploitability** None **Status** Reported

Type Implementation **Impact** None

Involved artifacts

- `zrchain/clients/hush-wasm/src/lib.rs`

High

- No overflow checks when computing balances (code ref).
- Avoid leaking private info via `Err` propagation. If possible, use `&'static str` for error strings.
- Avoid using `unwrap`s or `unwrap_or`s.
- Careful when using `getrandom` (code ref). We shouldn't rely on platform specific entropy generation. Better to rely on hash of `spending_key` (main source of entropy), transaction `sequence`, `CHAIN_ID` and domain tags to generate entropy.

Medium

- Careful about unbounded data. Validate all inputs for its length. We would suggest to implement everything in fixed length bytes and have glue code that deals with `&[u8]` and `Vec<u8>` for wasm boundary.
- `rand` label is too generic (code ref).
- There are still some `zerorize` calls missing.
  - We recommend using proper structs with `ZeroizeOnDrop` auto derivation to automatically `zerorize()` calling (ref).
  - `ZeroizeOnDrop` is recommended because of pitfalls like — returning early due to Error propagation (example) which forgets to call `zeroize()`.
- Check length of deserialised `note_secret` and `randomness` values for `balance_note` (code ref) and `incoming_notes` (code ref).

Low

- Fix all `cargo clippy` warnings.
- Use `let fixed_bytes = [u8: CONST_SIZE] = value.try_into()` for fixed length bytes.
- Use proper structs to deserialize JSON data using `serde_json::Value`.
- Use `&str` or enums for error types.
- Validate byte size directly on hex string, before `hex::decode` to avoid unnecessary decoding.
- Use `cfg!(feature = "prover")` instead of `#[cfg(feature = "prover")]` (code ref).
- Move loop-independent code blocks out of loop. For example, this code block doesn't depend of `seq` loop variable.
- Instead of `serde_value::Value[KEY].as_str()` use proper deserializable struct.
- To append a vector with empty/zero values, use `Vec::resize_with` (code ref 1, code ref 2).
- Careful when using `Vec::with_capacity`—it doesn't match in some cases (code ref 1, code ref 2, code ref 3).

- Use `as_chunks::<SIZE>()` over `chunks(SIZE)`. Careful when discarding `remainder` (code ref).
- Add mutation tests.
- Keep code linear. Instead of `if let Some(value) = wrapped_value {...}` try to use `let Some(value) = wrapped_value else { // other branch }` (same of `Ok(value)` too).
- Modularize the `lib.rs` file according to the usage and importance.
- Public function `pub fn rpo_hash` (code ref) should be calling the private function `fn rpo_hash_internal` (code ref) instead of duplicating the logic.
- Function `fn derive_viewing_keypair_internal` (code ref) is only used for tests, it should live in the tests module.
- This comment is misleading. It should say: "nullifier_key - 32-byte nullifier key (derived from spending_key, part of full viewing key)"

Resolution

Security-critical input validation and overflow issues were fixed. Zeroization of fields and secret leaks through logs are handled.

Miscellaneous findings in hush.masm

Severity Informational **Exploitability** None **Status** Resolved

Type Implementation **Impact** None

Unconstrained asset value

Asset is only validated to be non-zero (code ref). Asset values are used in commitment computation but never validated against a list of valid assets.

Recommendation: Sanitise asset value (code ref) before inserting in advice tape. Ideally do also range check in circuit.

Resolution: The development team addressed this recommendation in PR #913.

No validation of `incoming_count`

Count is loaded (code ref) but never used.

Recommendation: Consider removing it if not needed.

Resolution: The development team addressed this recommendation in PR #869.

Miscellaneous comments on x/hush

Severity Informational **Exploitability** None **Status** Reported

Type Implementation **Impact** None

Recommendations

High

- Implement the Cosmos SDK `ValidateBasic` to validate the `Unshield` and `ShieldTransfer` message fields.
- Use a ***cached*** *context* in `PreBlocker` and only write the changes (i.e., `writeCaches`) at the end to prevent future-code changes from introducing bugs (e.g., having a successful `AddCommitment` but then failing to add the voucher in the store in the processing of shield events).
- It is **not** clear why `MsgUnshield` and `MsgShieldedTransfer` have both the balance `nullifier` and the `has_balance_note` fields because just having `nullifier` could point to whether we have a balance note (e.g., if `nullifier` is not all zeros). Additionally, if there is no balance note, `balanceNullifier` is set to zeros, nevertheless `msg.Nullifier` that might not be all zeros is passed in `UnshieldRequestParams` and `ShieldedTransfer` that can lead to inconsistencies.
- Regarding migrations, since there is no upcoming migration, we did not audit migration code. However, looking at the v13 migration it is important to note that this migration does not interact with the Merkle tree, potentially leaving unused commitments in the tree. Also, clearing the processed shield events might lead to creating duplicate vouchers for the same shielding. Our recommendation is to be extremely cautious when migrating `x/hush` state to make sure it remains consistent with what is in the Merkle tree. Additionally during migrations, hard forks, etc. take extreme care when handling nullifiers to avoid reseting them.
- We understand the benefit of the *admin authority* account but it slightly contradicts a potential decentralization argument. We recommend making it clear on who owns this account (e.g., multi-sig, etc.). Additionally, in a disaster scenario, even if such an account exists, the account might not be extremely helpful if a lot of time is needed to fix a contract, upload a contract etc. It might also make sense to start with a smaller-fixed set of trusted validators that can help with stopping chain, fast migrations, etc.
- `msg.Nullifier` is passed in `UnshieldRequestParams` and `shieldedTransfer` even though a different `balanceNullifier` might have been used in case `!msg.HasBalanceNote` (e.g., see here) leading to inconsistencies. Make sure to use the same `balanceNullifier` across your `Unshield` and `ShieldedTransfer` methods.

Low

- Instead of doing `nullifierHex := string(nullifier)` use `hex.EncodeToString` as is done everywhere else where the `nullifierHex` is computed.
- Remove `status` from `VoucherStatus` since it is not used.
- Refrain from iterating over `map`s in `InitGenesis` (i.e., nullifiers) to prevent non-determinism.
- Unused code can be removed:
  - `realNullifierCount`;
  - `buildUnshieldPublicInputs`, `ComputeCommitment`, `ComputeNullifier`, and `RpoHash`.
  - `UnveilID` store in the `NullifierStore`

Miscellaneous findings on CW contracts

Severity Informational **Exploitability** None **Status** Resolved

Type Implementation **Impact** None

- While the unbounded root history storage does not pose a security risk or affect query performance, adding an admin-gated cleanup function to the `miden-merkle` contract (code ref) would enable storage optimization for long-running deployments. A simple `execute_prune_root_history()` function restricted to the admin could accept a `keep_recent` parameter to retain only the most recent N roots or roots newer than a specified block height, removing ancient historical data that is unlikely to be referenced. This would be purely optional maintenance, operators who prefer complete historical provenance can simply never invoke it, while those optimizing for storage costs can periodically prune entries beyond their compliance requirements. The function should enforce minimum retention (e.g., at least `HISTORY_SIZE` entries) to prevent accidental deletion of roots within the active validity window. This issue has been addressed by the development team in PR #869 and PR #901.
- The `outputs` parameter in the `sudo_verify()` function (code ref) uses an unnecessarily nested `Vec<Vec<u64>>` structure, despite the underlying `StackOutputs::new()` function requiring only a single `Vec<u64>` (code ref). The implementation silently uses only `outputs[0]` and discards any additional vectors, creating potential developer confusion during integration. While this API design flaw does not introduce security vulnerabilities, it represents a code quality issue that could lead to misuse or incorrect assumptions about the verification interface. A simplified `Vec<u64>` parameter would improve clarity and reduce the risk of integration errors. This issue has been addressed by the development team in PR #846.
- The `inputs` parameter in the `sudo_verify()` function (code ref) relies on Miden's `StackInputs::try_from_ints()`, which silently pads input vectors up to 16 elements with zeros. While this cannot be exploited maliciously, it can mask programming errors where the wrong number of public inputs is passed, leading to cryptic verification failures. Adding explicit input count validation at both the keeper layer (`len(publicInputs) != 8`) and contract layer would provide fail-fast behavior with clear error messages, making integration bugs immediately obvious. This issue has been addressed by the development team in PR #869.
- The `sudo_add_commitment()` function (code ref) lacks an explicit validation check to reject commitments that equal the `empty_leaf()` value (the RPO hash of [0,0,0,0]), which serves as the default placeholder for unoccupied leaf positions in the sparse Merkle tree. If such a collision were to occur, it would create semantic ambiguity where a stored commitment produces the same Merkle root as an unoccupied position. Adding a simple equality check `if commitment == word_to_bytes(&empty_leaf()) { return Err(...) }` would provide defense-in-depth at negligible gas cost, eliminating any theoretical semantic confusion, though it offers no practical security benefit, given the collision resistance guarantees, and should be considered a code clarity improvement rather than a vulnerability fix. This issue has been addressed by the development team in PR #869.

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
🟠 **High**	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
🟡 **Medium**	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
🟢 **Low**	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
🔵 **None**	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small

capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
🟠 **High**	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
🟡 **Medium**	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
🟢 **Low**	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
🔵 **None**	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
🔴 **Critical**	Halting of chain via a submission of a specially crafted transaction
🟠 **High**	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
🟡 **Medium**	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
🟢 **Low**	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🔵 **Informational**	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.