OtterSec

Zcash

Security Assessment

Prepared for
Valargroup

Prepared by
Otter Audits, LLC

Prepared on
July 18th, 2026

Himanshu Sheoran deuterium @osec.io
Sangsoo Kang sangsoo @osec.io
Quasar quasar @osec.io
Robert Chen r @osec.io

Table of Contents

1. Executive Summary

1.1. Overview

Zcash Foundation engaged OtterSec to conduct a security assessment of three core components of the Zcash ecosystem: Halo2, the zero-knowledge proving system (`halo2_proofs` and `halo2_gadgets`) that underpins Zcash's shielded proofs; Orchard, the Rust implementation of the Orchard shielded protocol, including note handling, actions, and bundle authorization; and Zebra, the Zcash Foundation's Rust full-node implementation, spanning its networking, synchronization, mempool, and state subsystems. Together these layers form the foundation of Zcash's privacy-preserving consensus — soundness of the circuits and cryptographic validation guarantees that shielded value cannot be forged, while the robustness of the node's network layer determines the chain's resilience against malicious peers.

This assessment was conducted between June 5th and July 16th, 2026. Our methodology consisted of manual source code review across all three codebases, with particular emphasis on circuit soundness in Halo2′s gadgets — underconstrained gates and exceptional cases in incomplete arithmetic — cryptographic validation in Orchard's bundle and signature handling, and network-layer resilience in Zebra, including peer reputation, inventory tracking, block and transaction propagation, and denial-of-service resistance in the syncer and mempool. For more information on our auditing methodology, refer to Appendix B.

1.2. Key Findings

We produced 26 findings throughout this audit engagement.

Among the most impactful results, we identified that Zebra can retain a stale `parent_error_map` entry after rejecting an invalid block body that reuses a legitimate NU5 header. Even if the valid same-header block is subsequently committed, the stale entry can cause its descendants to be rejected and stall block synchronization (● OS-ZCA-ADV-00). We also found that malformed `FindBlocks` responses reach an unrecoverable error path in Zebra's syncer, cancelling all in-flight downloads and forcing a restart delay, allowing a single peer to repeatedly stall block synchronization (● OS-ZCA-ADV-01). Additionally, in Halo2′s Sinsemilla gadget, hashing a zero-length message leaves the final output's y-coordinate underconstrained, permitting a malicious prover to substitute an arbitrary — even off-curve — point for the expected initial point (● OS-ZCA-ADV-14).

2. Scope

2.1. Halo 2

Halo2 is the zero-knowledge proving toolkit that powers Zcash's Orchard shielded protocol. It lets a prover convince a verifier that a computation was performed correctly over private inputs, revealing nothing beyond the public statement. Two properties distinguish it from earlier SNARKs: it requires **no trusted setup** as commitment parameters are derived transparently by hashing to the curve, and it supports **recursion** via proof accumulation over the Pallas/Vesta ("Pasta") curve cycle, the technique that gives Halo its name.

commit: 261faac

2.2. Orchard

Orchard is the shielded transfer protocol activated in Zcash Network Upgrade 5 (NU5, May 2022), the third generation of Zcash's private pools after Sprout and Sapling. The subject of this review is the `orchard` crate (v0.14.0), the reference Rust implementation: `no_std`-compatible, `unsafe`-free, and containing everything a wallet or node needs — key derivation and addresses, note commitment and nullifier logic, transaction construction, the Halo 2 circuit, proof and signature verification, and note encryption. Chain-level enforcement (the nullifier set, anchor history, block validity) deliberately lives in the consuming node, not in this crate; §4.2 maps that split. Where a transparent payment reveals sender, receiver, and amount, an Orchard payment reveals none of these, and its correctness rests on four interlocking ideas: **notes** as the unit of money, **commitments** that publish a note without revealing it, **nullifiers** that spend a note without identifying it, and **Actions** that make every transaction look alike.

commit: 82e0739

2.3. Zebra node

Zebra is the Zcash Foundation's independent implementation of a Zcash consensus node, written in Rust. The subject of this review is the `zebrad` workspace (v5.0.0) — twelve crates that connect to the peer-to-peer network, download and validate every block since genesis, maintain the canonical chain on disk, and answer zcashd-compatible JSON-RPC queries from wallets, miners, and indexers.

commit: f5feadb

3. Architecture

3.1. Halo 2

An overview of the Halo2 scopes architecture is shown in Figure 1. The halo2_proofs crate (v0.3.2) is the engine: the circuit-authoring API, the PLONK-style prover and verifier, polynomial commitments, the Fiat–Shamir transcript, and development tooling such as MockProver. On top of it, halo2_gadgets (v0.5.0) provides the reusable circuit components built for Orchard (elliptic-curve operations, Sinsemilla hashing with its Merkle gadget, Poseidon, and an unstable SHA-256 chip) with halo2_poseidon supplying the out-of-circuit Poseidon reference the gadget must agree with.

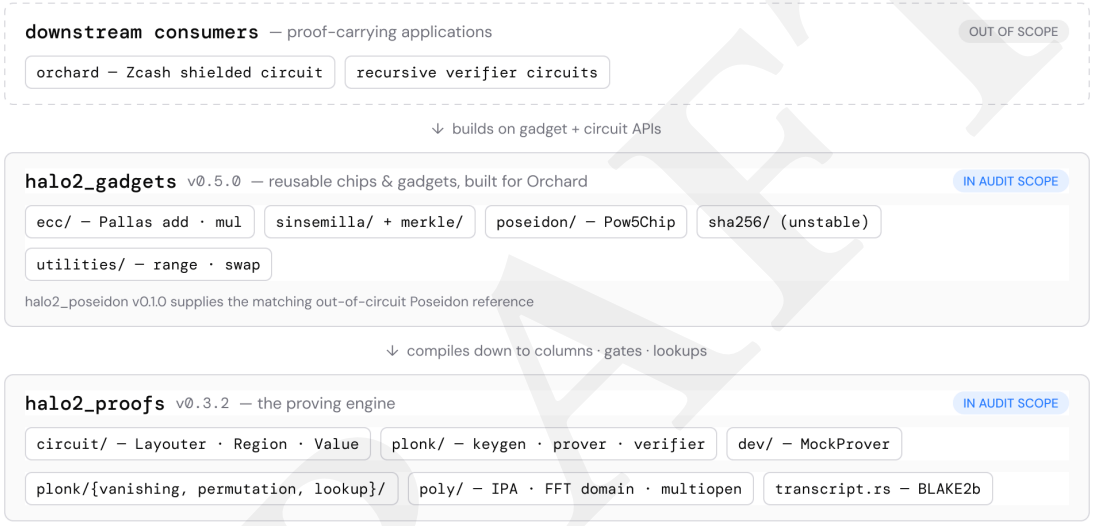


Figure 1: Crate and module architecture: downstream consumers build on halo2_gadgets chips, which compile down to the halo2_proofs engine.

The PLONKish circuit model

A Halo2 circuit is a rectangular table of finite-field elements with 2^k rows, depicted in Figure 2. Columns come in four types: **advice** columns hold the prover's private witness, **instance** columns hold public inputs, **fixed** columns hold constants baked in at key generation, and **selectors** are fixed on/off switches that activate gates on specific rows. Constraints take three forms. **Custom gates** are selector-gated polynomial identities that must vanish on every row — e.g. $s_{\text{mul}} \cdot (a \cdot b - c) = 0$ — and may reference neighboring rows through rotations. **Copy constraints** assert equality between arbitrary cells and are enforced globally by a permutation argument. **Lookups** assert that a tuple of expressions appears in a table, the workhorse behind range checks and hash S-boxes.

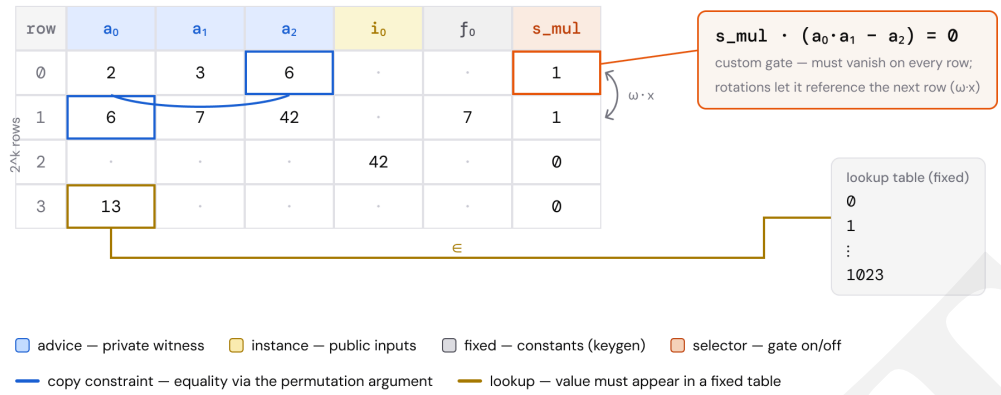


Figure 2: The PLONKish table: column types, a selector-gated custom gate with a rotation, a permutation-enforced copy constraint, and a lookup into a fixed table.

3.2. Orchard

Orchard is a redesign of Sapling (Figure 3). The proof system moved from Groth16 to **Halo 2**, eliminating the trusted-setup ceremony whose compromise would have allowed undetectable counterfeiting. Sapling's separate Spend and Output descriptions (which leak input/output counts) were unified into **Actions** carrying exactly one spend and one output, restoring the arity hiding Sprout had and Sapling lost. The toolbox was rebuilt around the **Pallas/Vesta** curve cycle with circuit-friendly primitives (Sinsemilla for commitments and Merkle hashing, Poseidon for the nullifier PRF), and the key hierarchy was simplified: no nsk, and every diversifier yields a valid address.

	Sapling (2018)	Orchard (NU5, 2022)
Proof system	Groth16 zk-SNARK — soundness rests on a multi-party trusted-setup ceremony	Halo 2 (inner-product argument) — no trusted setup; deterministic public parameters
Transaction shape	Separate Spend and Output descriptions — input / output counts are visible on-chain	Unified Actions, each spending one note and creating one note — arity is hidden
Curves	Jubjub, embedded over BLS12-381	Pallas / Vesta cycle ("Pasta") — protocol math on Pallas, proofs over Vesta
In-circuit primitives	Pedersen hashes and BLAKE2s — expensive inside the circuit	Sinsemilla (commitments, Merkle CRH) and Poseidon (nullifier PRF) — circuit-friendly
Keys & addresses	Extra nsk key component; some diversifiers are invalid	Simpler hierarchy (no nsk); every diversifier yields a valid address
Dummy padding	Arity hiding lost (Sprout had it via fixed 2-in / 2-out)	Regained: bundles are padded with dummy Actions to at least two

Figure 3: Protocol-level differences between Sapling and Orchard.

Core data model: notes, commitments, nullifiers

Orchard has no account balances and value exists only as **notes**. A note is private data: a recipient address, an unsigned 64-bit value, a uniqueness anchor rho, and a 32-byte rseed (ZIP 212) from which the commitment trapdoor rcm, nullifier randomizer psi, and encryption key esk are derived rather than stored. A wallet's balance is the sum of notes addressed to it whose nullifiers have not appeared on-chain; Figure 4 traces the three public artifacts a note produces over its life.

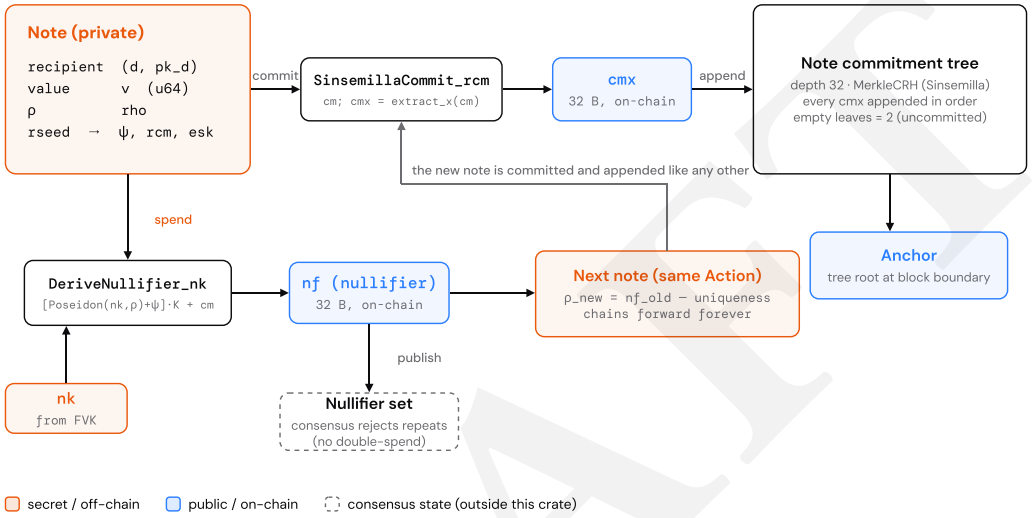


Figure 4: Note lifecycle. Creation publishes a commitment cmx; spending publishes a nullifier nf; the new note's rho is the spent note's nullifier, chaining uniqueness forward.

Commitments. The note never touches the chain; what is published is

$$\mathrm{cm} = \mathrm{SinsemillaCommit}_{\mathrm{rcm}}(g_d \parallel \mathrm{pk}_d \parallel v \parallel \rho \parallel \psi), \quad \mathrm{cmx} = \mathrm{extract}_x(\mathrm{cm}),$$

hiding yet binding: cmx (32 bytes) reveals nothing, but the creator cannot later claim different contents. Every cmx is appended, in consensus order, to a single depth-32 incremental Merkle tree hashed with the Sinsemilla-based MerkleCRH^Orchard; empty positions hold the sentinel 2, provably not the x-coordinate of any Pallas point. The root at a block boundary is an **anchor**; spending means proving in zero knowledge that a commitment sits under a published anchor — identifying the note only as "one of everything ever created" (Figure 5).

Nullifiers. Since commitments are never removed, double-spending is prevented by publishing

$$\mathrm{nf} = \mathrm{extract}_x\big(\big[\big(\mathrm{PRF}^{\mathrm{nf}}_{\mathrm{nk}}(\rho) + \psi\big) \bmod p\big] \cdot K^{\mathrm{Orchard}} + \mathrm{cm}\big),$$

with Poseidon as the PRF and nk the holder's nullifier-deriving key. nf is deterministic (consensus keeps a seen-set and rejects repeats) yet unlinkable to its commitment without nk, and uncomputable by the note's sender. Soundness requires every rho to be globally unique: a collision would let an attacker mint a note whose nullifier duplicates an existing one, freezing the victim's funds (the "Faerie Gold" attack). Orchard closes this structurally — a new note's rho **is** the nullifier of the note spent in the same Action, enforced in-circuit

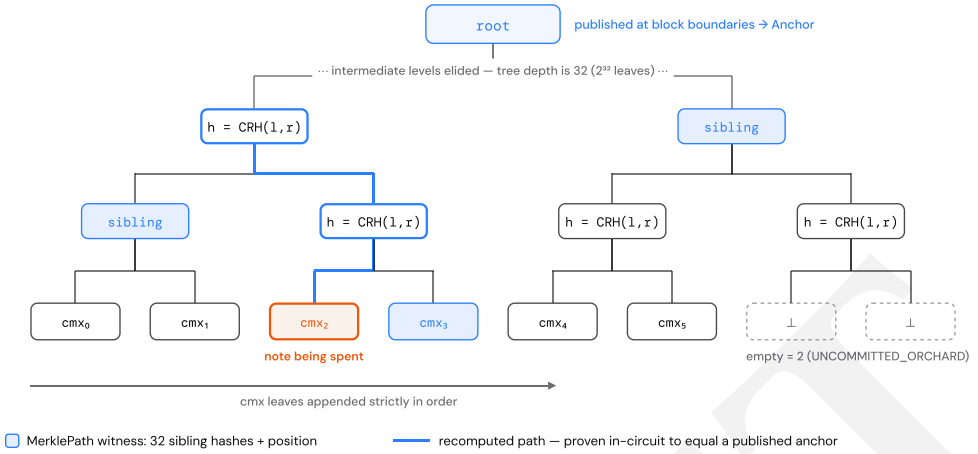


Figure 5: The note commitment tree. A spend witnesses one leaf with a MerklePath of 32 siblings; the recomputed root is constrained in-circuit to equal a public anchor.

(Circuit::from_action_context refuses to build a witness otherwise), so uniqueness chains forward indefinitely: the loop in Figure 4.

Keys and addresses

The key hierarchy separates spending from viewing from receiving, with strictly one-way derivations (Figure 6). From the 32-byte SpendingKey descend the spend-authorizing key ask (whose public half ak validates spends), the nullifier-deriving key nk, and rivk. The triple (ak, nk, rivk) is the 96-byte FullViewingKey — it sees all incoming and outgoing activity and derives everything below, but cannot spend. From it descend ivk = Commit^"ivk"_"rivk" (ak, nk), the diversifier key dk, and the outgoing viewing key ovk; the pair (dk, ivk) forms the IncomingViewingKey, sufficient only to detect and decrypt incoming payments, while ovk lets the sender's wallet recover what it sent.

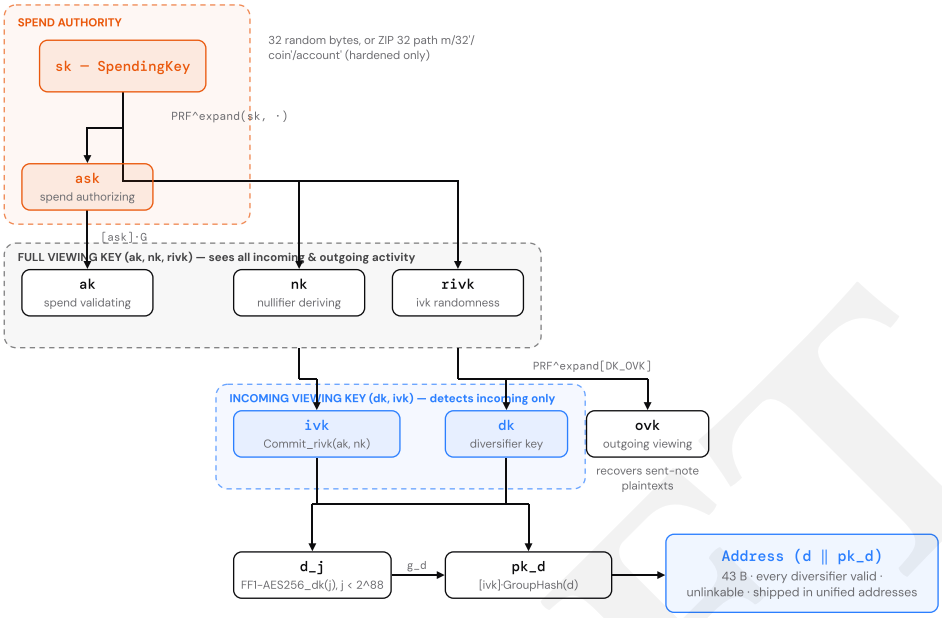


Figure 6: Key derivation (src/keys.rs) with capability zones. PRF^expand is BLAKE2b keyed on the parent secret; one-way arrows are cryptographic derivations.

Addresses are **diversified**: dk drives FF1-AES256 over a 2^{88} index space, index j yielding diversifier d_j and transmission key $\mathrm{pk}_d = [\mathrm{ivk}] \cdot \mathrm{GroupHash}(d_j)$. The 43-byte address $(d \parallel \mathrm{pk}_d)$ is unlinkable to its siblings, and every diversifier is valid. Each full viewing key also derives an **internal** scope (derive_internal) whose addresses absorb change without external exposure. HD derivation follows ZIP 32 at m/32'/coin'/account', hardened-only. Parse-time invariants: SpendingKey::from_bytes rejects keys whose ask is zero, and ak is sign-canonicalized so its encoding round-trips uniquely.

Actions and bundles: transaction anatomy

The on-chain unit is the **Action**: one spend and one output, welded together; its full public payload appears in Figure 7. Everything correlatable is absent — no addresses, no amounts, no spend-output linkage, and no way to tell a real half from a **dummy** (a zero-value note under a fresh random key, exempted from the Merkle check in-circuit). Padding and shuffling with dummies makes a 1-in-3-out payment indistinguishable from 3-in-1-out. A Bundle<T, V> adds bundle-wide data: flags (spend/output enable bits; the other six are consensus-required zero), the declared net value_balance, the shared anchor, and a typestate authorization payload. Amounts deliberately straddle three domains, namely: NoteValue (u64), ValueSum (i128, checked), and value_balance (signed 63-bit).

Value conservation without amounts. Each Action publishes $\mathrm{cv}^{\mathrm{net}} = \left(v_{\mathrm{spent}} - v_{\mathrm{created}}\right) \cdot V + \mathrm{rcv} \cdot R$. Commitments add homomorphically, so the verifier recomputes a **binding verification key** from public data:

$$\mathrm{bvk} = \sum_i \mathrm{cv}_i^{\mathrm{net}} - \mathrm{ValueCommit}_0(\mathrm{value\_balance}) = \left(\sum_i \mathrm{rcv}_i\right) \cdot R \quad \text{iff the declared balance is honest.}$$

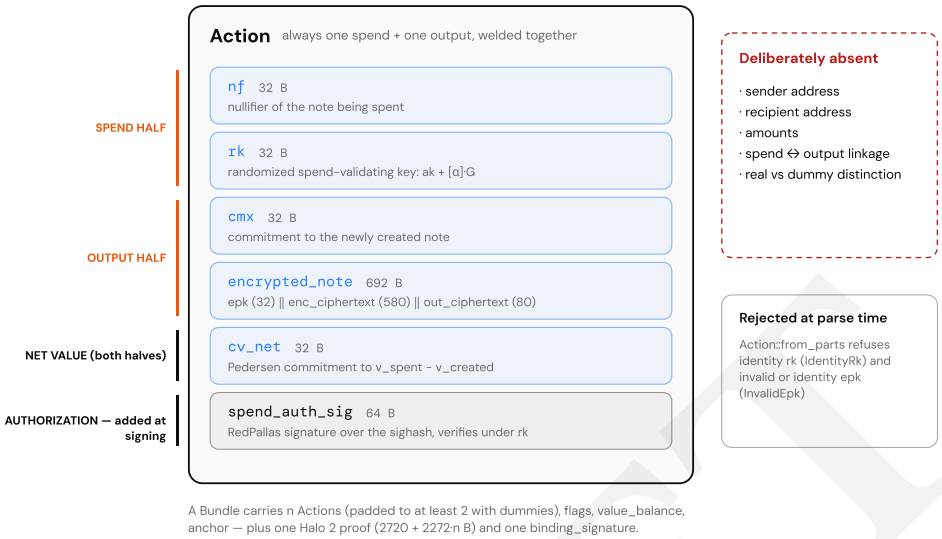


Figure 7: Anatomy of an Action. `Action::from_parts` rejects an identity `rk` and an invalid or identity `epk` at parse time, mirroring consensus rules.

Only then does the V term vanish, leaving a key whose secret $\text{bsk} = \sum \text{rcv}$ the builder knows — producing the bundle's RedPallas **binding signature** is itself the proof of value conservation (Figure 8). Spend authority is delegated per-Action: `ask` is randomized by a fresh α so the published $\text{rk} = \text{ak} + [\alpha] \cdot G$ is unlinkable across transactions, the circuit proving `rk` well-formed. All signatures cover the same ZIP 244 sighash.

The Halo 2 circuit

One small circuit (`src/circuit.rs`; $K = 11$, ten advice columns) proves every Action: nine public inputs against a fully private witness (Figure 9), assembled from `halo2_gadgets` chips (ECC, Sinsemilla, Poseidon) plus the custom `q_orchard` gate. Seven statements are proven at once: the old note's commitment opens; it sits under the public

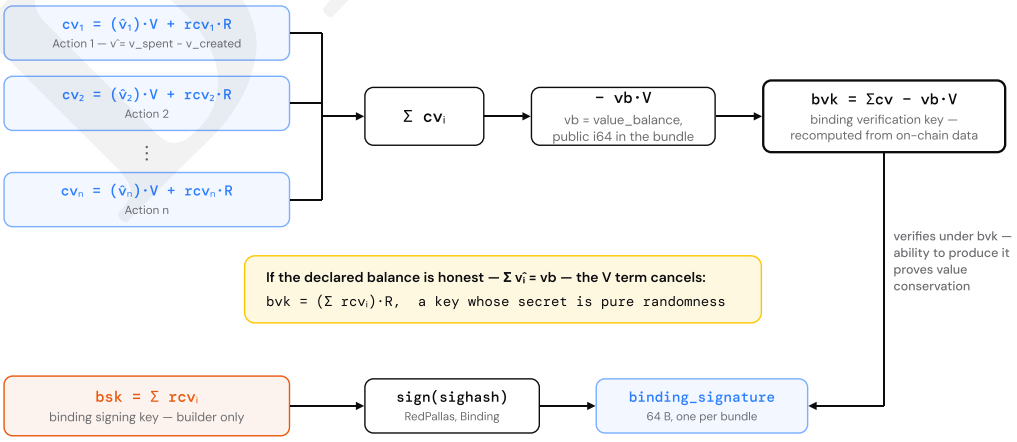


Figure 8: Homomorphic balance check. The verifier recomputes `bvk` from on-chain data; the binding signature verifies only if the value commitments net to the declared balance.

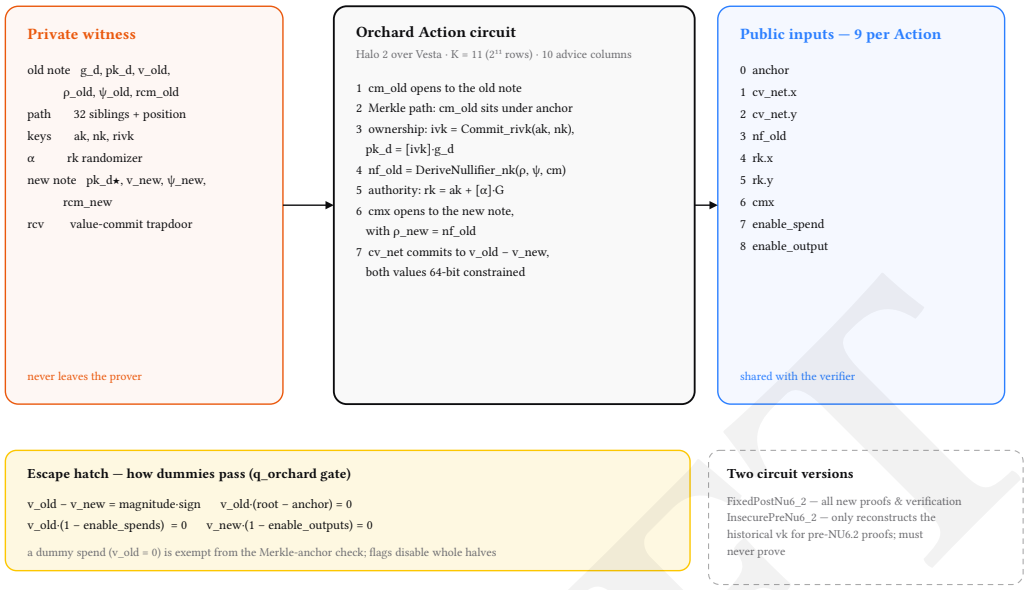


Figure 9: The Action circuit's statement. The q_orchard gate's second equation is the dummy escape hatch: a zero-value spend need not exist in the tree.

anchor; the prover owns it (ivk derivation, $\mathrm{pk}_d = [\mathrm{ivk}] \cdot g_d$); the nullifier is correctly derived; rk is ak randomized by α; the new commitment opens with $\rho_{\mathrm{new}} = \mathrm{nf}_{\mathrm{old}}$; and cv_net is correct with 64-bit range checks. The enable_spend / enable_output public flags mirror the bundle Flags and switch off whole halves.

3.3. Zebra Node

Every major component in Zebra, including the peer set, the verifiers, the state, the mempool is a *Tower service*, a uniform async request/response interface with an explicit readiness check. Generic middleware (timeouts, retries, bounded buffers, concurrency limits, load-shedding) composes around any service, and because every buffer is bounded, overload propagates *backward* as backpressure instead of accumulating in unbounded queues. This acts asa structural defense against resource exhaustion (4.3).

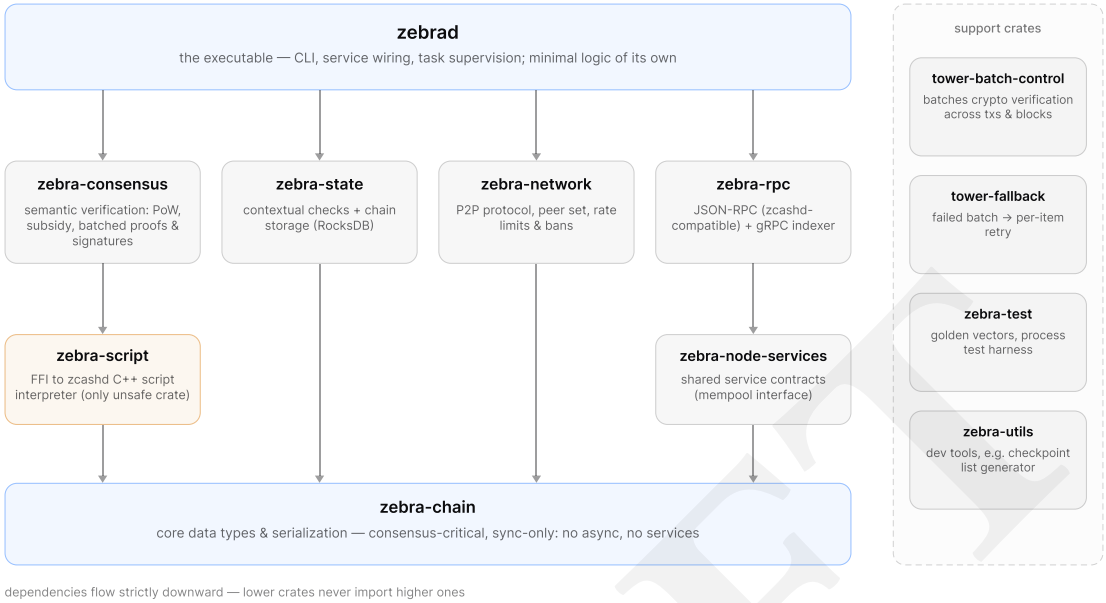


Figure 10: The zebrad workspace. Arrows are compile-time dependencies and flow only downward; zebra-script is the single crate permitted unsafe code.

The workspace enforces two structural rules, both visible in Figure 10: dependencies flow strictly downward, and zebra-chain(the crate every other crate speaks through) is *sync-only*: no async, no services, just data types and canonical (de)serialization. Structural validity is enforced there by the type system itself: Transaction is an enum whose variants physically contain only the fields legal for that version, and money is the checked Amount<Constraint> type whose arithmetic returns Result. As a result, silent-overflow inflation bugs are unrepresentable.

Validation Pipeline

The validation logic is split into three tiers: *structural* validity, enforced by zebra-chain types during parsing; *semantic* validity checkable per block in isolation by zebra-consensus; and *contextual* validity checked by zebra-state against a specific fork at commit time. Tiers 1 and 2 are independent per block, so hundreds of blocks verify concurrently during sync; only tier 3 serializes. Figure 11 traces one block's path.

The router's fork is the pipeline's key security/performance trade. Below the final hard-coded checkpoint, blocks are not re-verified cryptographically: they must hash-chain into checkpoint hashes baked into the source, so a fabricated history is as hard to inject as a source-code compromise. Above it, SemanticBlockVerifier checks Equihash proof-of-work, subsidy and funding streams, the merkle root, and sigop limits, and routes every transaction through script (via zebra-script), signature, proof, value-balance, expiry, and ZIP-317 fee checks. Every input source gets the same treatment, where a miner's submitblock enters the identical verifier stack as a peer's gossip.

State Management

Recent blocks cannot be treated as settled — the network occasionally reorganizes onto a heavier fork. Zebra models this with a two-tier state (Figure 12): a *non-finalized* set of in-memory Chain forks ordered by accumu-

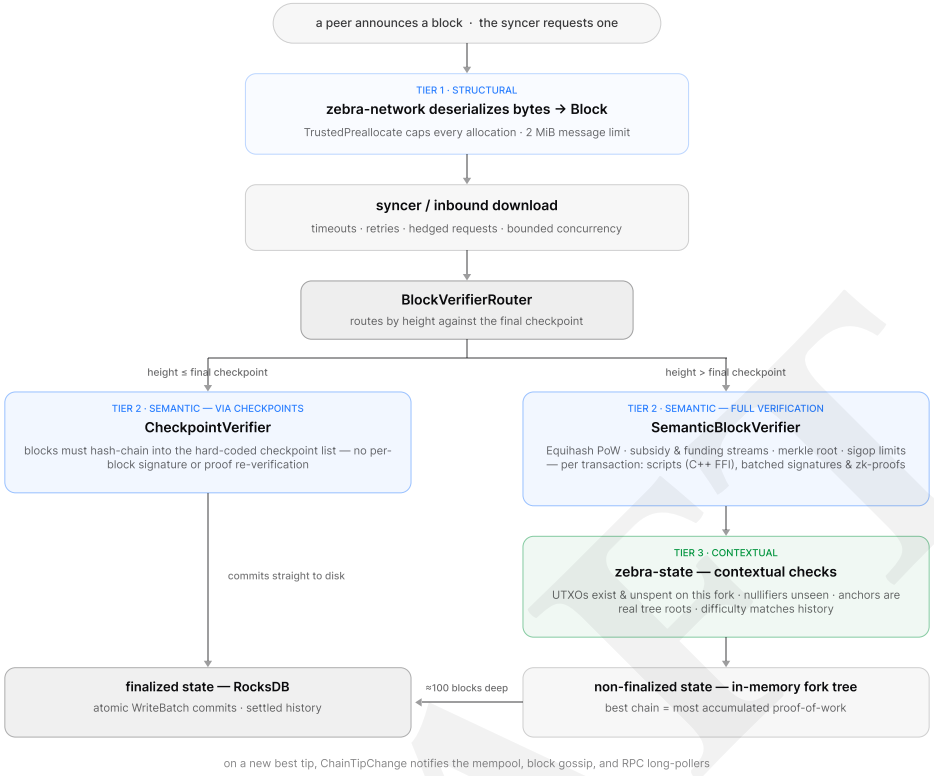


Figure 11: Life of a block. The router sends historical blocks down the checkpoint path straight to disk; post-checkpoint blocks receive full semantic verification, then fork-aware contextual checks.

lated proof-of-work, and a *finalized* RocksDB store for blocks at least `MAX_BLOCK_REORG_HEIGHT` (≈100) deep, past which reorgs are treated as impossible. Contextual checks — "is this UTXO unspent?", "has this nullifier appeared?" — are answered against the specific fork being extended, which is why tier-3 validation lives here.

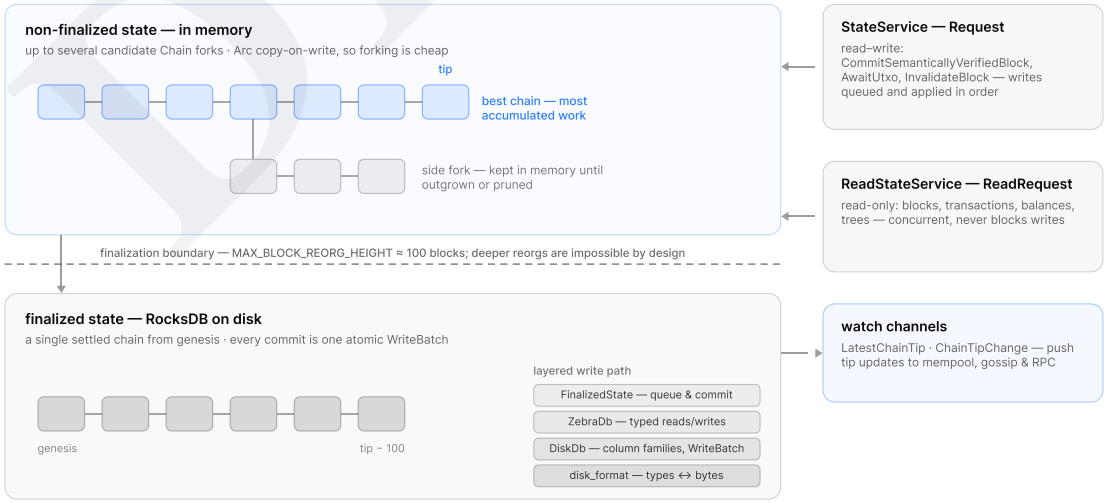


Figure 12: Two-tier state. Recent forks live in memory as copy-on-write `Chain` values; blocks ≈100 deep on the best chain are finalized into RocksDB, every commit one atomic `WriteBatch`.

Two storage properties are security-relevant. Writes descend through four layers (`FinalizedState` ⟶ `ZebraDb` ⟶ `DiskDb` ⟶ `disk_format`) and reach disk only as atomic `WriteBatch` operations, so a crash cannot half-commit a block; the on-disk format is versioned with in-place migrations. And the service surface is split: `StateService` handles the read-write `Request` API with writes queued in order, while the concurrent `ReadStateService` serves the read-only `ReadRequest` API used by RPC — readers can never block or reorder commits.

4. Security Boundaries and Trust Model

4.1. Halo 2

There are no privileged roles, ceremonies, or on-chain components in scope — parameters are publicly recomputable, both crates deny unsafe code, and soundness rests on the discrete-logarithm assumption over the Pasta curves together with Fiat–Shamir in the hardened BLAKE2b transcript. The boundaries that matter, mapped in Figure 13, are API boundaries between the untrusted prover and the verifier, and the central one is the **witness/constraint boundary**: everything the prover supplies — witness values, advice assignments, the proof bytes themselves — is attacker-controlled, and only properties enforced in-circuit by gates, lookups, and permutation constraints are guaranteed.

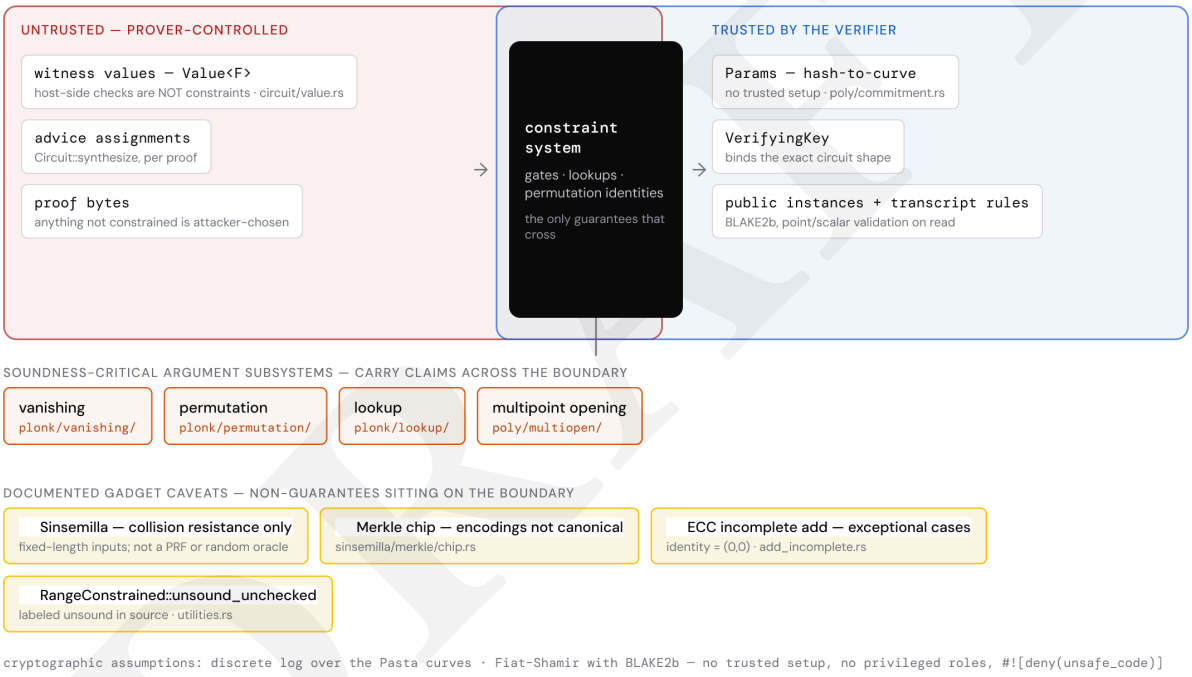


Figure 13: Trust boundaries: attacker-controlled prover inputs are separated from verifier-trusted material by the constraint system; the four argument subsystems carry claims across it, and gadget caveats sit on the boundary.

Claims cross that boundary through four soundness-critical argument subsystems — the vanishing, permutation, and lookup arguments and the multipoint opening. These, together with the gadget layer's documented sharp edges (Sinsemilla's fixed-length-only collision resistance and non-canonical Merkle encodings, ECC incomplete addition's excluded exceptional cases, and `RangeConstrained::unsound_unchecked`) form the focus of this assessment, including the Sinsemilla hash-to-point constraint gap and the incomplete-addition precondition fragility identified during the engagement.

4.2. Orchard

Figure 14 places every guarantee at the layer that enforces it. The distinctions that matter most: **the circuit does not prevent double-spends**, rejecting a repeated nullifier, like checking the anchor's presence in history, is consensus's job; **value conservation is a signature property, not a proof property**, only the binding signature ties per-Action `cv_net` to the declared `value_balance`; **the money cap is the consumer's job**, the circuit constrains 64 bits, but MAX_MONEY $\approx 2^{51}$ zatoshi must be applied by the node through the `V: TryFrom<i64>` bound, an easy responsibility to misplace; and **parse-time strictness is a consensus rule**, weakening the `from_parts` identity/length checks in a reimplementation reintroduces known malleability.

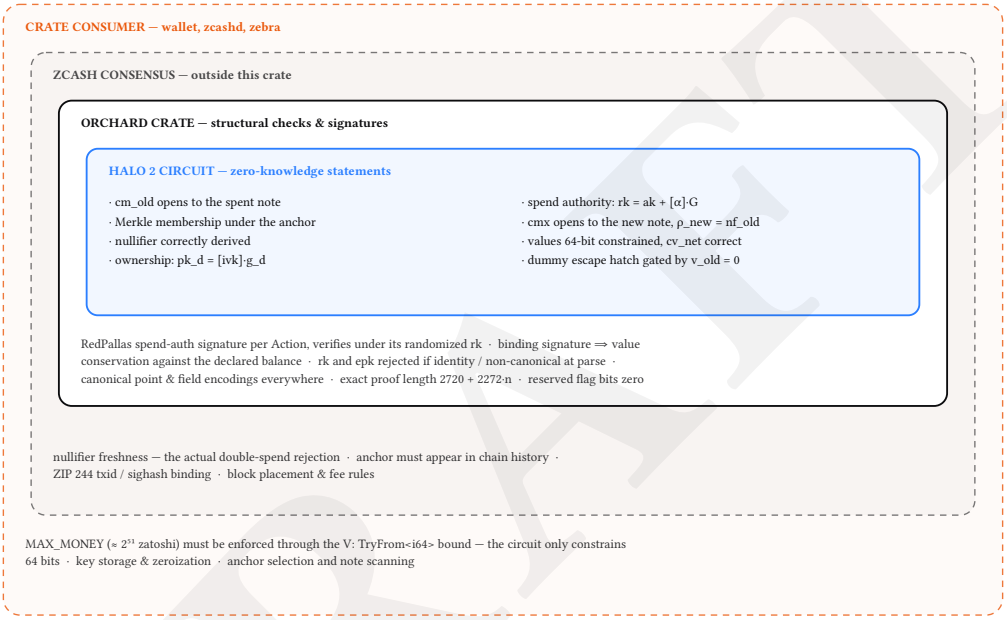


Figure 14: Enforcement boundaries. A guarantee missing from its zone — or silently assumed to live in a different one — is a protocol vulnerability.

4.3. Zebra Node

Zebra assumes every peer may lie, flood, stall, or send crafted bytes, with controls at each trust-zone crossing. The *P2P boundary* is the largest attack surface: deserialization is allocation-bounded by `TrustedPreallocate` (every length-prefixed collection is capped, derived from the 2 MiB message limit), misbehaving peers are scored and banned at 100 points, connections and downloads are rate-limited and capped per peer, overload sheds peers randomly so an attacker cannot force Zebra to drop its honest ones, and the address book distrusts self-reported addresses to resist eclipse attacks. Every external wait carries a timeout, every queue is bounded, the mempool admits only fully verified transactions and evicts randomly under its ZIP-401 cost limit so honest transactions cannot be predictably flushed, and checkpoints double as an anti-DoS wall, below them, fake chains die by cheap hash comparison.

5. Findings

Findings were rated using the criteria in Appendix A.

We split the findings into **advisories** and **suggestions**. Advisories have an immediate impact and should be remediated as soon as possible. Suggestions do not have an immediate impact but will aid in mitigating future vulnerabilities.

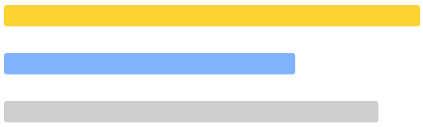
Items are tracked with a stable identifier: `OS-ZCA-ADV-*` for advisories and `OS-ZCA-SUG-*` for suggestions.

5.1. Severity by component

Component	Critical	High	Medium	Low	Info	Total
Zebra	—	—	10	6	6	22
Orchard	—	—	—	—	1	1
Halo2	—	—	—	1	2	3
Total	0	0	10	7	9	26

5.2. Findings summary

Severity	Count
Critical	0
High	0
Medium	10
Low	7
Informational	9



6. Advisories

Here, we present a technical analysis of the 17 vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

ID	Severity	Status	Description
OS-ZCA-ADV-00	Medium (Zebra)	✗ To-do	Header-Hash Cache Key Block Suppression
OS-ZCA-ADV-01	Medium (Zebra)	✗ To-do	Malformed FindBlocks Response Aborts Sync Attempt
OS-ZCA-ADV-02	Medium (Zebra)	✓ Resolved	Mempool Peer-Source Tracking Bypass
OS-ZCA-ADV-03	Medium (Zebra)	✗ To-do	Unsolicited notfound Inventory Eviction
OS-ZCA-ADV-04	Medium (Zebra)	✗ To-do	ZIP-401 Eviction Missing Invalidated Events
OS-ZCA-ADV-05	Medium (Zebra)	✗ To-do	Unreliable Locally Mined Block Advertisement
OS-ZCA-ADV-06	Medium (Zebra)	✗ To-do	Concurrent Best-Tip Checks Waste Shielded Verification Capacity
OS-ZCA-ADV-07	Medium (Zebra)	✗ To-do	Stale Sent Hashes in KnownBlock Lookup
OS-ZCA-ADV-08	Medium (Zebra)	✗ To-do	Root Invalidation Panic in State Writer
OS-ZCA-ADV-09	Medium (Zebra)	✗ To-do	Stale Finalized Block Panic in TrustedChainSync
OS-ZCA-ADV-10	Low (Zebra)	✗ To-do	Non-Ping Timeouts Leave Stalling Peers Connected
OS-ZCA-ADV-11	Low (Zebra)	✗ To-do	Gossiped Block Download Lacks Fallback Retry
OS-ZCA-ADV-12	Low (Zebra)	✗ To-do	Invalid-block Scoring Lost on Reset
OS-ZCA-ADV-13	Low (Zebra)	✗ To-do	Lookahead / Sync Progress Calculation Error
OS-ZCA-ADV-14	Low (Halo2)	✗ To-do	Sinsemilla Empty-Message Output Underconstrained
OS-ZCA-ADV-15	Low (Zebra)	✗ To-do	Far-Ahead Hash Rejection is Misattributed
OS-ZCA-ADV-16	Low (Zebra)	✗ To-do	Sendrawtransaction Lacks Dedicated RPC Work Limits

Header-Hash Cache Key Block Suppression

ID	Severity	Status	Found in
OS-ZCA-ADV-00	Medium	✗ To-do	bc2da19

Description

Zebra peers are susceptible to block-suppression attacks when block-processing state is keyed only by the block header hash. For NU5 blocks, the header's `hashBlockCommitments` value commits to the block's authorizing-data root. Therefore, modifying authorizing data while reusing the original header produces a candidate body that is inconsistent with that header, rather than a second valid block that is cryptographically indistinguishable at the header level.

The suppression issue arises when such a candidate body can populate or leave behind hash-only processing state before the mismatch with the header commitment is rejected, or when that state is not cleared after rejection. A later valid body that uses the same header can then be mistaken for a previously sent, rejected, or otherwise processed block. Affected patterns include stale entries in `parent_error_map`, non-drained `SentHashes` consulted through `KnownBlock`, and retained `SemanticallyVerifiedBlock` state.

Consequently, a malicious peer can first submit an invalid body that reuses a legitimate header and then cause Zebra to suppress the legitimate body or its descendants when a hash-only cache entry is consulted. The GHSA-4m69-67m6-prqp advisory demonstrates this general failure mode.

Remediation

Validate the candidate body's authorizing-data root against the header's `hashBlockCommitments` before updating cache state that can suppress later processing. Clear stale rejected or sent hash-only entries after failure or when a valid same-header body succeeds. If bodies must be distinguished before validation, use a full-body or aggregate authorizing-data fingerprint rather than a single transaction `auth_digest`.

Malformed FindBlocks Response Aborts Sync Attempt

ID	Severity	Status	Found in
OS-ZCA-ADV-01	Medium	✗ To-do	09964df

Description

This issue is a syncer response-validation variant of GHSA-h9hm-m2xj-4rq9, where malformed nonempty discovery responses are accepted far enough to disrupt the active sync attempt. During `obtain_tips`, Zebra sends `FindBlocks` fanout requests and processes returned block hashes. The code below shows that Zebra locates the first hash not already known to state and treats that position as the boundary between known chain data and hashes to download.

zebrad/src/components/sync.rs

```
#[instrument(skip(self))]
async fn obtain_tips(&mut self) -> Result<IndexSet<block::Hash>, Report> {
 [...]
   Ok(zn::Response::BlockHashes(hashes)) => {
     [...]
     let mut first_unknown = None;
     for (i, &hash) in hashes.iter().enumerate() {
       if !self.state_contains(hash).await? {
         first_unknown = Some(i);
         break;
       }
     }
     [...]
   }
 [...]
}
```

First unknown hash boundary

The issue is that the response is not validated before the suffix beginning at the first unknown hash is treated as usable discovery material. For an attacker-controlled sequence such as `[U,K,R]`, where `U` is unknown, `K` is already known, and `R` is random or later discarded, the suffix beginning at `U` can still be added to the download set, as shown below.

zebrad/src/components/sync.rs

```
#[instrument(skip(self))]
async fn obtain_tips(&mut self) -> Result<IndexSet<block::Hash>, Report> {
  [...]
    Ok(zn::Response::BlockHashes(hashes)) => {
      [...]
      let prev_download_len = download_set.len();
      download_set.extend(unknown_hashes);
      let new_download_len = download_set.len();
```

```
            let new_hashes = new_download_len - prev_download_len;
            [...]
        }
    [...]
}
```

Unvalidated suffix added to the download set

After collecting peer responses, `obtain_tips` performs another state-membership check over the queued hashes. In the malformed sequence, this later check reaches the embedded known hash `K`, and the path returns an error instead of discarding the malformed peer response and continuing with other fanout results.

zebrad/src/components/sync.rs

```
#[instrument(skip(self))]
async fn obtain_tips(&mut self) -> Result<IndexSet<block::Hash>, Report> {
    [...]
    // Check that the new tips we got are actually unknown.
    for hash in &download_set {
        debug!(?hash, "checking if state contains hash");
        if self.state_contains(*hash).await? {
            return Err(eyre!("queued download of hash behind our chain tip"));
        }
    }
    [...]
}
```

Known queued hash converts response into a sync error

That error propagates out of the sync attempt to the outer sync loop. The loop cancels current downloads on any `try_to_sync` error and then waits for the non-regtest `SYNC_RESTART_DELAY`, resulting in the syncer trying again after 67 seconds.

zebrad/src/components/sync.rs

```
pub async fn sync(mut self) -> Result<(), Report> {
    // We can't download the genesis block using our normal algorithm,
    // due to protocol limitations
    self.request_genesis().await?;

    loop {
        if self.try_to_sync().await.is_err() {
            self.downloads.cancel_all();
        }
        self.update_metrics();
        let restart_delay = if self.is_regtest {
            REGTEST_SYNC_RESTART_DELAY
        } else {
            SYNC_RESTART_DELAY
```

```
        };
        [...]
        sleep(restart_delay).await;
    }
}
```

Sync error cancels downloads before restart delay

Since this behavior is not punished, a malformed nonempty discovery response can repeatedly turn discovery into a sync error, cancel active downloads, and force the syncer back through the restart delay when the malicious response is selected. This is not the only response shape that can trigger such issues. Several non-empty responses, such as `[R]`, may also result in no effective action.

Remediation

Validate discovery responses before mutating `download_set`: after the first unknown hash is found, reject that peer response if any subsequent hash is already known to state or otherwise does not match the expected discovery sequence. A malformed nonempty `FindBlocks` response should be discarded and attributed through an appropriate stall or misbehavior path, rather than returned as an unrecoverable sync error that cancels all downloads.

Mempool Peer-Source Tracking Bypass

ID	Severity	Status	Found in
OS-ZCA-ADV-02	Medium	✓ Resolved	f5feadb

Description

Transactions entering the mempool through `Request::Queue` are recorded with `source` set to `None`. For peer-originated transaction load, this removes the attribution needed to charge the work to the sending peer, so the admission path is not subject to the same per-peer accounting as paths that preserve a peer source.

Because the per-peer limit depends on source attribution, transactions queued this way can consume the global `MAX_INBOUND_CONCURRENCY` budget without counting against `MAX_INBOUND_CONCURRENCY_PER_PEER`. This creates a resource-accounting bypass: one origin can occupy shared mempool transaction-handling capacity while avoiding the per-peer cap intended to keep that capacity available to other peers. Note that this issue is a variant of advisories GHSA-4fc2-h7jh-287c and GHSA-h9hm-m2xj-4rq9.

Remediation

Preserve source attribution for peer-originated transactions that enter through `Request::Queue`, or apply an equivalent per-origin budget when the source cannot be represented directly.

Patch

Resolved in 6c14a396.

Unsolicited notfound Inventory Eviction

ID	Severity	Status	Found in
OS-ZCA-ADV-03	Medium	✗ To-do	aae2059

Description

Zebra records peer inventory status from inbound P2P messages in `register_inventory_status`. The excerpt below shows that function matching an inbound `Message::NotFound(missing)` from a connected peer that has a transient address. It then filters the provided inventory to block hashes and unmined transaction IDs, creates a Missing inventory change, and sends it to the inventory collector. This path is not tied to the request handler state, so a `notfound` message can be recorded even when the peer was not responding to an outstanding inventory request.

zebra-network/src/peer/handshake.rs

```
pub(crate) async fn register_inventory_status(
    msg: Result<Message, SerializationError>,
    connected_addr: ConnectedAddr,
    inv_collector: broadcast::Sender<InventoryChange>,
) -> Result<Message, SerializationError> {
    match (&msg, connected_addr.get_transient_addr()) {
        [...]
        (Ok(Message::NotFound(missing)), Some(transient_addr)) => {
        // Ignore Errors and the unsupported FilteredBlock type
        let missing = missing.iter().filter(|missing| {
            missing.unmined_tx_id().is_some() || missing.block_hash().is_some()
        });
        [...]
        if let Some(change) = InventoryChange::new_missing_multi(missing, transient_addr) {
            let _ = inv_collector.send(change);
        }
    }[...]
}
```

NotFound match arm in inventory registration

`InventoryRegistry::register` then stores the resulting status in the shared `current` inventory map. As shown below, the same insertion path is used for all inventory status changes, and the map is trimmed by removing the oldest hash entry once `current` exceeds `MAX_INV_PER_MAP`. Thus, `Available` and `Missing` entries compete for the shared bounded inventory-hash budget as they are stored under the same 1,000-hash cap.

zebra-network/src/peer_set/inventory_registry.rs

```rust
fn register(&mut self, change: InventoryChange) {
  [...]
  for inv in invs {
    [...]
    // # Security
    //
    // Limit the number of stored inventory hashes, removing the oldest entries,
    // because newer entries are likely to be more relevant.
    //
    // TODO: do random or weighted random eviction instead?
    if self.current.len() > MAX_INV_PER_MAP {
      // Performance: `MAX_INV_PER_MAP` is small, so O(n) performance is acceptable.
      self.current.shift_remove_index(0);
    }
  }
}
```

Shared inventory cap evicts oldest hash entries

As a result, unsolicited `Missing` entries will occupy the same bounded `current` map as `Available` advertisements, enabling a malicious peer to repeatedly send `notfound` messages containing random block or transaction hashes and cause fresh `Missing` records to displace older entries. This can evict honest peers' recent `Available` advertisements and reduce the accuracy of Zebra's inventory-based request routing, which prefers peers that recently advertised a requested hash before falling back to peers not marked missing.

Remediation

Record `notfound` inventory as `Missing` only from the request/response path after confirming that the hashes correspond to an outstanding request for the same peer. Additionally, split `Available` and `Missing` inventory accounting, or enforce separate caps, so `Missing` entries cannot consume the capacity reserved for recent `Available` advertisements.

ZIP-401 Eviction Missing Invalidated Events

ID	Severity	Status	Found in
OS-ZCA-ADV-04	Medium	✗ To-do	fcf5e6a

Description

Zebra publishes mempool changes through `MempoolChange` events, but the ZIP-401 eviction path can remove verified transactions without producing a corresponding `Invalidated` event. `Storage::insert()` enforces the mempool cost limit after inserting a verified transaction, while `Mempool::poll_ready()` builds broadcast changes from the insertion result and other explicit invalidation paths. Because the insertion API does not expose the transaction IDs removed by ZIP-401 eviction, those removals are not available to the event-building logic.

The excerpt below shows the root of the information loss in `VerifiedSet::evict_one()`: ZIP-401 eviction removes the randomly selected transaction and any transactions that depend on it, but the function returns only a single removed transaction to its caller.

zebrad/src/components/mempool/storage/verified_set.rs

```rust
pub fn evict_one(&mut self) -> Option<VerifiedUnminedTx> {
  use rand::distributions::{Distribution, WeightedIndex};
  use rand::prelude::thread_rng;

  let (keys, weights): (Vec<transaction::Hash>, Vec<u64>) = self
    .transactions
    .iter()
    .map(|(&tx_id, tx)| (tx_id, tx.eviction_weight()))
    .unzip();

  let dist = WeightedIndex::new(weights).expect(
    "there is at least one weight, all weights are non-negative, and the total is positive",
  );

  let key_to_remove = keys
    .get(dist.sample(&mut thread_rng()))
    .expect("should have a key at every index in the distribution");

  // Removes the randomly selected transaction and all of its dependents from the set,
  // then returns just the randomly selected transaction
  self.remove(key_to_remove).pop()
}
```

ZIP-401 eviction returns only the selected victim

This means existing mempool transactions evicted to make room for a new transaction can disappear from storage while indexer gRPC subscribers receive no `Invalidated` notification for them. In dependent-transaction cases, a newly inserted child can be removed as a dependent of an evicted parent, while `Storage::insert()` still reports success for the child because it only compares the returned victim with the inserted transaction ID. `Mempool::poll_ready()` can then emit `Added` for a transaction that has already been removed, allowing downstream subscribers to retain stale pending transaction state.

Remediation

Return the full set of transaction IDs removed by ZIP-401 eviction from the storage insertion path, including both the selected victim and all dependent transactions. `Storage::insert()` should report whether the newly inserted transaction remains in the verified set, and `Mempool::poll_ready()` should remove any evicted IDs from the pending `Added` set and include them in `MempoolChange::invalidated`.

Unreliable Locally Mined Block Advertisement

ID	Severity	Status	Found in
OS-ZCA-ADV-05	Medium	✗ To-do	d9d29b6

Description

Zebra's block gossip task treats locally mined blocks differently from ordinary committed-tip gossip: when a mined block notification wins the selection, the task builds an `AdvertiseBlockToAll` request and sends it through the peer-set service. The excerpt below shows that this same path is wrapped in the syncer's tips-response timeout, so the mined-block advertisement is constrained by a short network-service timeout before the rest of the peer-set behavior is considered.

zebrad/src/components/sync/gossip.rs

```
pub async fn gossip_best_tip_block_hashes<ZN>([...]) -> Result<(), BlockGossipError> where ZN: Service<zn::Request, Response = zn::Response, Error = BoxError> + Send + Clone + 'static, ZN::Future: Send, {
    [...]
    // so broadcasts don't delay the syncer too long
    let mut broadcast_network = Timeout::new(broadcast_network, TIPS_RESPONSE_TIMEOUT);
    [...]
}
```

For mined block submissions, `gossip_best_tip_block_hashes` selects the mined-block receiver branch and constructs an all-peers advertisement before spawning the returned broadcast future. As shown in the excerpt below, the task does not await the broadcast future in-line, because it is intended not to block sync progress while peers are unready.

zebrad/src/components/sync/gossip.rs

```
pub async fn gossip_best_tip_block_hashes<ZN>([...]) -> Result<(), BlockGossipError>
  [...]
    info!(?height, ?request, log_msg);
    let broadcast_fut = broadcast_network
        .ready()
        .await
        .map_err(PeerSetReadiness)?
        .call(request);

    // Await the broadcast future in a spawned task to avoid waiting on
    // `AdvertiseBlockToAll` requests when there are unready peers.
    // Broadcast requests don't return errors, and we'd just want to ignore them anyway.
    tokio::spawn(broadcast_fut);
    [...]
    // Mark the last change hash of `chain_state` as the last block submission hash to avoid
    // advertising a block hash to some peers twice.
    if is_block_submission
        && mined_block_receiver
```

```
            .as_ref()
            .is_some_and(|rx| rx.is_empty())
        && chain_state.latest_chain_tip().best_tip_hash() == Some(hash)
    {
        chain_state.mark_last_change_hash(hash);
    }
  [...]
}
```

The task then immediately suppresses the normal committed-tip fallback when the mined block is still the current best tip and no newer mined-block notification is pending. This is implemented by marking the best-tip hash as already seen after spawning the advertisement future, not after confirming that the advertisement was dispatched by the peer set.

The reliability issue comes from the service boundary between `zebrad` and `zebra-network`: the peer set returned by network initialization is a buffered Tower service. As shown below, the peer set being wrapped in a bounded buffer, meaning a call accepted by `zebrad` can still be waiting behind that buffer before the underlying peer-set dispatch logic runs.

zebra-network/src/peer_set/initialize.rs

```
pub async fn init<S, C>([...]) -> ( Buffer<BoxService<Request, Response, BoxError>, Request>,
Arc<std::sync::Mutex<AddressBook>>, mpsc::Sender<(PeerSocketAddr, u32)>, ) where S: Service<Request, Response = Response,
Error = BoxError> + Clone + Send + Sync + 'static, S::Future: Send + 'static, C: ChainTip + Clone + Send + Sync + 'static, {
  [...]
  );
  let peer_set = Buffer::new(BoxService::new(peer_set), constants::PEERSET_BUFFER_SIZE);
  [...]
}
```

Only once the request reaches the underlying `PeerSet::call` does `AdvertiseBlockToAll` route to the all-ready-peers broadcast path, as shown in the excerpt below. If the buffered future times out or is dropped before this dispatch point, the local mined block may not be advertised through that request, while the gossip task has already marked the tip hash as seen and suppressed the committed-tip reannouncement path. This can make local mined block propagation unreliable rather than guaranteeing at least one completed advertisement or retry opportunity.

zebra-network/src/peer_set/set.rs

```rust
fn call(&mut self, req: Request) -> Self::Future {
    let fut = match req {
        [...]
        // Broadcast advertisements to lots of peers
        Request::AdvertiseTransactionIds(_, _) => self.route_broadcast(req),
        Request::AdvertiseBlock(_, _) => self.route_broadcast(req),
        Request::AdvertiseBlockToAll(_) => self.broadcast_all(req),
        [...]
    };
    self.update_metrics();
    fut
}
```

Remediation

For mined-block submissions, move `chain_state.mark_last_change_hash(hash)` behind a positive dispatch signal. At minimum, keep the mined block hash eligible for later committed-tip gossip unless the `AdvertiseBlockToAll` future completes successfully.

Concurrent Best-Tip Checks Waste Shielded Verification Capacity

ID	Severity	Status	Found in
OS-ZCA-ADV-06	Medium	✗ To-do	57ddc31

Description

Zebra's prior advisory in GHSA-84j3-rw4c-gqmj describes the peer-driven availability class relevant here: default peer-to-peer networking exposes transaction validation to connected peers, and the documented impact is degraded node responsiveness under sustained load rather than a crash, consensus fault, or loss of funds.

In `Verifier::call`, the transaction verifier first creates version-specific async checks through `dispatch_version_verification`. The surrounding implementation shows those checks can include transparent input verification and shielded Sapling or Orchard verification for applicable transaction versions. Only after that does the mempool branch append the `CheckBestChainTipNullifiersAndAnchors` state query.

zebra-consensus/src/transaction.rs

```
fn call(&mut self, req: Request) -> Self::Future {
  [...]
  async move {
    [...]
    // Select version-specific async verification pipeline
    let mut async_checks = Self::dispatch_version_verification(
      tx.as_ref(),
      nu,
      script_verifier,
      cached_ffi_transaction.clone()
    )?;
    if let Some(unmined_tx) = req.mempool_transaction() {
      let check_anchors_and_revealed_nullifiers_query = state
        .clone()
        .oneshot(zs::Request::CheckBestChainTipNullifiersAndAnchors(
          unmined_tx,
        ))
        .map(|res| {
          assert!(
            res? == zs::Response::ValidBestChainTipNullifiersAndAnchors,
            "unexpected response to CheckBestChainTipNullifiersAndAnchors request"
          );
          Ok(())
        }
        );
      async_checks.push(check_anchors_and_revealed_nullifiers_query);
    }
    tracing::trace!(?tx_id, "awaiting async checks...");
    async_checks.check().await?;
    [...]
```

```
    }
    [...]
}
```

Best-tip check queued with async verification

The failure mode is cancellation asymmetry. `AsyncChecks::check` returns as soon as one check fails and drops the remaining futures, so a transaction whose anchors or revealed nullifiers are invalid against the best chain can be rejected quickly at the async layer. However, proof-verifier work launched through Rayon is not canceled merely because the future waiting for its oneshot receiver has been dropped. As shown in `spawn_fifo`, the Rayon closure continues until it computes and attempts to send its result.

zebra-consensus/src/primitives.rs

```rust
pub async fn spawn_fifo<T: 'static + Send, F: 'static + FnOnce() -> T + Send>(
    f: F,
) -> Result<T, RecvError> {
    // Rayon doesn't have a spawn function that returns a value,
    // so we use a oneshot channel instead.
    let (rsp_tx, rsp_rx) = tokio::sync::oneshot::channel();

    rayon::spawn_fifo(move || {
        let _ = rsp_tx.send(f());
    });

    rsp_rx.await
}
```

Rayon task outlives dropped waiter

The root cause is that the cheap contextual rejection is scheduled in parallel with, rather than before, expensive shielded verification for mempool transactions. A remote peer that can submit transactions to the mempool can therefore cause transactions that state would reject on anchors or nullifiers to occupy shared Rayon-backed verification capacity until those jobs finish. As a result, mempool and block validation may become slower under sustained peer-driven load.

Remediation

For mempool transactions, run the best-chain-tip anchor and revealed-nullifier check before dispatching version-specific shielded proof or Rayon-backed primitive verification work. The verifier should await `CheckBestChainTipNullifiersAndAnchors` and only enqueue expensive shielded verification jobs after the state service returns the valid best-chain response.

Stale Sent Hashes in KnownBlock Lookup

ID	Severity	Status	Found in
OS-ZCA-ADV-07	Medium	✗ To-do	57ddc31

Description

The GHSA-4m69-67m6-prqp security advisory provides the context for this issue, where rejected same-hash non-finalized blocks could leave stale SentHashes state and cause later deliveries to be treated as duplicates. The fix for this advisory added a rejected-hash channel and a drain helper that removes rejected non-finalized block hashes from non_finalized_block_write_sent_hashes. That drain is performed in the CommitSemanticallyVerifiedBlock path, inside queue_and_commit_to_non_finalized_state, immediately before the path checks whether the semantically verified hash is already present in SentHashes. The below snippet shows the commit-path cleanup before the duplicate check:

zebra-state/src/service.rs

```
fn queue_and_commit_to_non_finalized_state(
    [...]
    semantically_verified: SemanticallyVerifiedBlock,
) -> oneshot::Receiver<Result<block::Hash, CommitSemanticallyVerifiedError>> {
    tracing::debug!(block = %semantically_verified.block, "queueing block for contextual verification");
    let parent_hash = semantically_verified.block.header.previous_block_hash;

    // Drop hashes of any blocks the write task has rejected before checking
    // the SentHashes membership below. Without this, a rejected same-hash
    // block would lock out a later honest re-delivery of a block at the
    // same hash as a false "duplicate".
    self.drain_non_finalized_rejected_hashes();

    if self
        .non_finalized_block_write_sent_hashes
        .contains(&semantically_verified.hash)
    {{...}}
    [...]
}
```

Rejected-hash cleanup in the commit path

Request::KnownBlock uses known_sent_hash to read non_finalized_block_write_sent_hashes and can return KnownBlock::WriteChannel when the hash is still present, without first draining the rejected-hash receiver. That leaves this lookup path able to observe stale SentHashes state even though the commit path now cleans it before its own membership check.

As a result, a re-delivered block whose earlier same-hash body was rejected may still be reported as present in the block write channel before the commit path gets a chance to run the cleanup. This preserves a stale-hash window in Request::KnownBlock and can cause an honest re-delivery or validation attempt to be short-circuited as already known until another path drains the rejection channel.

Remediation

Drain `non_finalized_rejected_receiver` before any `SentHashes` membership check used to answer `Request::KnownBlock`.

Root Invalidation Panic in State Writer

ID	Severity	Status	Found in
OS-ZCA-ADV-08	Medium	✗ To-do	90ea4a9

Description

The `invalidateblock` and `reconsiderblock` path can reach a state that the write worker still treats as impossible. When `NonFinalizedState::invalidate_block` is asked to invalidate the non-finalized root, the surrounding implementation removes the entire chain from the chain set, so the non-finalized state may no longer have a best chain. `WriteBlockWorkerTask::run` invalidates and reconsiders messages being processed as non-commit messages, followed by an unconditional call to update the latest-chain channels before the loop continues.

zebra-state/src/service/write.rs

```
pub fn run(mut self) {
  [...]
  while let Some(msg) = non_finalized_block_write_receiver.blocking_recv() {
    let queued_child_and_rsp_tx = match msg {
      NonFinalizedWriteMessage::Commit(queued_child) => Some(queued_child),
      NonFinalizedWriteMessage::Invalidate { hash, rsp_tx } => {
        tracing::info!(?hash, "invalidating a block in the non-finalized state");
        let _ = rsp_tx.send(non_finalized_state.invalidate_block(hash));
        None
      }
      NonFinalizedWriteMessage::Reconsider { hash, rsp_tx } => {
        tracing::info!(?hash, "reconsidering a block in the non-finalized state");
        let _ = rsp_tx.send(non_finalized_state.reconsider_block(hash, &finalized_state.db));
        None
      }
    };
    let Some((queued_child, rsp_tx)) = queued_child_and_rsp_tx else {
      update_latest_chain_channels(
        non_finalized_state,
        chain_tip_sender,
        non_finalized_state_sender,
        backup_dir_path.as_deref(),
      );
      continue;
    };
    [...]
  }
}
```

Unconditional channel update after invalidate or reconsider

`update_latest_chain_channels` still assumes it is only called after at least one non-finalized block has been committed. As shown in the helper below, it reads the current best chain and tip through panicking expectations before it sends the cloned non-finalized state and publishes the best non-finalized tip.

zebra-state/src/service/write.rs

```rust
fn update_latest_chain_channels([...]) -> block::Height {
    let best_chain = non_finalized_state.best_chain().expect("unexpected empty non-finalized state: must commit at least one block before updating channels");
    let tip_block = best_chain
        .tip_block()
        .expect("unexpected empty chain: must commit at least one block before updating channels")
        .clone();
    let tip_block = ChainTipBlock::from(tip_block);
    let tip_block_height = tip_block.height;
    if let Some(backup_dir_path) = backup_dir_path {
        non_finalized_state.write_to_backup(backup_dir_path);
    }
    // If the final receiver was just dropped, ignore the error.
    let _ = non_finalized_state_sender.send(non_finalized_state.clone());
    chain_tip_sender.set_best_non_finalized_tip(tip_block);
    tip_block_height
}
```

After PR#10592 allowed the root-invalidation edge case to complete without panicking inside invalidation itself, the unconditional channel update becomes the remaining panic point. If root invalidation empties `NonFinalizedState`, the helper receives an empty state immediately after the response is sent and panics during follow-up channel propagation, terminating the write worker path for later state updates.

Remediation

Make the latest-chain channel update handle an empty non-finalized state explicitly.

Stale Finalized Block Panic in TrustedChainSync

ID	Severity	Status	Found in
OS-ZCA-ADV-09	Medium	✗ To-do	90ea4a9

Description

The read-state syncer can accept a stale block from the non-finalized stream when its secondary finalized database advances during the commit path. The surrounding `sync` loop reads streamed `BlockAndHash` messages, checks whether a block is already finalized before calling `try_commit`, and then `try_commit` calls `try_catch_up_with_primary` again. The snippet below shows that after this catch-up, `try_commit` only handles a missing finalized gap before forwarding the same block to `commit`. It does not re-check whether the incoming block is now at or below the secondary finalized tip.

zebra-rpc/src/sync.rs

```
async fn try_commit(
    &mut self,
    block: SemanticallyVerifiedBlock,
) -> Result<(), ValidateContextError> {
    self.try_catch_up_with_primary().await;
    // When the non-finalized state is empty and the incoming block doesn't build directly on
    // the secondary's finalized tip, the secondary's finalized state is lagging the primary's.
    // The streamed non-finalized blocks start at the primary's finalized tip, which can be
    // several blocks above ours, so the incoming block has no parent to attach to. Bridge that
    // gap by fetching the missing (already-finalized) blocks from the primary, so the incoming
    // block has a contiguous chain to commit onto.
    if self.non_finalized_state.best_chain().is_none()
        && self.db.finalized_tip_hash() != block.block.header.previous_block_hash
    {
        self.fill_finalized_gap(block.height).await;
    }
    self.commit(block)
}
```

Post-catch-up commit path

If the catch-up made the queued block stale, `commit` can still be reached with that block. As shown in `commit` below, a block whose parent matches the finalized tip starts a new non-finalized chain, while other blocks are committed against the existing non-finalized state, and the function proceeds toward pruning and channel publication rather than treating an already-finalized block as a no-op.

zebra-rpc/src/sync.rs

```
fn commit(&mut self, block: SemanticallyVerifiedBlock) -> Result<(), ValidateContextError> {
    if self.db.finalized_tip_hash() == block.block.header.previous_block_hash {
        self.prune_finalized();
        self.non_finalized_state.commit_new_chain(block, &self.db)?;
    } else {
```

```
        self.non_finalized_state.commit_block(block, &self.db)?;
        self.prune_finalized();
      }
      self.update_channels();
      Ok(())
    }
```

Commit and pruning path

Pruning is keyed to the secondary finalized tip height. When the finalized state has advanced to or beyond the queued block, `prune_finalized` can remove the newly inserted non-finalized root and leave `NonFinalizedState::best_chain()` empty. `update_channels` implementation then treats an empty non-finalized state as an invariant violation and panics while publishing the chain tip. This panic stops the `TrustedChainSync` task that maintains the read-state syncer's non-finalized view and latest chain tip channels for RPC/indexer consumers.

zebra-rpc/src/sync.rs

```
/// Sends the new chain tip and non-finalized state to the latest chain channels.
// TODO: Replace this with the `update_latest_chain_channels()` fn in `write.rs`.
fn update_channels(&mut self) {
    // If the final receiver was just dropped, ignore the error.
    let _ = self
        .non_finalized_state_sender
        .send(self.non_finalized_state.clone());

    let best_chain = self.non_finalized_state.best_chain().expect("unexpected empty non-finalized state: must commit at least one block before updating channels");

    let tip_block = best_chain
        .tip_block()
        .expect(
            "unexpected empty chain: must commit at least one block before updating channels",
        )
        .clone();

    self.chain_tip_sender
        .set_best_non_finalized_tip(Some(tip_block.into()));
}
```

Empty non-finalized state panic

Remediation

After every `try_catch_up_with_primary()` call that can advance the secondary finalized database, re-check the incoming block against the secondary finalized tip before calling `commit`. If the block height is at or below the finalized tip, skip it as already finalized. Also make channel publication tolerate an empty non-finalized state after pruning by publishing the finalized tip from the secondary database instead of unconditionally unwrapping `best_chain()`.

Non-Ping Timeouts Leave Stalling Peers Connected

ID	Severity	Status	Found in
OS-ZCA-ADV-10	Low	✗ To-do	f5feadb

Description

In `connection.rs` timeouts are unpunished. When Zebra sends an outbound client request to a peer, `handle_client_request` moves the connection into `State::AwaitingResponse` and arms `request_timer` with `REQUEST_TIMEOUT`. In the `Connection::run` timeout branch, `Handler::Ping` is treated as a special case that fails the connection, but every other pending handler returns `PeerError::ConnectionReceiveTimeout` to the waiting request and moves the same connection back to `State::AwaitingRequest`.

Because the non-ping timeout path does not call `fail_with` or update the connection error slot, the peer remains connected after each non-ping receive timeout. The state machine also allows only one active client request while a response is pending, so a peer that repeatedly advertises useful data and then withholds non-ping responses can occupy that request slot until the timer expires, causing failed requests and retries without accumulating connection-level timeout health.

A malicious peer can remain connected across repeated non-ping timeouts, delaying each request routed to it until `REQUEST_TIMEOUT`. Broader peer-set degradation requires repeated selection or multiple attacker-controlled connections. Existing FindBlocks and FindHeaders stall handling does not cover this path.

Remediation

Track non-ping outbound receive timeouts per peer instead of only returning `ConnectionReceiveTimeout` to the individual request. Record the timeout and disconnect or penalize the peer once it exceeds a defined threshold.

Gossiped Block Download Lacks Fallback Retry

ID	Severity	Status	Found in
OS-ZCA-ADV-11	Low	✗ To-do	f5feadb

Description

Zebra processes each single-block inv(H) in two places. register_inventory_status() records (H, peer) as Available in InventoryRegistry, while the inbound service calls Downloads::download_and_verify(H, advertiser). The first call inserts H into cancel_handles and spawns a BlocksByHash({H}) task; later calls for H return DownloadAction::AlreadyQueued. Those later peer advertisements are still retained in the registry, and PeerSet::route_inv() selects among the ready peers recorded for H, so the first advertiser does not become the only source.

If the selected peer times out, disconnects, or returns NotFound, the peer can be marked Missing and the task finishes with an error. Downloads::poll_next() removes H from cancel_handles, but its error item returned to Inbound::poll_ready() contains the error and optional peer address, not the block hash. Inbound::poll_ready() consumes that result without calling download_and_verify(H) again. The remaining Available peers are therefore unused until another inv(H) arrives or the syncer independently discovers H, delaying retrieval even though an alternate source is already known.

Remediation

Return the block hash with failed download results and requeue it after retryable network failures, allowing route_inv() to select another Available peer. Use a per-hash retry limit and backoff, and do not retry consensus-invalid blocks or other permanent validation failures.

Invalid-block Scoring Lost on Reset

ID	Severity	Status	Found in
OS-ZCA-ADV-12	Low	✗ To-do	d9d29b6

Description

Zebra emits invalid-block peer penalties only after a completed download and verification result reaches `Syncer::handle_block_response`. The excerpt below shows that the `Invalid` response path sends a misbehavior score for the advertising peer only inside that handler, when the verifier error carries a nonzero score and an `advertiser_addr`. The surrounding sync logic polls `Downloads` results and then calls this handler, so task completion alone is not enough to record the score.

zebrad/src/components/sync.rs

```
fn handle_block_response(
  &mut self,
  response: Result<(Height, block::Hash), BlockDownloadVerifyError>,
) -> Result<(), BlockDownloadVerifyError> {
  match response {
    [...]
    Err(BlockDownloadVerifyError::Invalid {
      ref error,
      advertiser_addr: Some(advertiser_addr),
      ..
    }) if error.misbehavior_score() != 0 => {
      let _ = self
        .misbehavior_sender
        .try_send((advertiser_addr, error.misbehavior_score()));
    }
    [...]
  };[...]
}
```

Invalid-block responses are scored in `handle_block_response`.

A reset can interrupt that path before every completed verifier result has been observed. In `sync`, a failed `try_to_sync` call leads to `Downloads::cancel_all`, and `try_to_sync_once` uses `?` after `handle_block_response`, so the first restart-triggering block result can exit the drain loop. As shown below, `cancel_all` clears the pending task set and drains cancellation handles without first polling pending tasks for already-completed verifier results.

zebrad/src/components/sync/downloads.rs

```
/// Cancel all running tasks and reset the downloader state.
pub fn cancel_all(&mut self) {
  // Replace the pending task list with an empty one and drop it.
  let _ = std::mem::take(&mut self.pending);

  // Signal cancellation to all running tasks.
```

```
    // Since we already dropped the JoinHandles above, they should
    // fail silently.
    for (_hash, cancel) in self.cancel_handles.drain() {
        let _ = cancel.send(());
    }[...]
}
```

Reset drops pending downloader tasks in `cancel_all`.

Because the `Downloads` stream removes cancellation handles and returns task results only when it is polled, completed invalid verifier results that remain in the pending set at reset may be discarded before they enter `handle_block_response`. If several peers concurrently advertise invalid blocks, the first observed invalid result can score its peer and trigger reset, while other already-completed invalid results may not emit their misbehavior events. This weakens peer scoring for concurrent invalid-block failures by allowing some bad peers in that reset batch to avoid the intended penalty.

Remediation

Before resetting the downloader, drain all immediately ready `Downloads` results and route them through the same scoring logic used by `handle_block_response`, while preserving the original restart decision.

Lookahead / Sync Progress Calculation Error

ID	Severity	Status	Found in
OS-ZCA-ADV-13	Low	✗ To-do	fcf5e6a

Description

The syncer computes a dynamic lookahead limit in `lookahead_limit` and uses it to decide how many announced block hashes to queue near the transition from checkpoint verification to full verification. The excerpt below shows the transition branch. The surrounding comment states the limit should allow the remaining checkpoint work plus a separate full-verification lookahead, but the calculation derives the checkpoint component from how far the current batch of newly supplied hashes extends past the checkpoint height.

zebrad/src/components/sync.rs

```
fn lookahead_limit(&self, new_hashes: usize) -> usize {
  [...]
  if verified_height >= max_checkpoint_height {
    self.full_verify_concurrency_limit
  } else if (verified_height + new_hashes) >= max_checkpoint_height {
    // If we're just about to start full verification, allow enough for the remaining checkpoint,
    // and also enough for a separate full verification lookahead.
    let checkpoint_hashes = verified_height + new_hashes - max_checkpoint_height;
    self.full_verify_concurrency_limit + checkpoint_hashes
  } else {
    self.checkpoint_verify_concurrency_limit
  }
}
```

That batch length is not purely local state. In `obtain_tips` and `extend_tips`, Zebra builds the download set from peer `FindBlocks` responses, records the number of newly added hashes, and passes the resulting hash set into `request_blocks` `request_blocks` then calls `lookahead_limit` with the hash-set length before splitting excess hashes into `extra_hashes`. The same limit is also used by `try_to_sync_once` to pause while downloads are in flight. As a result, when the verified tip is below the maximum checkpoint height but the peer-provided batch crosses that height, the lookahead decision is influenced by the response shape instead of only by the local distance to the checkpoint and the configured full-verification capacity. This can distort lookahead enforcement based on attacker-controlled response shape.

Remediation

Recompute the transition limit from local chain state and configuration only.

Sinsemilla Empty-Message Output Underconstrained

ID	Severity	Status	Found in
OS-ZCA-ADV-14	Low	✗ To-do	27b2b33

Description

`SinsemillaChip::configure` constrains the accumulator transition and final output y-coordinate in the main Sinsemilla gate, and that gate is guarded by `q_sinsemilla1`. The final y-coordinate is read from the next row's `lambda_1` cell inside the `q_sinsemilla1`-selected constraints, so those constraints are absent whenever that selector is not enabled.

`hash_all_pieces` only invokes `hash_piece` while iterating over message pieces. Because `hash_piece` is the path that enables `q_sinsemilla1` for each processed word, the loop shown below leaves `q_sinsemilla1` disabled when `message` contains no pieces; the surrounding function then still assigns the final `y_a` into the `lambda_1` column at the current offset.

halo2_gadgets/src/sinsemilla/chip/hash_to_point.rs

```
fn hash_all_pieces([...]) -> Result<[...]> {
 [...]
 // Hash each piece in the message.
 for (idx, piece) in message.iter().enumerate() {
  let final_piece = idx == message.len() - 1;

  // The value of the accumulator after this piece is processed.
  let (x, y, zs) = self.hash_piece(region, offset, piece, x_a, y_a, final_piece)?;

  // Since each message word takes one row to process, we increase
  // the offset by `piece.num_words` on each iteration.
  offset += piece.num_words();

  // Update the accumulator to the latest value.
  x_a = x;
  y_a = y;
  zs_sum.push(zs);
 }[...]
}
```

`hash_all_pieces` skips `hash_piece` for an empty message

The only Sinsemilla selector that remains active on the empty-message path is the initialization selector, as shown in the `Initial y_Q` gate below. That gate enforces consistency with `y_Q` through the initial double-and-add expression, but it does not independently bind the returned `lambda_1` output cell to `Q.y` or prove that the returned coordinates are exactly the initial point `Q`.

halo2_gadgets/src/sinsemilla/chip.rs

```
pub fn configure([...]) -> <Self as Chip<pallas::Base>>::Config {
```

```
    [...]
    meta.create_gate("Initial y_Q", |meta| {
      let q_s4 = meta.query_selector(config.q_sinsemilla4);
      let y_q = if allow_init_from_private_point {
        meta.query_advice(config.double_and_add.x_p, Rotation::prev())
      } else {
        meta.query_fixed(config.fixed_y_q)
      };

      // Y_A = (lambda_1 + lambda_2) * (x_a - x_r)
      let Y_A_cur = Y_A(meta, Rotation::cur());

      // 2 * y_q - Y_{A,0} = 0
      let init_y_q_check = y_q * two - Y_A_cur;

      Constraints::with_selector(q_s4, Some(("init_y_q_check", init_y_q_check)))
    });
    [...]
  }
```

Initialization gate guarded by `q_sinsemilla4`

For a zero-length Sinsemilla invocation, this selector layout leaves the output y-coordinate underconstrained under the circuit constraints. The returned x-coordinate remains the initialized `x_Q`, but the y-coordinate is taken from the final `lambda_1` cell without the `q_sinsemilla1` y-check that normally binds it to the accumulator, allowing a prover to satisfy the initialization equation by choosing compatible `lambda_2` and `x_p` while selecting an arbitrary `lambda_1`. As a result, the output may differ from Q and may not be a valid curve point. Note that Orchard's current fixed-length uses are not affected.

Remediation

Special-case `message.len() == 0` in both `hash_message` and `hash_message_with_private_init` before calling `hash_all_pieces`. In that branch, return or assign output coordinates that are explicitly constrained to the initial point `Q`.

Far-Ahead Hash Rejection is Misattributed

ID	Severity	Status	Found in
OS-ZCA-ADV-15	Low	✗ To-do	fcf5e6a

Description

Zebra accepts block hashes returned by `FindBlocks` in the syncer's tip-acquisition and tip-extension paths, then stores accepted unknown hashes in an `IndexSet<block::Hash>` named `download_set`. Those hashes are later passed into `request_blocks`, which accepts only hashes and queues each one with `downloads.download_and_verify(hash)`. Because the queue entry contains no peer address from the `FindBlocks` response, the peer that caused a far-ahead hash to be queued is no longer available when the block is downloaded and checked.

When a queued hash later downloads to a block above the lookahead height limit, the rejection path uses the `advertiser_addr` carried by the block download/verify error. The code below shows `handle_block_response` assigning a misbehavior score for `AboveLookaheadHeightLimit` to that address, which is sourced from the peer that served the block rather than from the peer that supplied the original far-ahead hash.

zebrad/src/components/sync.rs

```
fn handle_block_response(
  &mut self,
  response: Result<(Height, block::Hash), BlockDownloadVerifyError>,
) -> Result<(), BlockDownloadVerifyError> {
  match response {
    [...]
    Err(BlockDownloadVerifyError::AboveLookaheadHeightLimit {
      advertiser_addr: Some(advertiser_addr),
      ..
    }) => {
      let _ = self.misbehavior_sender.try_send((advertiser_addr, 100));
    }
    [...]
  };
  [...]
}
```

This can misattribute the lookahead rejection as an unrelated peer that merely served the requested block may be scored, while the peer that returned the bad locator response may avoid scoring.

Remediation

Carry peer attribution with queued hashes and score the peer responsible for the bad locator response.

Sendrawtransaction Lacks Dedicated RPC Work Limits

ID	Severity	Status	Found in
OS-ZCA-ADV-16	Low	✗ To-do	d9d29b6

Description

`send_raw_transaction` accepts the raw transaction supplied to the RPC method, deserializes it, records it in the RPC retry channel, and then submits the transaction to the mempool as a normal queue request. The code below shows that the immediate mempool request is built as `Request::Queue` from `Gossip::Tx` and awaited for a verification result, without attaching a peer or RPC-origin identifier to the queued candidate.

zebra-rpc/src/methods.rs

```
async fn send_raw_transaction([...]) -> Result<SendRawTransactionResponse> {
    let mempool = self.mempool.clone();
    let queue_sender = self.queue_sender.clone();

    let raw_transaction_bytes = Vec::from_hex(raw_transaction_hex)
        .map_error(server::error::LegacyCode::Deserialization)?;
    let raw_transaction = Transaction::zcash_deserialize(&*raw_transaction_bytes)
        .map_error(server::error::LegacyCode::Deserialization)?;

    let transaction_hash = raw_transaction.hash();

    // send transaction to the rpc queue, ignore any error.
    let unmined_transaction = UnminedTx::from(raw_transaction.clone());
    let _ = queue_sender.send(unmined_transaction);

    let transaction_parameter = mempool::Gossip::Tx(raw_transaction.into());
    let request = mempool::Request::Queue(vec![transaction_parameter]);

    let response = mempool.oneshot(request).await.map_misc_error()?;
    [...]
}
```

RPC submission enters the generic mempool queue

The mempool service treats `Request::Queue` as a source-less queue path. In the handler, each candidate is checked against local mempool and rejection state, then passed to `download_if_needed_and_verify` with no source. Downloader accounting depends on whether that source is present. In the downloader implementation, the per-peer pending cap is enforced only when the source argument is present. As a result, authorized `sendrawtransaction` submissions bypass the peer attribution used for peer-originated transaction pushes and advertisements.

After admission, the same downloader path performs state and verification work: it checks whether the transaction is already in the best chain, requests the current tip, and submits the transaction to the mempool verifier. Because this work uses the shared mempool downloader, state service, and verifier rather than a separate RPC budget or reserved state capacity, sustained authorized RPC submissions may occupy shared work capacity and delay unrelated state users.

Remediation

Add a separate RPC budget, reserve state capacity, or attach an RPC origin for accounting.

7. Suggestions

Here, we present a discussion of the 9 suggestions we identified during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Severity	Status	Description
OS-ZCA-SUG-00	Informational	✗ To-do	Presence of Incomplete-Addition Preconditions
OS-ZCA-SUG-01	Informational	✗ To-do	Halo2 Verifier MSM Optimization
OS-ZCA-SUG-02	Informational	✗ To-do	Acceptance of Identity Binding Key
OS-ZCA-SUG-03	Informational	✗ To-do	Panic on Invalid TransmissionKey Encoding
OS-ZCA-SUG-04	Informational	✗ To-do	Panics due to Improper Error Handling
OS-ZCA-SUG-05	Informational	✗ To-do	FromDisk Panics on Corrupt Finalized DB Bytes
OS-ZCA-SUG-06	Informational	✗ To-do	Mempool Gossip Drops Added Events
OS-ZCA-SUG-07	Informational	✗ To-do	AdvertiseTransactionIds Ignores Relay Flag
OS-ZCA-SUG-08	Informational	✗ To-do	Failed Peers Stranded by Stale Last-Seen

Presence of Incomplete-Addition Preconditions

ID	Severity	Status	Found in
OS-ZCA-SUG-00	Informational	✗ To-do	27b2b33

Description

`hash_message_with_private_init` is the Sinsemilla chip entry point used for hashing with a private initial point. The issue is that Sinsemilla uses incomplete addition formulas, so private initial points may need to satisfy non-exceptional point relations before the gadget can safely rely on those formulas. In particular, if a private initial point leads to equal x-coordinate additions or other exceptional cases, the gadget may need extra constraints or caller-side preconditions. Without documented or enforced non-exceptional point relations, this API may leave callers relying on implicit assumptions about which private initial points and messages are supported.

halo2_gadgets/src/sinsemilla/chip/hash_to_point.rs

```
impl<Hash, Commit, Fixed, Lookup> SinsemillaChip<Hash, Commit, Fixed, Lookup> where Hash: HashDomains<pallas::Affine>, Fixed: FixedPoints<pallas::Affine>, Commit: CommitDomains<pallas::Affine, Fixed, Hash>, Lookup: PallasLookupRangeCheck, {
  [...]
  #[allow(clippy::type_complexity)]
  pub(super) fn hash_message_with_private_init(
    &self,
    [...]
}
```

Private-initialization hash entry point

Remediation

Document the required non-exceptional relations for `hash_message_with_private_init`, including that private initial points must avoid equal-x-coordinate additions or other exceptional incomplete-addition cases. If arbitrary private initial points are accepted across a security boundary, enforce the required relations with in-circuit constraints before relying on incomplete formulas.

Halo2 Verifier MSM Optimization

ID	Severity	Status	Found in
OS-ZCA-SUG-01	Informational	✗ To-do	261faac

Description

`verify_proof` computes commitments for public instance columns before hashing those commitments into the transcript and before later using them as verifier queries. Each instance column is length-checked, copied into a new vector, padded with zeros to the full circuit domain size, converted into a Lagrange-basis polynomial, and then passed to `params.commit_lagrange` with the default blinding term.

halo2_proofs/src/plonk/verifier.rs

```
pub fn verify_proof< 'params, C: CurveAffine, E: EncodedChallenge<C>, T: TranscriptRead<C, E>, V:
VerificationStrategy<'params, C>, >([...]) -> Result<V::Output, Error> {
 [...]
 let instance_commitments = instances
    .iter()
    .map(|instance| {
      instance
        .iter()
        .map(|instance| {
          if instance.len() > params.n as usize - (vk.cs.blinding_factors() + 1) {
            return Err(Error::InstanceTooLarge);
          }
          let mut poly = instance.to_vec();
          poly.resize(params.n as usize, C::Scalar::ZERO);
          let poly = vk.domain.lagrange_from_vec(poly);

          Ok(params.commit_lagrange(&poly, Blind::default()).to_affine())
        })
        .collect::<Result<Vec<_>, _>>()
    })
    .collect::<Result<Vec<_>, _>>()?;
 [...]
}
```

Verifier pads instance columns before committing

This construction makes the verifier's instance commitment work proportional to the full domain size even when the supplied instance column is short. The commitment is mathematically sparse after padding, so the zero tail does not contribute to the result, but `commit_lagrange` still builds its multi-scalar multiplication over the full Lagrange generator set plus the blinding base. For workloads with short public instance columns, this can perform substantially more MSM work than necessary.

The same commitment can be computed directly from the non-padded instance values by using only the first `instance.len()` Lagrange bases and the blinding base. This would reduce the instance commitment MSM from

`n + 1` points to `len + 1` points (such as `2049` to `10` for Orchard-action workload), and would improve single-action verifier performance by approximately `20%`.

Remediation

Implement the above mentioned optimization.

Acceptance of Identity Binding Key

ID	Severity	Status	Found in
OS-ZCA-SUG-02	Informational	✗ To-do	8de1724

Description

In Orchard, the `Bundle::binding_validating_key` implementation shown below derives the bundle-level binding validating key by summing all Action value commitments and subtracting a zero-trapdoor commitment to the public `value_balance`, converting the resulting point into a `VerificationKey<Binding>`.

orchard/src/bundle.rs

```
pub fn binding_validating_key(&self) -> redpallas::VerificationKey<Binding> {
  // https://p.z.cash/TCR:bad-txns-orchard-binding-signature-invalid?partial
  (self
    .actions.iter().map(|a| a.cv_net())
    .sum::<ValueCommitment>()
    - ValueCommitment::derive(
      ValueSum::from_raw(self.value_balance.into()),
      ValueCommitTrapdoor::zero(),
    ))
  .into_bvk()
}
```

Unlike spend authorization keys (`rk`), which are explicitly rejected if they encode the identity point during deserialization, the computed binding validating key is not subjected to an equivalent identity check.

zebra-chain/src/orchard/action.rs

```
fn zcash_deserialize<R: io::Read>(mut reader: R) -> Result<Self, SerializationError> {
  [...]
  Ok(Action {
    [...]
    rk: {
      let rk_bytes = reader.read_32_bytes()?;
      if rk_bytes == [0u8; 32] {
        return Err(SerializationError::Parse([...]));
      }
      rk_bytes.into()
    },
    [...]
  })
}
```

Furthermore, the generic `reddsa::VerificationKey::try_from` constructor intentionally accepts identity and other small-order verification keys to comply with the RedDSA specification. Consequently, it is possible for a validly constructed bundle to derive an identity binding validating key. If the binding validating key is the identity,

the RedDSA verification equation simplifies from $sB = R + cA$ to $sB = R$, since $A = 0$ implies $cA = 0$. While this changes the form of the signature verification equation, it would require a concrete inflation, consensus-divergence, or acceptance-bypass path before this becomes a security vulnerability.

Remediation

Ensure to take the above design considerations into account when implementing the Orchard binding signature verification.

Panic on Invalid TransmissionKey Encoding

ID	Severity	Status	Found in
OS-ZCA-SUG-03	Informational	✗ To-do	bc2da19

Description

`TransmissionKey::try_from` is documented as a fallible parser for a 32-byte Sapling diversified transmission key, including malformed, non-canonical, and non-prime-subgroup inputs. As shown below, the implementation unwraps the Jubjub point decoding result before it can return through the function's `Result` error path, so bytes that do not decode to an affine point panic instead of being rejected cleanly.

zebra-chain/src/sapling/keys.rs

```
fn try_from(bytes: [u8; 32]) -> Result<Self, Self::Error> {
  let affine_point = jubjub::AffinePoint::from_bytes(bytes).unwrap();
  // Check if it's identity or has prime order (i.e. is in the prime-order subgroup).
  if affine_point.is_torsion_free().into() {
    Ok(Self(affine_point))
  } else {
    Err(\"Invalid jubjub::AffinePoint value for Sapling TransmissionKey\")
  }
}
```

Premature unwrap in `TransmissionKey::try_from`

The root cause is that the fallible result from `jubjub::AffinePoint::from_bytes` is treated as already-valid input, while the only explicit validation that remains is the torsion-free subgroup check after the unwrap.

The impact is scoped to callers that parse Sapling transmission keys from 32-byte input. This is reachable from note-scanning or wallet-style code rather than Zebra's block-validation hot path. In those contexts, malformed input may cause a panic instead of a structured parse error.

Remediation

Update `TransmissionKey::try_from` to keep the `CtOption` returned by `jubjub::AffinePoint::from_bytes`, check whether it contains a point before unwrapping, and return the existing `Result` error for invalid or non-canonical encodings. Only after successful decoding should it run the torsion-free subgroup check.

Panics due to Improper Error Handling

ID	Severity	Status	Found in
OS-ZCA-SUG-04	Informational	✗ To-do	bc2da19

Description

zebra-chain::serialization::serde_helpers defines remote Serde helpers for several Zcash crypto field and group encodings. The affected conversions cover Jubjub, Pallas/Pasta, Sapling value commitments, extracted note commitments, and Sapling tree node types.

In those remote From conversions, the module calls fallible from_bytes, from_repr, and from_bytes_not_small_order constructors and immediately unwraps the returned CtOption. Because these conversions run during Serde deserialization for fields annotated with serde_helpers, invalid 32-byte encodings are not reported through the deserializer and instead panic when the conversion result is empty. This differs from nearby Zcash deserialization paths, which validate the same classes of values and return parse errors for invalid encodings. A corrupted stored value that reaches one of these Serde decode paths may therefore abort decoding with a panic instead of yielding a recoverable deserialization error.

Remediation

Replace the remote-Serde conversions that currently rely on infallible From implementations with custom Serde deserializers that read the 32-byte representation, call the appropriate fallible constructor, convert the CtOption into an Option, and return a deserialization error when the encoding is invalid.

FromDisk Panics on Corrupt Finalized DB Bytes

ID	Severity	Status	Found in
OS-ZCA-SUG-05	Informational	✗ To-do	bc2da19

Description

Finalized-state disk decoding uses `FromDisk` implementations that treat DB bytes as trusted and use `expect()` / `unwrap()` instead of returning integrity errors. The trait documentation in `disk_format.rs` states that deserialization failures panic, and the generic database read paths call `FromDisk` while reading values from RocksDB. In `shielded.rs`, the Sapling tree root decoder shown below treats the stored bytes as trusted and uses panic-on-error conversions; nearby shielded finalized-state decoders use the same pattern for other roots, note commitment trees, and subtree nodes.

zebra-state/src/service/finalized_state/disk_format/shielded.rs

```
impl FromDisk for sapling::tree::Root {
 fn from_bytes(bytes: impl AsRef<[u8]>) -> Self {
  let array: [u8; 32] = bytes.as_ref().try_into().unwrap();
  array.try_into().expect(\"finalized data must be valid\")
 }
}
```

Sapling root decoding trusts finalized shielded bytes

Transparent UTXO data has the same issue. The `transparent::Output` decoder shown below is used for the `utxo_by_out_loc` finalized-state column, and callers such as `utxo_by_location` obtain the stored output through the generic database read path. If the bytes in that column no longer deserialize as a transparent output, the failure is not returned to the state layer as a database integrity problem.

zebra-state/src/service/finalized_state/disk_format/transparent.rs

```
impl FromDisk for transparent::Output {
 fn from_bytes(bytes: impl AsRef<[u8]>) -> Self {
  bytes.as_ref().zcash_deserialize_into().unwrap()
 }
}
```

Transparent output decoding panics on invalid bytes

Block data is also decoded through panic-on-error `FromDisk` implementations. The block header decoder shown below is called when `block_and_size` reconstructs finalized blocks from raw header bytes, and the same file contains an analogous transaction decoder used for raw transaction bytes. Corruption in either stored header or transaction bytes can therefore terminate the process during normal finalized-state reads.

zebra-state/src/service/finalized_state/disk_format/block.rs

```
impl FromDisk for block::Header {
 fn from_bytes(bytes: impl AsRef<[u8]>) -> Self {
  bytes
```

```
221         .as_ref()
222         .zcash_deserialize_into()
223         .expect("deserialization format should match the serialization format used by IntoDisk")
224     }
225 }
```

Block header decoding panics on deserialization failure

Because these failures occur while reading persistent finalized database columns, restarting the node against the same damaged database may encounter the same panic again until the affected data is repaired or the database is rebuilt. Thus, since finalized-state byte decoding conflates the invariant expected from Zebra-written data with the integrity of already-persisted RocksDB contents, malformed on-disk bytes cannot be reported with enough context for controlled recovery.

Remediation

Make finalized-state disk decoding fallible for persisted bytes. Replace the affected `FromDisk` panic paths for shielded roots/nodes, transparent outputs, block headers, and transactions with conversions that return an integrity error containing the column family, key or location when available, decoded type, and underlying deserialization failure so that Zebra can report, quarantine, repair, or rebuild the damaged column instead of panicking.

Mempool Gossip Drops Added Events

ID	Severity	Status	Found in
OS-ZCA-SUG-06	Informational	✗ To-do	d9d29b6

Description

Zebra's mempool transaction gossip task receives `MempoolChange` events from a `tokio::sync::broadcast` receiver and advertises only changes whose kind is `Added`. In `gossip_mempool_transaction_id`, the receive loop treats lag on that broadcast channel as an informational condition: the excerpt below shows that `RecvError::Lagged` is logged, but the task does not reload the current mempool contents or otherwise reconstruct the skipped `Added` transaction IDs before waiting for the next event.

zebrad/src/components/mempool/gossip.rs

```
pub async fn gossip_mempool_transaction_id<ZN>([...]) -> Result<(), BoxError> where ZN: Service<zn::Request, Response = zn::Response, Error = BoxError> + Send + Clone + 'static, ZN::Future: Send, {
  [...]
  loop {
    [...]
    let mut txs = loop {
      match receiver.recv().await {
        [...]
        }
        Err(RecvError::Lagged(skip_count)) => info!(
          ?skip_count,
          [...]
        )
      };
      [...]
    }
  }
```

Mempool gossip logs receiver lag without resynchronizing skipped additions

The surrounding mempool logic creates this broadcast channel with a bounded capacity and sends `MempoolChange::added` when newly verified transactions should be sent to peers and RPC listeners. Because the gossip task continues after lag without reconciliation, unread `Added` events that were evicted from the receiver may be lost for this consumer.

Remediation

Handle lag as a state-reconciliation event instead of only logging it. On `RecvError::Lagged` and `TryRecvError::Lagged`, make the gossip task resynchronize from authoritative mempool state. Alternatively, replace this path with a non-lossy or backpressured event mechanism for `Added` events that downstream gossip must not miss.

AdvertiseTransactionIds Ignores Relay Flag

ID	Severity	Status	Found in
OS-ZCA-SUG-07	Informational	✗ To-do	d9d29b6

Description

Zebra records each peer's transaction relay preference as part of the negotiated `VersionMessage`, and the peer connection metadata retains that remote version message. However, transaction-inventory advertisements are dispatched through the same generic broadcast branch as block advertisements. As shown below, `PeerSet::call` routes `AdvertiseTransactionIds` directly into the generic broadcast path rather than a relay-aware transaction path.

zebra-network/src/peer_set/set.rs

```
fn call(&mut self, req: Request) -> Self::Future {
  let fut = match req {
    [...]
    // Broadcast advertisements to lots of peers
    Request::AdvertiseTransactionIds(_, _) => self.route_broadcast(req),
    [...]
  };
  self.update_metrics();
  fut
}
```

In the peer-set implementation, the generic broadcast path chooses random entries from `ready_services` and sends the same request to those selected ready peers. That selection accounts for peer readiness and supported protocol version, but it does not filter candidates by the remote peer's stored `version.relay` value. As a result, a ready peer that set `relay` to false can still be selected for transaction inventory gossip.

Remediation

Filter broadcast candidates by relay preference when sending transaction inventory.

Failed Peers Stranded by Stale Last-Seen

ID	Severity	Status	Found in
OS-ZCA-SUG-08	Informational	✗ To-do	d9d29b6

Description

`MetaAddr::is_probably_reachable` gates whether an address is still considered worth dialing after prior connection activity. As shown in the excerpt below, a peer in `Failed` remains probably reachable only if its `last_seen` value is recent, and the surrounding comments explain that an address with an old or absent gossiped `last_seen` can be tried once and then not tried again after a failed attempt.

zebra-network/src/meta_addr.rs

```
/// Should this peer considered reachable?
///
/// A peer is probably reachable if:
/// - it has never been attempted, or
/// - the last connection attempt was successful, or
/// - the last successful connection was less than 3 days ago.
/// [...]
/// The `untrusted_last_seen` time is used as a fallback time if the local node has never
/// itself seen the peer. If the reported last seen time is a long time ago or `None`, then the local
/// node will attempt to connect the peer once, and if that attempt fails it won't
/// try to connect ever again. (The state can't be `Failed` until after the first connection attempt.)
pub fn is_probably_reachable(&self, now: chrono::DateTime<Utc>) -> bool {
  self.last_connection_state != PeerAddrState::Failed || self.last_seen_is_recent(now)
}
```

Reachability check for failed peers

`AddressBook::reconnection_peers` applies that readiness logic before returning outbound reconnection candidates. The iterator filters stored peers through `is_ready_for_connection_attempt`, so a failed entry whose stale `last_seen` makes it not probably reachable is excluded from the reconnection iterator instead of becoming eligible after the normal recent-update delays expire.

zebra-network/src/address_book.rs

```
/// Return an iterator over peers that are due for a reconnection attempt,
/// in reconnection attempt order.
pub fn reconnection_peers([...]) -> impl DoubleEndedIterator<Item = MetaAddr> + '_ {
  let _guard = self.span.enter();

  // Skip live peers, and peers pending a reconnect attempt.
  // The peers are already stored in sorted order.
  self.by_addr
    .descending_values()
    .filter(move |peer| {
      peer.is_ready_for_connection_attempt(instant_now, chrono_now, &self.network)
```

```
        && self.is_ready_for_connection_attempt_with_ip(&peer.addr.ip(), chrono_now)
    })
    .cloned()
}
```

Reconnection candidate filtering

Fresh gossip does not necessarily repair the stale timestamp once Zebra has attempted the address. In `MetaAddrChange::apply_to_meta_addr`, `NewGossiped` maps to a never-attempted address state in the surrounding code, and the excerpt below shows never-attempted changes being rejected for entries that have already been attempted unless they are misbehavior updates. As a result, later gossiped information for the same address can be ignored after a transient local dial failure records attempted state.

zebra-network/src/meta_addr.rs

```
pub fn apply_to_meta_addr([...]) -> Option<MetaAddr> {
  [...]
  let previous_has_been_attempted = !previous.last_connection_state.is_never_attempted();
  let change_to_never_attempted = self.peer_addr_state().is_never_attempted();
  let is_misbehavior_update = self.misbehavior_score() != 0;
  // Invalid changes
  if change_to_never_attempted && previous_has_been_attempted && !is_misbehavior_update {
    // Existing entry has been attempted, change is NeverAttempted
    // - ignore the change
    //
    // # Security
    //
    // Ignore NeverAttempted changes once we have made an attempt,
    // so malicious peers can't keep changing our peer connection order.
    return None;
  }
  [...]
}
```

Rejected gossiped update after an attempt

The same update path also preserves the prior gossiped timestamp for attempted, responded, or failed entries. The branch for non-never-attempted entries keeps the existing `untrusted_last_seen`, so updates that reach this branch do not refresh the fallback timestamp used by `last_seen` when Zebra has no local response time for that address.

zebra-network/src/meta_addr.rs

```
pub fn apply_to_meta_addr([...]) -> Option<MetaAddr> {
  [...]
  } else {
    [...]
    Some(MetaAddr {
      addr: self.addr(),
```

```
        // Always update optional fields, unless the update is None.
        //
        // We want up-to-date services, even if they have fewer bits
        services: self.untrusted_services().or(previous.services),
        // Only NeverAttempted changes can modify the last seen field
        untrusted_last_seen: previous.untrusted_last_seen,
        // This is a wall clock time, but we already checked that responses are in order.
        // Even if the wall clock time has jumped, we want to use the latest time.
        [...]
      })
    }
  }
```

Preserved gossiped last-seen timestamp

Together, these behaviors can strand an otherwise usable peer address in `Failed` with an old or missing `last_seen`. Reconnection selection skips the entry, and later gossip may not make it eligible again.

Remediation

Update `MetaAddrChange::apply_to_meta_addr` so fresh `NewGossiped` information can refresh `untrusted_last_seen` for an already-attempted or failed address without resetting `last_connection_state`, `last_attempt`, or `last_failure`. Alternatively, add a bounded decay for `Failed` entries so a stale failure becomes eligible again after a defined interval.

Appendix A.

Vulnerability rating scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the general findings.

Severity	Description
Critical	Vulnerabilities that immediately result in a loss of availability, consensus divergence, or exploitable memory corruption with minimal preconditions.
High	Vulnerabilities that may result in consensus divergence or denial of service but require specific transaction or network conditions to exploit.
Medium	Vulnerabilities that may result in incorrect behavior or state divergence under specific configurations or edge cases.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.
Informational	Best practices to mitigate future security risks. These are classified as general findings.

Resolution status is one of ✓ Resolved, ○ Ack'd, or ✗ To-do.

Appendix B.

Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design, we aim to ensure that the underlying protocol is sound. For the Halo2 proving system and Orchard's circuits, this means the constraints faithfully capture the intended statement, so that no malicious prover can forge a valid proof or inflate shielded value. For Zebra, it means the consensus and networking rules preserve chain integrity and cannot be manipulated by adversarial peers to divide the network or exhaust resources. This requires a deep understanding of the cryptographic primitives and the protocol's game-theoretic and adversarial assumptions.

On the other hand, auditing the implementation requires a deep understanding of how these designs are realized in Rust. For Halo2 and Orchard, common implementation issues include underconstrained gates, exceptional cases in incomplete arithmetic, and mismatches between in-circuit and reference computations. For Zebra, they include panics on malformed input, denial-of-service vectors in the syncer and mempool, and flaws in peer reputation and inventory tracking.

As we approach any new target, we strive to comprehensively understand the codebase first. In our audits, we always approach targets with a team of auditors, allowing us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.