



# Zcash Zebra Security Assessment

Zcash Foundation  
Version 1.0 – June 27, 2023

©2023 – NCC Group

Prepared by NCC Group Security Services, Inc. for Zcash Foundation. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

## Prepared By

Aleksandar Kircanski  
Thomas Pornin  
Eli Sohl  
Kevin Henry  
Parnian Alimi

## Prepared For

Jack Gavigan  
Deirdre Connolly  
Teor  
Maria Pilar Guerra-Arias  
Arya Solhi  
Alfredo Garcia  
Marek Bielik

# 1 Executive Summary

---

## Synopsis

In Spring 2023, the Zcash Foundation engaged NCC Group to conduct a security assessment of the Zebrad application. Zebrad is a network client that participates in the Zcash consensus mechanism by validating blocks, maintaining the blockchain state (best chain and viable non-finalized chains), and gossiping blocks, transactions, and peer addresses. Five consultants performed the review, in a total of 60 person-days.

## Scope

The entire [Zebra](#) repository on branch `audit-v1.0.0-rc.0` was in scope, with the following modules highlighted as the main areas of focus: `zebra-chain`, `zebra-client`, `zebra-consensus`, `zebra-network`, `zebra-node-services`, `zebra-rpc`, `zebra-script`, `zebra-state`, `zebra-utils`. NCC Group also reviewed some issues that were fixed after the audit tag was created. The list of these Pull Requests along with some notes are included in the [Additional Code Changes](#) section as an appendix.

In addition, some of Zebra's dependencies were in scope. These were listed in the [zebra-dependencies-for-audit](#) document and are as follows:

- [ed25519-zebra](#)
- [zcash\\_proofs](#)
- [zcash\\_script](#)
- [redjubjub](#)
- [reddsa](#)

NCC Group's evaluation included the following 3 areas:

- **Block Validation:** This target involved ensuring that Zebrad closely follows the Zcash protocol<sup>1</sup> and Zcashd implementation in block validation. A block must be structurally valid when deserialized (`zebra-chain`), it must be semantically valid when its transactions' spending proofs and output commitments are validated (`zebra-consensus`), and it must be contextually valid when appended to the tip of the chain (`zebra-state`).
- **Cryptographic Dependencies:** This target included the items that were marked as needing *Full Audit* or *Partial Audit* in the [zebra-dependencies-for-audit](#) document. These libraries' primitives' implementations, namely signature schemes and batched zero-knowledge proof verifications, were reviewed for cryptographic vulnerabilities and potential consensus breaking oversights.
- **Peer-to-peer Network:** This target included network protocol handling, and peer discovery and peer-set maintenance. Each peer's state is isolated from other peers, and strategies for load balancing are implemented.

This review was performed primarily by manual code review.

## Limitations

The primary focus of this audit was the correctness and completeness of Zebrad as a Zcash chain verifier node and a network participant. As such, NCC group focused on matching the implementation to the Zcash protocol (at NU5 upgrade) and the scope did not include testing its compatibility with the Zcashd implementation.

## Key Findings

- **Failure to reject out of order address change requests** incorrectly alters the Address Book's state and opens the Zebra node to address book state manipulation attacks.

---

1. <https://zips.z.cash/protocol/nu5.pdf>



- 
- **Incomplete Zeroization of the Private Key** allows an attacker able to scavenge data from deallocated memory to possibly obtain enough information to reconstruct the private part of an Ed25519 key pair, and then forge signatures.
  - **Unbounded Rejection Sampling with Possibility of Panics** results in a fragile dummy Orchard note generation during mining.
  - **Uncaught Nonce Reuse and Fragile Nonce Cache Eviction** may prevent the self-connection detection from working as intended.
  - **Unenforced Constraint on Header Version in zcash\_serialize** may affect consensus or interoperability.

## Strategic Recommendations

Overall, the Zebra project is well documented and the implementation is well commented. NCC Group encourages the Zcash Foundation team to continue these practices as the library is developed and expanded.

A few instances of closed TODOs were found and reported during the assessment. These were thoroughly investigated and resolved where appropriate by the Zcash Foundation team.



## 2 Dashboard

### Target Data

Name	Zebrad
Type	Cryptography Libraries, Zcash Node
Platforms	Rust, C++


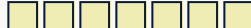

### Engagement Data

Type	Cryptography and Implementation Review
Method	Source Code Review, Dynamic Testing
Dates	2023-02-27 to 2023-05-12
Consultants	5
Level of Effort	60 person-days

### Targets

<a href="https://github.com/ZcashFoundation/zebra/tree/audit-v1.0.0-rc.0">https://github.com/ZcashFoundation/zebra/tree/audit-v1.0.0-rc.0</a>	A Zcash node implementation in Rust
<a href="https://github.com/ZcashFoundation/ed25519-zebra/tree/3.1.0/src">https://github.com/ZcashFoundation/ed25519-zebra/tree/3.1.0/src</a>	A Zcash-flavored Ed25519 for use in Zebra
<a href="https://github.com/zcash/librustzcash/tree/zcash_proofs-0.8.0/zcash_proofs/src">https://github.com/zcash/librustzcash/tree/zcash_proofs-0.8.0/zcash_proofs/src</a>	A zk-SNARK circuits implementation for Zcash with APIs for creating and verifying proofs. Review was limited to the proof parameter download code
<a href="https://github.com/ZcashFoundation/zcash_script/tree/v0.1.8/depend/zcash/src/script">https://github.com/ZcashFoundation/zcash_script/tree/v0.1.8/depend/zcash/src/script</a>	A Zcash script implementation. Review was limited to Zebra's use of the zcash_script crate as a dependency
<a href="https://github.com/ZcashFoundation/redjubjub/tree/0.5.0/src">https://github.com/ZcashFoundation/redjubjub/tree/0.5.0/src</a>	A minimal RedJubjub implementation for use in Zebra
<a href="https://github.com/ZcashFoundation/reddsa/tree/0.4.0/src">https://github.com/ZcashFoundation/reddsa/tree/0.4.0/src</a>	A minimal RedDSA implementation for use in Zebra

### Finding Breakdown

Critical issues	0
High issues	0
Medium issues	1 
Low issues	7 
Informational issues	8 
<b>Total issues</b>	<b>16</b>

### Category Breakdown




Data Exposure	2 
Data Validation	3 
Denial of Service	2 
Error Reporting	2 
Other	1 
Patching	1 



---

















## Category Breakdown


---

Security Improvement Opportunity	3	  
Session Management	2	 

## Component Breakdown

---

ed25519-zebra	2	 
zcash_proofs	1	
zebra	1	
zebra-chain	3	  
zebra-consensus	2	 
zebra-network	6	     
zebra-state	1	

 Critical     High     Medium     Low     Informational



### 3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Fragile State Transition During Address Book Update	Fixed	7DU	Medium
Inconsistent Error and Constraint Checks for Arithmetic Operations on Block Height	Fixed	XVE	Low
Incomplete Zeroization of the Private Key	Fixed	3WU	Low
Unbounded Rejection Sampling with Possibility of Panics	Fixed	DBV	Low
Uncaught Nonce Reuse and Fragile Nonce Cache Eviction	Fixed	MMC	Low
Power-of-Two-Choices Load Balancing May Deprioritize Honest Peers	Risk Accepted	6AN	Low
Unenforced Constraint on Header Version in zcash_serialize	Fixed	M2F	Low
Cargo Audit and RustSec Advisories	Partially Fixed	GCR	Low
Buffer Length Validation after Memory Allocation	Fixed	HV6	Info
Private Keys May Be Written to Log Files	Fixed	AQM	Info
Potential Panic on Integer Overflow when Hashing a Large Stream	Fixed	NQ6	Info
Off-by-One Errors and Inconsistent Usage of PARAM ETER_DOWNLOAD_MAX_RETRIES	Fixed	WVM	Info
Redundant Computation in Sapling and Orchard Note Validation	Fixed	MU2	Info
Off-by-One Error in zebra-network Retry Parameter	Fixed	VM7	Info
Incorrectly Disabled Consistency Check	Fixed	GHX	Info
Fragile Address Limit Implementation	Fixed	4FM	Info



## 4 Finding Details

Medium

# Fragile State Transition During Address Book Update

Overall Risk Medium

Impact Medium

Exploitability Medium

Finding ID NCC-E005955-7DU

Component zebra-network

Category Data Validation

Status Fixed

### Impact

Failure to reject out of order address change requests incorrectly alters the Address Book's state and opens the Zebra node to address book state manipulation attacks.

### Description

The `zebra_network`'s `AddressBook` update implementation uses `MetaAddrChange`'s `apply_to_meta_addr()` to update the entry's previous state to the received updated state. The `apply_to_meta_addr()` function validates the change against the previous state and optionally returns the new `MetaAddr`. If the received state is not the never-attempted state (the `else` condition on line 831) the current state is one of `{AttemptPending, Responded, Failed}`. In order to tolerate an address change request that is received out of order, the implementation picks the maximum of `{last_response, last_attempt, last_failure}` timestamps. Thus these timestamps will never revert to their previous values. However, independent of what the previous state was, on line 853, the new address state is returned. The `last_connection_state` records the outcome of local node's most recent communication attempt with this peer:

```
785 // Apply this change to a previous `MetaAddr` from the address book,
786 // producing a new or updated `MetaAddr`.
787 //
788 // If the change isn't valid for the `previous` address, returns `None`.
789 pub fn apply_to_meta_addr(&self, previous: impl Into<Option<MetaAddr>>) ->
↳ Option<MetaAddr> {
790     if let Some(previous) = previous.into() {
791         assert_eq!(previous.addr, self.addr(), "unexpected addr mismatch");
792
793         let previous_has_been_attempted = !
↳ previous.last_connection_state.is_never_attempted();
794         let change_to_never_attempted = self
795             .into_new_meta_addr()
796             .map(|meta_addr| meta_addr.last_connection_state.is_never_attempted())
797             .unwrap_or(false);
798
799         if change_to_never_attempted {
800             if previous_has_been_attempted {
801                 // Existing entry has been attempted, change is NeverAttempted
802                 // - ignore the change
803                 //
804                 // # Security
805                 //
806                 // Ignore NeverAttempted changes once we have made an attempt,
807                 // so malicious peers can't keep changing our peer connection order.
808                 None
```



```

809     } else {
810         // Existing entry and change are both NeverAttempted
811         // - preserve original values of all fields
812         // - but replace None with Some
813         //
814         // # Security
815         //
816         // Preserve the original field values for NeverAttempted peers,
817         // so malicious peers can't keep changing our peer connection order.
818         Some(MetaAddr {
819             addr: self.addr(),
820             services: previous.services.or_else(|| self.untrusted_services()),
821             untrusted_last_seen: previous
822                 .untrusted_last_seen
823                 .or_else(|| self.untrusted_last_seen()),
824             // The peer has not been attempted, so these fields must be None
825             last_response: None,
826             last_attempt: None,
827             last_failure: None,
828             last_connection_state: self.peer_addr_state(),
829         })
830     }
831 } else {
832     // Existing entry and change are both Attempt, Responded, Failed
833     // - ignore changes to earlier times
834     // - update the services from the change
835     //
836     // # Security
837     //
838     // Ignore changes to earlier times. This enforces the peer
839     // connection timeout, even if changes are applied out of order.
840     Some(MetaAddr {
841         addr: self.addr(),
842         // We want up-to-date services, even if they have fewer bits,
843         // or they are applied out of order.
844         services: self.untrusted_services().or(previous.services),
845         // Only NeverAttempted changes can modify the last seen field
846         untrusted_last_seen: previous.untrusted_last_seen,
847         // Since Some(time) is always greater than None, `max` prefers:
848         // - the latest time if both are Some
849         // - Some(time) if the other is None
850         last_response: self.last_response().max(previous.last_response),
851         last_attempt: self.last_attempt().max(previous.last_attempt),
852         last_failure: self.last_failure().max(previous.last_failure),
853         last_connection_state: self.peer_addr_state(),
854     })
855 }
856 } else {
857     // no previous: create a new entry
858     self.into_new_meta_addr()
859 }
860 }

```

Figure 1: `zebra-network/src/meta_addr.rs`

This can be leveraged into a replay/rewind attack in the following way. Consider a peer address that is marked as *Failed*, if the peer manages to send a *Ping* message before the connection shuts down, the address state will revert to *Responded*. Or, due to concurrent





---

address book updates, address state changes could execute out of order, which would accidentally revert a state to its previous value.<sup>2</sup>

## Recommendation

Update `apply_to_meta_addr()` to return `None` when the state transition is invalid, e.g., the request is received out-of-order and reverts the address state to a previous value.

## Location

*zebra-network/src/meta\_addr.rs*, line 853

## Retest Results

**2023-06-07 – Fixed**

This issue is resolved in [PR 6717](#). The fix ensures that:

1. An address state change that is received out-of-order, but not due to concurrency, is rejected.
2. A change that is received close to the latest update, due to concurrency, is only effective when it advances the state.
3. The remaining valid transitions are accepted.

---

2. When concurrent state updates are applied, the address state transitions must be resolved in the following order: `NeverAttemptedAlternate` < `NeverAttemptedGossiped` < `AttemptPending` < `Responded` < `Failed`.



# Inconsistent Error and Constraint Checks for Arithmetic Operations on Block Height

Overall Risk	Low	Finding ID	NCC-E005955-XVE
Impact	Medium	Component	zebra-chain
Exploitability	Low	Category	Error Reporting
		Status	Fixed

## Impact

An instance of the `Sub` function for `Height` fails to enforce the necessary constraints and will panic on overflow. This behavior is inconsistent with similar arithmetic functions in the same file.

## Description

Arithmetic operations `Add` and `Sub` are implemented for `Height` in `zebra-chain/src/block/height.rs` for both `Height` and `i32`. For example:

```
68 impl Add<Height> for Height {
69     type Output = Option<Height>;
70
71     fn add(self, rhs: Height) -> Option<Height> {
72         // We know that both values are positive integers. Therefore, the result is
73         // positive, and we can skip the conversions. The checked_add is required,
74         // because the result may overflow.
75         let height = self.0.checked_add(rhs.0)?;
76         let height = Height(height);
77
78         if height <= Height::MAX && height >= Height::MIN {
79             Some(height)
80         } else {
81             None
82         }
83     }
84 }
85
86 impl Sub<Height> for Height {
87     type Output = i32;
88
89     /// Panics if the inputs or result are outside the valid i32 range.
90     fn sub(self, rhs: Height) -> i32 {
91         // We construct heights from integers without any checks,
92         // so the inputs or result could be out of range.
93         let lhs = i32::try_from(self.0)
94             .expect("out of range input `self`: inputs should be valid Heights");
95         let rhs =
96             i32::try_from(rhs.0).expect("out of range input `rhs`: inputs should be valid
97             ↳ Heights");
98         lhs.checked_sub(rhs)
99             .expect("out of range result: valid input heights should yield a valid result")
100     }
}
```

Figure 2: `zebra-chain/src/block/height.rs`



---

The `Add` function handles overflow with an `Option` result, but the `Sub` function will panic. A second `Sub` function later in the file for the `i32` type returns an `Option` as well, making the above panic behavior an outlier. Panics should be used as a last resort, when there is no possibility of recovery. Otherwise, an attacker may attempt to intentionally trigger a panic as part of a denial-of-service attack.

Additionally, the `Sub` function **does not** enforce the same constraint checks shown on line 78 above, unlike the other functions in the same file. Therefore, even if the result of the operation does not overflow, it may result in an invalid `Height` that is less than the defined constant `pub const MIN: Height = Height(0);`.

## Recommendation

Rewrite the `Sub` function to return an `Option`, with a result of `None` if overflow or constraint violations occur.

## Location

*zebra-chain/src/block/height.rs*

## Retest Results

### 2023-05-02 – Fixed

This issue has been resolved incidentally as part of a larger code refactor, carried out in [PR 6330](#), which introduced a new `HeightDiff` type and rewrote `Height` arithmetic around these types. This change, while nontrivial, appears to resolve the issue, and the new code was not found to introduce any new issues.



# Incomplete Zeroization of the Private Key

Overall Risk	Low	Finding ID	NCC-E005955-3WU
Impact	High	Component	ed25519-zebra
Exploitability	Low	Category	Data Exposure
		Status	Fixed

## Impact

An attacker able to scavenge data from deallocated memory may obtain enough information to reconstruct the private part of an Ed25519 key pair, and then forge signatures.

## Description

Memory scavenging attacks apply to situations where an attacker can obtain a partial view of the memory contents of a target system after some private operations have been performed. *Zeroization* is the act of erasing secret data elements (in particular private cryptographic keys) before releasing the memory slot that holds them, so that the secret data does not linger in the address space of the program long after such data has been formally discarded (in the abstract machine model). In the `ed25519-zebra` crate, zeroization of private keys is ensured by implementing the `Zeroize` trait on the `SigningKey` type, which contains private keys:

```
impl zeroize::Zeroize for SigningKey {
    fn zeroize(&mut self) {
        self.seed.zeroize();
        self.s.zeroize()
    }
}
```

However, an Ed25519 private key really consists of three elements: a `seed` (of length 32 bytes), and two extra elements derived from the seed through a hash function: the secret scalar `s`, and the secret `prefix` value. The `prefix` is used, conjointly with the message to sign, to deterministically derive the per-signature nonce, as specified in [RFC 8032](#). The `Zeroize` implementation, shown above, erases only the `seed` and the secret scalar `s`, but not the `prefix`.

If an attacker learns the value of the `prefix`, then, for any signature generated with the private key, that attacker can recompute the per-signature nonce value with the deterministic process used by the signer, and then immediately recover the secret scalar by applying the signature generation equation (using  $s = (S - r)/k \bmod L$ , in the notations of RFC 8032: `k` is the public “challenge” of the signature, `r` is the per-signature nonce, `S` is the second half of the signature value, and `L` is the prime group order). Therefore, if memory scavenging attacks are considered to be a plausible enough threat in a given usage context that zeroization of private data should be done, then not zeroizing the `prefix` is a potentially severe issue.

## Recommendation

The `prefix` field of `SigningKey` should be zeroized in the implementation of the `zeroize()` function.



---

## Location

*ed25519-zebra/src/signing\_key.rs*, line 109

## Retest Results

**2023-05-04 – Fixed**

NCC Group reviewed [PR 73](#) and found that it implements the recommendation by replacing the custom zeroizer function with a derived zeroizer on the full `SigningKey` struct.



# Unbounded Rejection Sampling with Possibility of Panics

Overall Risk	Low	Finding ID	NCC-E005955-DBV
Impact	Low	Component	zebra-chain
Exploitability	Low	Category	Security Improvement Opportunity
		Status	Fixed

## Impact

The `zebra-chain` module implements random number generation for some types which are primarily used for testing, without guarding against possible panics.

## Description

The use of `fill_bytes()` in the code snippet below (from `zebra-chain/src/sapling/keys.rs`) can potentially cause panics.<sup>3</sup> It is recommended to use `try_fill_bytes()` to catch the random number generator's failures.<sup>4</sup> In addition, the number of sampling retries by the loop has to be bounded.

```
impl Diversifier {
    /// Generate a new _Diversifier_ that has already been confirmed
    /// as a preimage to a valid diversified base point when used to
    /// derive a diversified payment address.
    ///
    /// <https://zips.z.cash/protocol/protocol.pdf#saplingkeycomponents>
    /// <https://zips.z.cash/protocol/protocol.pdf#concretediversifyhash>
    pub fn new<T>(csprng: &mut T) -> Self
    where
        T: RngCore + CryptoRng,
    {
        loop {
            let mut bytes = [0u8; 11];
            csprng.fill_bytes(&mut bytes);

            if diversify_hash(bytes).is_some() {
                break Self(bytes);
            }
        }
    }
}
```

There is one instance where this can potentially be problematic for a full node with mining support. Generating a `random Rho` for an Orchard note uses the same unchecked RNG API.

3. [https://docs.rs/rand/latest/rand/trait.RngCore.html#tymethod.fill\\_bytes](https://docs.rs/rand/latest/rand/trait.RngCore.html#tymethod.fill_bytes)

4. For instance `OsRng` can, in rare occasions, fail on some platforms, and `thread_rng` might return a warning if it fails to reseed itself. See <https://rust-random.github.io/book/guide-err.html> for more details.



---

A node with mining capabilities might use this API to generate a dummy Orchard input note<sup>5</sup> as a nullifier for the output note when creating a miner's reward note:

```
impl Rho {
    pub fn new<T>(csprng: &mut T) -> Self
    where
        T: RngCore + CryptoRng,
    {
        let mut bytes = [0u8; 32];
        csprng.fill_bytes(&mut bytes);

        Self(extract_p(pallas::Point::from_bytes(&bytes).unwrap()))
    }
}
```

## Recommendation

Consider replacing the `fill_bytes()` usage with `try_fill_bytes()` and properly handle its possible failure.

## Location

- [zebra-chain/src/orchard/keys.rs](#), line 110
- [zebra-chain/src/sapling/keys.rs](#), line 189
- [zebra-chain/src/orchard/commitment.rs](#), line 30
- [zebra-chain/src/sapling/commitment.rs](#), line 42
- [zebra-chain/src/orchard/note.rs](#), line 30
- [zebra-chain/src/orchard/note.rs](#), line 65

## Retest Results

### 2023-05-02 – Fixed

NCC Group reviewed [PR 6385](#) and found it adequately addresses this issue. Per the recommendation, `try_fill_bytes` is now used, and appropriate error types have been introduced.

---

5. <https://zips.z.cash/protocol/nu5.pdf#orcharddummynotes>



# Uncaught Nonce Reuse and Fragile Nonce Cache Eviction

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E005955-MMC

Component zebra-network

Category Session Management

Status Fixed

## Impact

Failure to check if the nonce cache already contains the new nonce may prevent the self-connection detection from working as intended. Incorrectly evicting an entry from the nonce cache will have the same effect.

## Description

The function `negotiate_version()` in `zebra-network/src/peer/handshake.rs` is used to negotiate the network version used when connecting to a new peer. In order to match outgoing messages with incoming responses, a nonce is included, and cached locally to help identify self-connection attempts:

```
584 // Create a random nonce for this connection
585 let local_nonce = Nonce::default();
586 // # Correctness
587 //
588 // It is ok to wait for the lock here, because handshakes have a short
589 // timeout, and the async mutex will be released when the task times
590 // out.
591 nonces.lock().await.insert(local_nonce);
```

Figure 3: `handshake.rs`

It was observed that the return value of the `insert` function is not checked. This function returns `true` if the value was successfully inserted, and `false` if the value was already contained in the set. Therefore, nonce reuse could be detected at this step by checking the return value of this function. The above freshly generates each nonce via `Nonce::default()`, so the probability of a collision is negligible, but it is nevertheless recommended to check the result of the insert operation as a precaution.

Later, in the same function, the following code handles received nonces:

```
677 // Check for nonce reuse, indicating self-connection
678 //
679 // # Correctness
680 //
681 // We must wait for the lock before we continue with the connection, to avoid
682 // self-connection. If the connection times out, the async lock will be
683 // released.
684 let nonce_reuse = {
685     let mut locked_nonces = nonces.lock().await;
686     let nonce_reuse = locked_nonces.contains(&remote_nonce);
687     // Regardless of whether we observed nonce reuse, clean up the nonce set.
688     locked_nonces.remove(&local_nonce);
689     nonce_reuse
689 }
```





```
690     };
691     if nonce_reuse {
692         Err(HandshakeError::NonceReuse)?;
693     }
```

Figure 4: [handshake.rs](#)

If the incoming message contains a nonce currently in the cache, that means it corresponds to a negotiation version message originating from itself, indicating a self-connection attempt. When this occurs, an error is returned and the nonce associated with the request is evicted from the cache.

Earlier, this finding highlighted a potential case where nonces could be re-used without correct detection. If such a nonce is evicted after the first connection attempt then a second connection attempt might succeed. Building on this observation, a malicious party may attempt to replay messages or craft messages containing an observed nonce, thereby causing the nonce to be evicted incorrectly. Subsequently, a self-connection attempt would not be correctly identified.

Based on the above, it is recommended that the approach to nonce handling in this use case be re-evaluated to ensure the intended security goals are met. It may be safer to cache all nonces for a set duration to ensure that malicious or incorrect behavior cannot force nonce eviction from the cache.

## Recommendation

- Consider adding an explicit check for nonce reuse when adding values to the cache and returning `HandshakeError::NonceReuse` as necessary.
- Consider caching nonce entries for a longer period of time to ensure that nonces are not prematurely evicted from the cache.

## Location

[zebra-network/src/peer/handshake.rs](#)

## Retest Results

### 2023-05-04 – Fixed

NCC Group reviewed [PR 6410](#) and found that it introduces a check on the result of nonce insertion to cover the rare edge case of nonce collisions. The logic around nonce reuse was also reworked to prevent malicious nonce removal. These changes appear to resolve the issue.



# Power-of-Two-Choices Load Balancing May Deprioritize Honest Peers

<b>Overall Risk</b>	Low	<b>Finding ID</b>	NCC-E005955-6AN
<b>Impact</b>	Medium	<b>Component</b>	zebra-network
<b>Exploitability</b>	Low	<b>Category</b>	Denial of Service
		<b>Status</b>	Risk Accepted

## Impact

Attackers with strong levels of influence over the network may be able to deploy low-latency peers which are disproportionately prioritized by power-of-two-choices routing, and may be able to decrease the prioritization of honest peers on the network by artificially inflating their latencies. This would result in situations where even though a peer retains connections to honest peers, it may deprioritize these honest peers to the point where the net effect is comparable to full compromise.

## Description

Zebra utilizes “power of two choices” (p2c) load-balancing,<sup>6</sup> which routes outbound requests by first randomly sampling a set of known peers and then sending the request to the lowest-latency peer in the set. While this certainly behaves well under normal conditions and has attractive worst-case bounds,<sup>7</sup> it also introduces risks against attackers with significant influence over the network.

An ISP-level attacker, for instance, could deploy peers with exceptionally low latency, and could even artificially inflate other peers’ latencies, thereby increasing their own level of influence over the network. Such an attack could be difficult to detect.

For instance, in the case of an attempted eclipse attack, suppose a peer’s set of outbound connections is 80% compromised, with only 20% of their remote peers being honest. Ordinarily, such a peer would still be connected to the honest part of the network, and would not be considered compromised. However, suppose further that using p2c routing, this peer chooses two remote peers as candidates for an outbound request. The likelihood of *both* these peers being honest is only 4%, and in all other cases it is expected that the attacker’s low-latency remote peer will be prioritized over a randomly sampled honest peer. If the sample size is larger, the probability of reaching an honest peer is even lower.

Thus, even though the peer retains honest connections, it may deprioritize these honest peers to the point where the net effect is comparable to full compromise.

## Recommendation

There are several ways this effect might be mitigated, and the following is just one suggestion. Latencies below a certain dynamically-determined threshold (e.g. mean or median latency) could be treated as equivalent, and the second choice could fall back to random selection as a tiebreaker. This would retain the desired behavior of deprioritizing abnormally high-latency peers while also avoiding prioritization of abnormally low-latency peers.

6. For full details, see <https://www.eecs.harvard.edu/~michaelm/postscripts/mythesis.pdf>.

7. For an accessible discussion, see <http://www.eecs.harvard.edu/~michaelm/postscripts/handbook2001.pdf>.



---

This solution comes with its own caveats: for instance, an attacker could instead flood the network with *high*-latency peers, attempting to artificially inflate median latency and thus ensure that most or all honest peers' latencies fall below the median cutoff. In this case, the algorithm would reduce to random selection, with effectively no load-balancing.

However, this would still be a net improvement, because an attack which inhibits load-balancing would seem to be significantly less severe than an attack which inhibits network connectivity.

## Location

zebra-network

## Retest Results

### 2023-05-04 – Not Fixed

The Zebra team has decided not to resolve this finding at this time. Discussion from the team is provided in [issue 6343](#).

## Client Response

We have hardened Zebra's peer set by limiting the number of connected peers, randomizing peer selection, and routing block and transaction downloads to peers that have advertised that inventory. We have considered other alternative fixes, but the risks outweighed the potential benefits. Peer selection in distributed networks is an open research problem, and we look forward to improvements in this area.



# Unenforced Constraint on Header Version in zcash\_serialize

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E005955-M2F

Component zebra-chain

Category Data Validation

Status Fixed

## Impact

Zebra assumes all serialization/deserialization functions cannot produce invalid data. Unenforced constraints violate this assumption and may affect consensus or interoperability.

## Description

Zebra implements the `ZCashSerialize` trait for consensus-critical data.

```

19 pub trait ZcashSerialize: Sized {
20     /// Write `self` to the given `writer` using the canonical format.
21     ///
22     /// This function has a `zcash_` prefix to alert the reader that the
23     /// serialization in use is consensus-critical serialization, rather than
24     /// some other kind of serialization.
25     ///
26     /// Notice that the error type is [std::io::Error]; this indicates that
27     /// serialization MUST be infallible up to errors in the underlying writer.
28     /// In other words, any type implementing `ZcashSerialize` must make illegal
29     /// states unrepresentable.
30     fn zcash_serialize<W: io::Write>(&self, writer: W) -> Result<(), io::Error>;

```

Figure 5: `zebra-chain/src/serialization/zcash_serialize.rs`

Note the highlighted requirement that illegal states must be unrepresentable.

The function `zcash_deserialize()` for `Header` contains historical information regarding the version field of the header. In particular, it specifies that the only valid version number is 4, but that incorrect implementations in the past resulted in several blocks with an incorrect value. The function has some special case handling for this, as well as an explicit check that the parsed version is greater than or equal to 4:

```

75     // # Consensus
76     //
77     // > The block version number MUST be greater than or equal to 4.
78     //
79     // https://zips.z.cash/protocol/protocol.pdf#blockheader
80     if version < 4 {
81         return Err(SerializationError::Parse("version must be at least 4"));
82     }

```

Figure 6: `zebra-chain/src/block/serialize.rs`

This is similarly captured in other annotations for the `version` field, where it is specified that:



---

The only constraint is that it must be at least 4 when interpreted as an `i32`.

In comparison, the corresponding `zcash_deserialize()` function **does not** perform validation on the version field:

```
28     fn zcash_serialize<W: io::Write>(&self, mut writer: W) -> Result<(), io::Error> {
29         writer.write_u32::<LittleEndian>(self.version)?;
30         self.previous_block_hash.zcash_serialize(&mut writer)?;
31         writer.write_all(&self.merkle_root.0[..])?;
32         writer.write_all(&self.commitment_bytes[..])?;
33         writer.write_u32::<LittleEndian>(
34             self.time
35                 .timestamp()
36                 .try_into()
37                 .expect("deserialized and generated timestamps are u32 values"),
38         )?;
39         writer.write_u32::<LittleEndian>(self.difficulty_threshold.0)?;
40         writer.write_all(&self.nonce[..])?;
41         self.solution.zcash_serialize(&mut writer)?;
42         Ok(())
43     }
```

Figure 7: `zebra-chain/src/block/serialize.rs`

No instances within the codebase will currently set this value incorrectly, but it appears to be a violation of the specified constraint. It is possible to construct a `Header` such that the output of `zcash_serialize` will not correctly deserialize within Zebra.

## Recommendation

Align the constraint checks for the version field such that any serialized `Header` will correctly deserialize without an error.

## Location

`zebra-chain/src/block/serialize.rs`

## Retest Results

2023-05-04 – Fixed

NCC Group reviewed [PR 6475](#) and found that it resolves this issue by refactoring the constraint checks into a function called on both serialization and deserialization, in alignment with the recommendation.



# Cargo Audit and RustSec Advisories

<b>Overall Risk</b>	Low	<b>Finding ID</b>	NCC-E005955-GCR
<b>Impact</b>	Low	<b>Component</b>	zebra
<b>Exploitability</b>	Undetermined	<b>Category</b>	Patching
		<b>Status</b>	Partially Fixed

## Impact

Failure to address RustSec advisories may leave publicly disclosed vulnerabilities in the project and may damage the reputation of the project.

## Description

The `cargo audit` tool can be used to check a project's dependency tree against published RustSec advisories. Running `cargo audit` on the `audit-v1.0.0-rc.0` branch results in 6 known vulnerable crates and 6 warnings:

- Vulnerabilities:
  - `time 0.1.44` - Potential segfault in the time crate
  - `owning_ref 0.4.1` - Multiple soundness issues in `owning_ref`
  - `git2 0.14.4` - `git2` Rust package suppresses ssh host key checking
  - `libgit2-sys 0.13.4+1.4.2` - `git2` does not verify SSH keys by default
  - `remove_dir_all 0.5.3` - Race Condition Enabling Link Following and Time-of-check Time-of-use (TOCTOU)
  - `tokio 1.21.2` - `reject_remote_clients` Configuration corruption
- Warnings:
  - `ansi_term 0.11.0` - `ansi_term` is Unmaintained
  - `ansi_term 0.12.1` - `ansi_term` is Unmaintained
  - `directories 4.0.1` - `directories` is unmaintained, use `directories-next` instead
  - `dirs 4.0.0` - `dirs` is unmaintained, use `dirs-next` instead
  - `mach 0.3.2` - `mach` is unmaintained
  - `net2 0.2.37` - `net2` crate has been deprecated; use `socket2` instead

The `audit-v1.0.0-rc.0` branch was frozen approximately two months prior to the writing of this finding. On the `main` branch of the project ([commit 45a96b5](#) at the time of writing), only the two most recently reported vulnerabilities are present:

- Vulnerabilities (`main` / [45a96b5](#)):
  - `time 0.1.44` - Potential segfault in the time crate
  - `owning_ref 0.4.1` - Multiple soundness issues in `owning_ref`

The 6 warnings are also present.

In general, it is recommended to keep all crates up-to-date, particularly when known advisories are relevant. Even if the identified vulnerabilities do not affect Zebra directly, failure to respond to advisories and update dependencies may affect the reputation of the project.

## Recommendation

Ensure dependencies are regularly audited for RustSec advisories using a tool such as `cargo audit` or `cargo deny`, and automate the detection and response to future



---

advisories to the greatest extent possible, such as the automatic filing of a security issue for review.

## Retest Results

### 2023-05-04 – Partially Fixed

NCC Group reviewed [PR 6217](#) which closes the GitHub issue associated with this finding. This pull request introduces several version bumps and updates some documentation. The noted vulnerable crates `time` and `owning_ref` are not updated, and `cargo audit` continues to flag them. However, discussion on GitHub [issue 6391](#) indicates that the Zebra team does not believe these vulnerabilities impact Zebra's security in release builds.

It was also noted that GitHub Dependabot was recently updated in [PR 6508](#) to use the correct label when flagging dependency updates. Additionally, [PR 6657](#) updated the release process documentation to explicitly include checkboxes for steps relating to dependency update and running `cargo update`. Provided the process is followed, the likelihood of missed updates should be minimized.

The release process includes a step specifying “If needed, update `deny.toml`”. During the re-test, it was observed that `deny.toml` contains several exceptions, with justification, for multiple version detection. Many suggest that version numbers should be updated in the future, but this work is not explicitly tracked elsewhere. Thus, the basic instructions to update `deny.toml` if needed may not make the work required in this task clear. The documented process could be strengthened with more detailed instructions or requirements.

## Client Response

We have updated our release checklist for a human release engineer to check for security updates or other dependency upgrades that have not been applied by our dependency update automation.



# Buffer Length Validation after Memory Allocation

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-HV6
<b>Impact</b>	None	<b>Component</b>	zebra-network
<b>Exploitability</b>	None	<b>Category</b>	Security Improvement Opportunity
		<b>Status</b>	Fixed

## Impact

Failure to validate buffers' length during parsing leads to unnecessary memory allocations.

## Description

The `AddrV2` type's deserialization API validates the length of the (variable-length) address after the address is read into a buffer. An unexpectedly large address can temporarily lead to a large memory allocation on the heap. It is recommended to inspect the length of the address and validate it before reading the `addr` in the code snippet below:

```
// > CompactSize      The length in bytes of addr.
// > uint8[sizeAddr]  Network address. The interpretation depends on networkID.
let addr: Vec<u8> = (&mut reader).zcash_deserialize_into()?;

// > uint16 Network port. If not relevant for the network this MUST be 0.
let port = reader.read_u16::<BigEndian>()?;

if addr.len() > MAX_ADDR_V2_ADDR_SIZE {
    return Err(SerializationError::Parse(
        "addr field longer than MAX_ADDR_V2_ADDR_SIZE in addrv2 message",
    ));
}
```

Note that the deserialization logic ensures that the length of the `addr` vector is smaller than `MAX_U8_ALLOCATION` (currently 2097147 bytes) which is significantly higher than `MAX_ADDR_V2_ADDR_SIZE` (currently set to 512).

A version 2 address can be received as part of an unsolicited [Addr message](#), and it would be prudent to reject malformed messages with minimum cost.

## Recommendation

Ensure that the length of the variable-length arrays are checked before memory allocation whenever possible.

## Location

[zebra-network/src/protocol/external/addr/v2.rs](#)

## Retest Results

2023-05-03 – Fixed

NCC Group reviewed [PR 6320](#) and found that it follows the provided recommendation. The size of the address is now read and validated prior to allocating memory for the address itself.





# Private Keys May Be Written to Log Files

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-AQM
<b>Impact</b>	High	<b>Component</b>	ed25519-zebra
<b>Exploitability</b>	None	<b>Category</b>	Data Exposure
		<b>Status</b>	Fixed

## Impact

Private key elements might leak to log files through the implemented `Debug` trait, if used in an application with extensive logging enabled for issue troubleshooting.

## Description

The `SigningKey` type contains an Ed25519 key pair. An implementation of the `Debug` trait for `SigningKey` is provided; that implementation converts the elements of the private key to hexadecimal strings:

```
impl core::fmt::Debug for SigningKey {
    fn fmt(&self, fmt: &mut core::fmt::Formatter) -> core::fmt::Result {
        fmt.debug_struct("SigningKey")
            .field("seed", &hex::encode(&self.seed))
            .field("s", &self.s)
            .field("prefix", &hex::encode(&self.prefix))
            .field("vk", &self.vk)
            .finish()
    }
}
```

This is a somewhat dangerous practice because it tends to lead to situations where an application, using the library and compiled with debug logs for issue troubleshooting, ends up leaking the private key into log files. This situation does not currently arise in Zebra, but the `ed25519-zebra` crate is a general-purpose library that could be used in other applications with more extensive logging.

## Recommendation

Only public data should be included in `Debug` output. In this case, the `vk` field contains the public key corresponding to that private key, and that value is enough to unambiguously designate a unique private key; the other fields should not be included in that output.

## Location

[ed25519-zebra/src/signing\\_key.rs](#), line 26

## Retest Results

2023-05-04 – Fixed

NCC Group reviewed [PR 70](#) and found that it resolves this issue in the recommended manner.



# Potential Panic on Integer Overflow when Hashing a Large Stream

**Overall Risk** Informational

**Impact** Low

**Exploitability** None

**Finding ID** NCC-E005955-NQ6

**Component** zcash\_proofs

**Category** Denial of Service

**Status** Fixed

## Impact

When hashing more than 4 gigabytes of data on a 32-bit architecture, the accounting of the amount of processed data hits an integer overflow condition that will trigger a panic if the code is compiled in debug mode.

## Description

The `HashReader` structure is defined in `zcash_proofs/src/hashreader.rs`; it wraps around a hash function (BLAKE2b instance) and hashes a long stream of bytes as they flow. The structure also keeps track of how many bytes have been processed so far; the `byte_count` field maintains that information, and has type `usize`:

```
/// Abstraction over a reader which hashes the data being read.
pub struct HashReader<R: Read> {
    reader: R,
    hasher: State,
    byte_count: usize,
}
```

On 32-bit architectures, `usize` has size 32 bits, and thus any input larger than about 4.29 gigabytes will overflow that counter. In particular, the addition on [line 51](#) may then trigger a panic (if the code was compiled in debug mode), or silently truncate the count to its low 32 bits (if compilation used release mode). This issue cannot be triggered with the Zebra implementation in its current state, since hashing is performed only on files whose size has been explicitly verified to match the expected size for parameter files, with a maximum of about 0.73 gigabytes (for Sprout Groth16 parameters).

## Recommendation

Defining `byte_count` to have type `u64` (i.e. with the same range as what is currently obtained with `usize` on a 64-bit architecture) would make the implementation more robust with regard to future development and protocol versions.

## Location

`zcash_proofs/src/hashreader.rs`, [line 14](#)

## Retest Results

2023-05-04 – Fixed

NCC Group reviewed [PR 805](#) and found that it implements the recommendation of changing `byte_count`'s type to `u64`.



# Off-by-One Errors and Inconsistent Usage of PARAMETER\_DOWNLOAD\_MAX\_RETRIES

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-WVM
<b>Impact</b>	None	<b>Component</b>	zebra-consensus
<b>Exploitability</b>	None	<b>Category</b>	Error Reporting
		<b>Status</b>	Fixed

## Impact

The `PARAMETER_DOWNLOAD_MAX_RETRIES` may not behave as expected due to a potential off-by-one error.

## Description

The parameter `PARAMETER_DOWNLOAD_MAX_RETRIES` is defined in [zebra-consensus/src/primitives/groth16/params.rs](#):

```
18 /// The maximum number of times to retry download parameters.
19 ///
20 /// Zebra will retry to download Sprout of Sapling parameters only if they
21 /// failed for whatever reason.
22 pub const PARAMETER_DOWNLOAD_MAX_RETRIES: usize = 3;
```

Figure 8: [params.rs](#)

As noted, this parameter represents the number of times to retry the download of groth16 parameters. However, later in the same file the following description is given:

```
63 impl Groth16Parameters {
64     /// Download if needed, cache, check, and load the Sprout and Sapling Groth16
65     /// ↳ parameters.
66     ///
67     /// # Panics
68     ///
69     /// If the parameters were downloaded to the wrong path.
70     /// After `PARAMETER_DOWNLOAD_MAX_RETRIES` failed download attempts.
71     /// If the downloaded or pre-existing parameter files are invalid.
```

Figure 9: [params.rs](#)

Here, the parameter is claimed to specify the number of download attempts, not the number of *retry* attempts. The implementation matches the described behavior and uses this parameter to cap the total number of download attempts; see `retry_download_sapling_parameters()`:

```
132     let mut retries = 0;
133     while let Err(error) = Groth16Parameters::download_sapling_parameters_once(
134         sapling_spend_path,
135         sapling_output_path,
136     ) {
137         retries += 1;
138         if retries >= PARAMETER_DOWNLOAD_MAX_RETRIES {
139             panic!(
140                 "error downloading Sapling parameter files after {} retries. {:?}",
141                 ↳ {},
142             );
```



```

141         PARAMETER_DOWNLOAD_MAX_RETRIES,
142         error,
143         Groth16Parameters::failure\_hint\(\),
144     );

```

Figure 10: [params.rs](#)

The same behavior is implemented for `retry_download_sprout_parameters()`. Documentation for both of these functions is inconsistent:

```

120     /// Download Sapling parameters and retry [PARAMETER_DOWNLOAD_MAX_RETRIES] if it
    ↪ fails.
121     ///
122     /// # Panics
123     ///
124     /// If the parameters were downloaded to the wrong path.
125     /// After PARAMETER_DOWNLOAD_MAX_RETRIES failed download attempts.

```

Figure 11: [params.rs](#)

For example, consider a case where `PARAMETER_DOWNLOAD_MAX_RETRIES = 1`. Then this comment specifies the following:

1. “Download Sapling parameters and retry 1 if it fails.” – This should likely read “1 time(s)” if it fails (currently missing the word “*time*”). This description matches the implied behavior based on the parameter name.
2. Panics “After 1 failed download attempts” – In other words, it *will not* retry 1 time. This description matches the implemented behavior.

For consistency, the parameter should be uniformly treated as the maximum number of download attempts, or the maximum number of retry attempts, and treated appropriately in all documentation and code.

## Recommendation

Revise documentation, parameter names, and implemented behavior to correctly capture the behavior of `PARAMETER_DOWNLOAD_MAX_RETRIES`.

## Location

[zebra-consensus/src/primitives/groth16/params.rs](#)

## Retest Results

2023-05-03 – Fixed

NCC Group reviewed [PR 6464](#) and found that the documentation and implemented behavior have been updated as recommended, resolving this issue.



# Redundant Computation in Sapling and Orchard Note Validation

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-MU2
<b>Impact</b>	Low	<b>Component</b>	zebra-consensus
<b>Exploitability</b>	None	<b>Category</b>	Other
		<b>Status</b>	Fixed

## Impact

During Sapling and Orchard output validation, one redundant elliptic curve scalar multiplication is performed. It may be considered for removal for efficiency purposes.

## Description

In order to verify that Sapling and Orchard outputs are decryptable and consistent with ZIP 212 rules, Zebra uses the `zcash_primitives` crate's `try_sapling_output_recovery` function. This function is called, for example, when coinbase transaction outputs are [validated](#) to adhere to the rules specified in ZIP 212.

Once decrypted, an output's note is [parsed](#). An ephemeral key validation function is passed as a lambda:

```
fn parse_note_plaintext_without_memo_ovk(
    &self,
    pk_d: &Self::DiversifiedTransmissionKey,
    esk: &Self::EphemeralSecretKey,
    ephemeral_key: &EphemeralKeyBytes,
    plaintext: &NotePlaintextBytes,
) -> Option<Self::Note, Self::Recipient> {
    sapling_parse_note_plaintext_without_memo(self, &plaintext.0, |diversifier| {
        if (diversifier.g_d()? * esk).to_bytes() == ephemeral_key.0 {
            Some(*pk_d)
        } else {
            None
        }
    })
}
```

The check implemented by the highlighted lambda function is mandated by ZIP 212. A few lines below, the `check_note_validity` function is [called](#):

```
fn check_note_validity<D: Domain>(
    note: &D::Note,
    ephemeral_key: &EphemeralKeyBytes,
    cmstar_bytes: &D::ExtractedCommitmentBytes,
) -> NoteValidity {
    if &D::ExtractedCommitmentBytes::from(&D::cmstar(note)) == cmstar_bytes {
        if let Some(derived_esk) = D::derive_esk(note) {
            if D::epk_bytes(&D::ka_derive_public(note, &derived_esk))
                .ct_eq(ephemeral_key)
                .into()
            {
                NoteValidity::Valid
            } else {
                NoteValidity::Invalid
            }
        }
    }
}
```



```
    NoteValidity::Invalid
  }
} else {
    // Before ZIP 212
    NoteValidity::Valid
  }
} else {
    // Published commitment doesn't match calculated commitment
    NoteValidity::Invalid
  }
}
```

The validation highlighted in the first code snippet happens regardless of whether ZIP 212 is activated or not. This is also what the original Zcash client does; see [zcash/Note.cpp](#). The validation highlighted in the second code snippet aims to only validate the public key post-ZIP 212 and is likely meant to be a blanket end-of-function validation helper. This helper includes an ECC point multiplication and as such is not inexpensive.

## Recommendation

It appears that the highlighted check inside the `check_note_validity` function overlaps with the previous validation, and, as such, may be removed.

## Retest Results

### 2023-06-07 – Fixed

This issue has been resolved in [PR 848](#) in the `librustzcash` crate and [PR 394](#) in the `orchard` crate. This issue will be resolved in Zebra, once it upgrades to using these crates' new releases as dependencies.



# Off-by-One Error in zebra-network Retry Parameter

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-VM7
<b>Impact</b>	None	<b>Component</b>	zebra-network
<b>Exploitability</b>	None	<b>Category</b>	Security Improvement Opportunity
		<b>Status</b>	Fixed

## Impact

The parameter `MAX_SINGLE_PEER_RETRIES` may not behave as expected due to an off-by-one error.

## Description

Finding "Off-by-One Errors and Inconsistent Usage of `PARAMETER_DOWNLOAD_MAX_RETRIES`" documented an off-by-one issue with retry behavior in `zebra-consensus`. This finding documents a similar issue in `zebra-network`. Note that this does not represent a security issue or vulnerability, outside of potential user error during configuration.

The following code is part of peer DNS resolution, where each peer is resolved individually, and the complete process is repeated if no resolution is successful. The parameter `MAX_SINGLE_PEER_RETRIES` implies that it dictates the number of *retry* attempts for a given peer.

```
161 /// The number of times Zebra will retry each initial peer's DNS resolution,
162 /// before checking if any other initial peers have returned addresses.
163 const MAX_SINGLE_PEER_RETRIES: usize = 1;
```

Figure 12: `zebra-network/src/config.rs`

```
161 // We retry each peer individually, as well as retrying if there are
162 // no peers in the combined list. DNS failures are correlated, so all
163 // peers can fail DNS, leaving Zebra with a small list of custom IP
164 // address peers. Individual retries avoid this issue.
165 let peer_addresses = peers
166     .iter()
167     .map(|s| Config::resolve_host(s, MAX_SINGLE_PEER_RETRIES))
168     .collect::<futures::stream::FuturesUnordered<_>>()
169     .concat()
170     .await;
```

Figure 13: `zebra-network/src/config.rs`

The `retry` function loops using `for retry_count in 1..=max_retries`, meaning that the value `MAX_SINGLE_PEER_RETRIES` will result in 1 single attempt and not 1 retry. Therefore, this parameter may be better named as `MAX_SINGLE_PEER_ATTEMPTS` or the behavior could be updated to loop one additional time.

## Recommendation

Ensure the naming of the parameter accurately reflects its usage. Either rename the parameter to reflect the maximum number of connection attempts or revise the behavior to reflect its current name.



---

## Location

*zebra-network/src/config.rs*

## Retest Results

**2023-05-04 – Fixed**

NCC Group reviewed [PR 6460](#) and found that the recommendation has been followed: in fact, the parameter has been renamed *and* the maximum number of attempts has been updated. These changes bring the name and behavior into alignment, resolving this issue. The DNS retry loop has also been expanded and now provides more detailed log messages.





# Incorrectly Disabled Consistency Check

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-GHX
<b>Impact</b>	Low	<b>Component</b>	zebra-state
<b>Exploitability</b>	Undetermined	<b>Category</b>	Data Validation
		<b>Status</b>	Fixed

## Impact

Disabling checks to pass incorrect tests rather than fixing the tests themselves may prevent the detection of bugs or other incorrect behavior.

## Description

The function `fetch_sprout_final_treestates()` in [zebra-state/src/service/check/anchors.rs](#) contains a block of commented code that performs an assertion that “that roots match the fetched tree during tests”. The annotations suggest that this check was disabled due to bad test data:

```
158     /* TODO:
159        - fix tests that generate incorrect root data
160        - assert that roots match the fetched tree during tests
161        - move this CPU-intensive check to sprout_anchors_refer_to_treestates()
162     assert_eq!(
163         input_tree.root(),
164         joinsplit.anchor,
165         "anchor and fetched input tree root did not match:\n|
166         anchor: {anchor:?}",
167         input_tree.root: {input_tree_root:?}",
168         input_tree: {input_tree:?}",
169         anchor = joinsplit.anchor
170     );
171     */
```

Figure 14: [zebra-state/src/service/check/anchors.rs](#)

As is, the code suggests that a required safety check has been disabled to prevent tests from failing, which prevents the test from being run in production as well. The preferred solution would be one of:

1. Fix the tests such that the check can remain enabled.
2. Disable this code only when tests are running.

Initial discussions with the Zebra team suggested that the latter approach would be preferable. Furthermore, the comment suggests that this check should be refactored and located elsewhere in the code. A task to track this change should be documented.

## Recommendation

1. Re-enable the check and either add a flag to disable it during tests, or update tests to correctly pass when the check is in place.
2. Refactor the code as recommended in the comment, or add a link to a tracked issue detailing this work item.

## Location

[zebra-state/src/service/check/anchors.rs](#), line 158



---

## Retest Results

2023-05-10 – Fixed

NCC Group reviewed [PR 6450](#) and found that the redundant (and expensive) assertion mentioned by this finding has been removed.

## Client Response

This assertion in the state Sprout anchor checking code and its accompanying comments are outdated. Since it was written, we have modified the anchor creation code to automatically enforce this constraint. The assertion had already been commented out in our code, so we removed it to avoid confusion.



# Fragile Address Limit Implementation

<b>Overall Risk</b>	Informational	<b>Finding ID</b>	NCC-E005955-4FM
<b>Impact</b>	None	<b>Component</b>	zebra-network
<b>Exploitability</b>	None	<b>Category</b>	Session Management
		<b>Status</b>	Fixed

## Impact

Enforcing an underspecified address limit leads to fragile address book update.

## Description

In the following code snippet from zebra-network's AddressBook implementation, the helper function adds a list of peer addresses to the address book. However, it first takes `addr_limit` elements from the list and then deduplicates it, as such if there are repeated elements in the truncated list, the resulting address book will be smaller than the required limit. It is more robust to deduplicate the list and then take `addr_limit` elements from it:

```
143 /// Construct an [AddressBook] with the given local_listener, network,
144 /// addr_limit, [tracing::Span], and addresses.
145 ///
146 /// addr_limit is enforced by this method, and by [AddressBook::update].
147 ///
148 /// If there are multiple [MetaAddr]s with the same address,
149 /// an arbitrary address is inserted into the address book,
150 /// and the rest are dropped.
151 ///
152 /// This constructor can be used to break address book invariants,
153 /// so it should only be used in tests.
154 #[cfg(any(test, feature = "proptest-impl"))]
155 pub fn new_with_addrs(
156     local_listener: SocketAddr,
157     network: Network,
158     addr_limit: usize,
159     span: Span,
160     addrs: impl IntoIterator<Item = MetaAddr>,
161 ) -> AddressBook {
162     let constructor_span = span.clone();
163     let _guard = constructor_span.enter();
164
165     let instant_now = Instant::now();
166     let chrono_now = Utc::now();
167
168     let mut new_book = AddressBook::new(local_listener, network, span);
169     new_book.addr_limit = addr_limit;
170
171     let addrs = addrs
172         .into_iter()
173         .map(|mut meta_addr| {
174             meta_addr.addr = canonical_socket_addr(meta_addr.addr);
175             meta_addr
176         })
177         .filter(|meta_addr| meta_addr.address_is_valid_for_outbound(network))
178         .take(addr_limit)
179         .map(|meta_addr| (meta_addr.addr, meta_addr));
```



```
180
181     for (socket_addr, meta_addr) in addresses {
182         // overwrite any duplicate addresses
183         new_book.by_addr.insert(socket_addr, meta_addr);
184     }
185
186     new_book.update_metrics(instant_now, chrono_now);
187     new_book
188 }
```

Figure 15: [zebra-network/src/address\\_book.rs](#)

As this function is only used in tests, it is marked as an informational finding.

## Recommendation

Clarify in the function's description whether the implementation guarantees to add `addr_limit` (unique) addresses to address book if at least `addr_limit` unique addresses exist in the list, or not.

Consider deduplicating the `addresses` list before taking `addr_limit` elements from it.

## Location

[zebra-network/src/address\\_book.rs](#), line 143

## Retest Results

2023-06-07 – Fixed

NCC Group reviewed [PR 6724](#) and found it adequately addresses this issue.



## 5 Implementation Review Notes

---

This section includes various remarks that are not considered security vulnerabilities, however fixing them will increase code quality.

### Ed25519-zebra

**Potentially incorrect comments:** Some of the comments in `src/batch.rs` seem slightly off:

- On [line 146](#), the documentation of the `verify()` function includes a warning about outputs differing between batched and individual verifications; however, the whole point of ZIP 215 and the `ed25519-zebra` crate is indeed to ensure that batched and non-batched verification always yield identical results on the same signatures.
- On [line 154](#), the described verification equation does not include the multiplication by the cofactor. Such an equation would indeed lead to differences between batched and non-batched outputs. Fortunately, the implementation itself includes the multiplication by the cofactor ([line 214](#)).

**Update:** NCC Group confirmed that these notes have been addressed in <https://github.com/ZcashFoundation/ed25519-zebra/pull/75>.

### Reddsa and redjubjub

The `redjubjub` crate implements the RedDSA signature scheme over the Jubjub curve. The `reddsa` crate mostly subsumes that task and provides RedDSA support for both the Jubjub and Pallas curves; the migration of the code from `redjubjub` to `reddsa`, and its generalization to unify support over distinct elliptic curves, are currently in an incomplete state, resulting in some code duplication. Thus, there are currently (at least) three mostly identical implementations of the multiplication of a curve point by a scalar with the w-NAF method, in `redjubjub/src/scalar_mul.rs`, `reddsa/src/scalar_mul.rs`, and `reddsa/src/orchard.rs`. In all three cases, the input scalar is converted to a w-NAF representation over exactly 256 digits:

```
fn non_adjacent_form(&self, w: usize) -> [i8; 256] {  
    // required by the NAF definition  
    debug_assert!(w >= 2);  
    // required so that the NAF digits fit in i8  
    debug_assert!(w <= 8);
```

The w-NAF representation converts an input value  $x$  into signed digits, using a window width  $w$  (expressed in bits), with the following characteristics:

- Each digit is either zero, or an odd signed integer between  $-2^{w-1}$  and  $+2^{w-1}$ .
- Between any two non-zero digits, there are at least  $w$  digits of value zero.
- $n+1$  output digits are sufficient to represent all possible  $x$  such that  $0 \leq x < 2^n + 2^{n-w}$ .

The `non_adjacent_form()` function requires that the window width  $w$  lies between 2 and 8. The use of 256 digits is then slightly wasteful for the Jubjub and Pallas curves, where scalars are lower than the prime (sub)group order of interest:

- Jubjub order is about  $2^{251.85}$ , and thus needs only 253 digits at most for w-NAF.
- Pallas order is slightly below  $2^{254} + 2^{125.1}$ , leading to a maximum w-NAF scalar size of 255 digits.

Since the number of iterations in the multiplication loop is exactly [equal to the number of digits](#), and each iteration includes at least a curve point doubling, then some of these doublings are wasted (they will repeatedly double the initial value of the accumulator point, i.e. the curve identity point). On the other hand, if the `non_adjacent_form()` and `optional_multiscalar_mul()` functions are turned into generic functions that handle arbitrary curves



---

whose scalars fit on 32 bytes, then 256 digits are not enough: a curve whose order is greater than  $2^{255} + 2^{247}$  (e.g. secp256k1, or NIST's P-256) may need up to 257 digits for its scalars in w-NAF representation.

For the Jubjub and Pallas curves, the performance effect is slight, but the implementation could be made slightly faster, *and* more robust (if generalized to all up-to-256-bits curves), by applying the two following changes:

- The `non_adjacent_form()` function may return 257 digits instead of 256.
- The `non_adjacent_form()` function may also return the actual index of the topmost non-zero digit, allowing `optional_multiscalar_mul()` to start its loop at that index, thus skipping extra digits of value zero.

**Update:** NCC Group confirmed that these notes have been addressed in <https://github.com/ZcashFoundation/reddsa/pull/63>.

### TODOs with Closed Tasks

There are a number of TODOs in the codebase that use task numbers that are already closed. For instance, [issue #862](#) states that the Sync process's `state_contains()` API only checks the best chain for a given block hash (`zebrad/src/components/sync.rs`), rather than checking all the available chains in the mempool. The audited version of the library still queries the best chain in state and the task is closed.

Another example is [issue #2214](#) which suggests that the fanout handling should be done by the PeerSet so that all the fanouts would not use the same peer and block themselves. This seems to be an issue only on testnet where nodes are not as well connected as mainnet and the task is closed. This issue is referenced in a TODO in `zebrad/src/components/sync.rs`.

**Update:** In response to this note, the Zcash team has created a [Tracking: TODOs with closed tasks](#) and is actively addressing them.

### XXX as a TODO Label

It was observed that several annotations in the codebase contain the string "XXX" and appear to represent a TODO or future task. These items may not be identified or documented via code searches for open issues. Examples include:

```
6  /// A very basic retry policy with a limited number of retry attempts.
7  ///
8  /// XXX Remove this when <https://github.com/tower-rs/tower/pull/414> lands.
```



```

9  #[derive(Copy, Clone, Debug, Eq, PartialEq, Hash)]
10 pub struct RetryLimit {
11     remaining_tries: usize,
12 }

```

Figure 16: *zebra-network/src/constants.rs*

```

13 // XXX should these constants be split into protocol also?
14 use crate::protocol::external::types::*;

```

Figure 17: *zebra-network/src/policies.rs*

```

136 fn process_message(&mut self, msg: Message) -> Option<Message> {
137     let mut ignored_msg = None;
138     // XXX can this be avoided?
139     let tmp_state = std::mem::replace(self, Handler::Finished(Ok(Response::Nil)));

```

Figure 18: *zebra-network/src/peer/connection.rs*

The above examples are not exhaustive. It is recommended to review the codebase for the XXX tag (or any other similar tag) and align them with a consistent approach to task tracking.

**Update:** NCC Group confirmed that this note has been addressed in <https://github.com/ZcashFoundation/zebra/pull/6417>.

### Peer Starving

The following open TODO item from *src/peer/connection.rs* is noted:

```

598 // CORRECTNESS
599 //
600 // Currently, select prefers the first future if multiple
601 // futures are ready.
602 //
603 // The peer can starve client requests if it sends an
604 // uninterrupted series of messages. But this is unlikely in
605 // practice, due to network delays.
606 //
607 // If both futures are ready, there's no particular reason
608 // to prefer one over the other.
609 //
610 // TODO: use `futures::select!`, which chooses a ready future
611 //       at random, avoiding starvation
612 //       (To use `select!`, we'll need to map the different
613 //       results to a new enum types.)

```

This would seem to be an worthwhile item to prioritize for resolution. While NCC Group concurs that it is unlikely in practice, it nevertheless could compound with other factors to strengthen network-level attacks.

**Update:** NCC Group confirmed that the documentation has been updated to address this issue in <https://github.com/ZcashFoundation/zebra/pull/6488>.

### Lack of Integration Testing

While investigating a potential mishandled failure scenario, NCC Group noticed a lack of integration testing between the consensus block verifier service and read state service. It is highly recommended that in addition to testing individual services, the interaction between them be tested for potential request/response mishandling. Consider the



following example: `zebra-consensus` prevents double spending by detecting duplicated nullifiers and anchors in a transaction, a block, and main chain's history. It uses the read state service from the `zebra-state` to ensure a nullifier or anchor has not been used to spend the input note in the past.

```
1555 ReadRequest::CheckBestChainTipNullifiersAndAnchors(unmined_tx) => {
1556     let timer = CodeTimer::start();
1557
1558     let state = self.clone();
1559
1560     let span = Span::current();
1561     tokio::task::spawn_blocking(move || {
1562         span.in_scope(move || {
1563             let latest_non_finalized_best_chain =
1564                 state.latest_non_finalized_state().best_chain().cloned();
1565
1566             check::nullifier::tx_no_duplicates_in_chain(
1567                 &state.db,
1568                 latest_non_finalized_best_chain.as_ref(),
1569                 &unmined_tx.transaction,
1570             )?;
1571
1572             check::anchors::tx_anchors_refer_to_final_treestates(
1573                 &state.db,
1574                 latest_non_finalized_best_chain.as_ref(),
1575                 &unmined_tx,
1576             )?;
1577
1578             // The work is done in the future.
1579             timer.finish(module_path!(), line!(), "ReadRequest::UnspentBestChainUtxo");
1580
1581             Ok(ReadResponse::ValidBestChainTipNullifiersAndAnchors)
1582         })
1583     })
1584     .map(|join_result| join_result.expect("panic in ReadRequest::UnspentBestChainUtxo"))
1585     .boxed()
1586 }
```

Figure 19: `zebra-state/src/service.rs`

The highlighted lines in the snippet above returns a `ValidateContextError` error if a duplicate is found. On the `zebra-consensus` side, this response is handled in the following snippet:

```
417 if let Some(unmined_tx) = req.mempool_transaction() {
418     let check_anchors_and_revealed_nullifiers_query = state
419         .clone()
420         .oneshot(zs::Request::CheckBestChainTipNullifiersAndAnchors(
421             unmined_tx,
422         ))
423         .map(|res| {
424             assert!(res? == zs::Response::ValidBestChainTipNullifiersAndAnchors,
425                 ↳ "unexpected response to CheckBestChainTipNullifiersAndAnchors request");
426             Ok(())
427         })
428 }
```





```

427     );
428
429     async_checks.push(check_anchors_and_revealed_nullifiers_query);
430 }

```

Figure 20: [zebra-consensus/src/transaction.rs](#)

The `res?` in the `assert!()` on line 424, will propagate this error. NCC Group browsed the repository for unit tests that would cover this interaction. It seems that the unit tests in the `zebra-consensus`, either test for a duplicate inside a single transaction or duplicates inside a block (see [v4\\_transaction\\_with\\_conflicting\\_sprout\\_nullifier\\_inside\\_joinsplit\\_is\\_ejected\(\)](#) as an example), and there are no tests to validate the above code-path.

**Update:** NCC Group confirmed that this note has been addressed in <https://github.com/ZcashFoundation/zebra/pull/6665>.

### Outdated Documentation

The `Chain::update_chain_state_with` that is mentioned in the documentation for the `with_block_and_spent_utxos()` function does not exist (see snippet below). It should probably be replaced with `update_chain_tip_with()`. The same error exists in the [book/src/dev/rfcs/0012-value-pools.md](#) document.

```

264 impl ContextuallyValidBlock {
265     /// Create a block that's ready for non-finalized `Chain` contextual validation,
266     /// using a [`PreparedBlock`] and the UTXOs it spends.
267     ///
268     /// When combined, `prepared.new_outputs` and `spent_utxos` must contain
269     /// the [`Utxo`](transparent::Utxo)s spent by every transparent input in this block,
270     /// including UTXOs created by earlier transactions in this block.
271     ///
272     /// Note: a [`ContextuallyValidBlock`] isn't actually contextually valid until
273     /// `Chain::update_chain_state_with` returns success.
274     pub fn with_block_and_spent_utxos(

```

Figure 21: Snippet from [zebra-state/src/request.rs](#).

**Client Comment:** “We don’t usually update RFCs, they are a point-in-time “request for comments” from developers, but we have updated the appropriate rustdoc.”

**Update:** The Zcash Foundation team has added more details regarding this note in <https://github.com/ZcashFoundation/zebra/issues/6673>. They also indicated that they do not usually update RFCs, as they are a point-in-time “request for comments” from developers. The documentation issue in the code snippet above is fixed in [PR 6781](#).

### A Closer Look at Zebra’s Finalized State

This note is mainly a discussion around `zebra-state`’s handling of finalized block state. The Zebra application’s `start` command spawns the `Syncer` service which will continuously request chain tips and blocks from the peers until it downloads and verifies enough history of the chain to be able to validate blocks. It obtains some prospective tips and iteratively tries to extend them and download the missing blocks. In the snippet below, the `Syncer`’s `download_and_verify()` function rejects blocks that are more than `MAX_BLOCK_REORG_HEIGHT` (99) blocks from the tip height:

```

364 // Get the finalized tip height, assuming we're using the non-finalized state.
365 //
366 // It doesn't matter if we're a few blocks off here, because blocks this low
367 // are part of a fork with much less work. So they would be rejected anyway.

```



```

368 //
369 // And if we're still checkpointing, the checkpointer will reject blocks behind
370 // the finalized tip anyway.
371 //
372 // TODO: get the actual finalized tip height
373 let min_accepted_height = tip_height
374     .map(|tip_height| {
375         block::(tip_height.0.saturating_sub(zs::MAX_BLOCK_REORG_HEIGHT))
376     })
377     .unwrap_or(block::(0));

```

Figure 22: Snippet from `zebrad/src/components/sync/downloads.rs`.

Then the Syncer uses a clone of the `ChainVerifier` to verify the newly downloaded block, which for blocks below the latest configured checkpoint height will use the `CheckpointVerifier` instead of full block verification. The `CheckpointVerifier` will not process blocks beyond the target checkpoint until it receives all the required blocks that it needs in order to be able to verify the blocks that chain back to the previous checkpoint. In the snippet below, the `CheckpointVerifier` uses the state service to contextually verify the block and commit it to the finalized state:

```

980 let state_service = self.state_service.clone();
981 let commit_finalized_block = tokio::spawn(async move {
982     let hash = req_block
983         .rx
984         .await
985         .map_err(Into::into)
986         .map_err(VerifyCheckpointError::CommitFinalized)
987         .expect("CheckpointVerifier does not leave dangling receivers");
988
989     // We use a `ServiceExt::oneshot`, so that every state service
990     // `poll_ready` has a corresponding `call`. See #1593.
991     match state_service
992         .oneshot(zs::Request::CommitFinalizedBlock(req_block.block))
993         .map_err(VerifyCheckpointError::CommitFinalized)
994         .await?
995     {
996         zs::Response::Committed(committed_hash) => {
997             assert_eq!(committed_hash, hash, "state must commit correct hash");
998             Ok(hash)
999         }
1000         _ => unreachable!("wrong response for CommitFinalizedBlock"),
1001     }
1002 });

```

Figure 23: Snippet from `zebra-consensus/src/checkpoint.rs`.

Note that a checkpointed/finalized block can be a settled network upgrade or a block beyond the rollback/reorg limit. The state service processes the finalized block and queues it to be committed to state (finalized blocks can be persisted on disk<sup>8</sup>). The state service

8. Zebrad can be run in a mode in which the application deletes its cache on shutdown. See the documentation for the configurable `ephemeral flag` for more details.



will not commit any non-finalized blocks until the finalized blocks' queue is fully drained and committed:

```
639 // We've finished sending finalized blocks when:
640 // - we've sent the finalized block for the last checkpoint, and
641 // - it has been successfully written to disk.
642 //
643 // We detect the last checkpoint by looking for non-finalized blocks
644 // that are a child of the last block we sent.
645 //
646 // TODO: configure the state with the last checkpoint hash instead?
647 if self.finalized_block_write_sender.is_some()
648     && self
649         .queued_non_finalized_blocks
650         .has_queued_children(self.last_sent_finalized_block_hash)
651     && self.read_service.db.finalized_tip_hash() == self.last_sent_finalized_block_hash
652 {
653     // Tell the block write task to stop committing finalized blocks,
654     // and move on to committing non-finalized blocks.
655     std::mem::drop(self.finalized_block_write_sender.take());
656
657     // We've finished committing finalized blocks, so drop any repeated queued blocks.
658     self.clear_finalized_block_queue(
659         "already finished committing finalized blocks: dropped duplicate block, \
660         block is already committed to the state",
661     );
662 }
```

Figure 24: Snippet from `zebra-state/src/service.rs`.

Once the finalized block writer is finished, the state service will validate and commit the non-finalized queued blocks whose parents have recently arrived in breadth-first ordering (lowest to highest heights). Thus, when `validate_and_commit_non_finalized()` calls the `check::initial_contextual_validity()` for the first time there are some finalized blocks in the state to be used to validate the block's difficulty threshold and timestamp. The `assert_eq!()` check below expects at least 28 blocks in the relevant chain:

```
85 // skip this check during tests if we don't have enough blocks in the chain
86 #[cfg(test)]
87 if relevant_chain.len() < POW_AVERAGING_WINDOW + POW_MEDIAN_BLOCK_SPAN {
88     return Ok(());
89 }
90 // process_queued also checks the chain length, so we can skip this assertion during testing
91 // (tests that want to check this code should use the correct number of blocks)
92 assert_eq!(
93     relevant_chain.len(),
94     POW_AVERAGING_WINDOW + POW_MEDIAN_BLOCK_SPAN,
95     "state must contain enough blocks to do proof of work contextual validation, \
96     and validation must receive the exact number of required blocks"
97 );
```

Figure 25: Snippet from `zebra-state/src/service/check.rs`.

For the first non-finalized block to be validated there must be at least 28 blocks in the finalized state. If an honest node fails to obtain enough finalized blocks from its peers during syncing, when it starts to contextually validate the pending non-finalized blocks, it will fail this assertion and crash the Zebra node.



---

After further discussions with the Zcash Foundation team, it was determined that this assertion will not be triggered in cases where the parent block or the UTXO transactions that are required to contextually validate the block are missing, as in these cases, the block validation will not proceed until sufficient history is acquired. Furthermore, blocks are validated in strict height order, and non-finalized blocks will not be processed until after all the checkpointed blocks are committed to the finalized state. The Zebra application is configured such that, by default, it pulls at least 2 million checkpointed blocks<sup>9</sup>, as such the assertion will always be satisfied.

A relevant issue (“[On startup, check that the finalized state blocks match the checkpoint hashes](#)”) was fixed by the maintainers after the audit branch was cut. It ensures that, on startup, the Zebra node would not indefinitely follow a forked chain that does not include the socially accepted checkpoints.

**Update:** The Zcash Foundation team has added more details regarding this note in <https://github.com/ZcashFoundation/zebra/discussions/6620>. In addition, this note’s concerns have been addressed by replacing the mentioned assertion in `block_is_valid_for_recent_chain()` with a validation error (`ValidateContextError::NotReadyToBeCommitted`). See <https://github.com/ZcashFoundation/zebra/pull/7072> for details.

### Notes on Peer Sourcing

When a peer initially connects to the network, they start by sourcing peers from a list of known-good mainnet peers. The default list is specified in code in `zebra/zebra-network/src/config.rs`, although users can override it by generating and editing a `zebrad.toml` config file.

```
277 let mainnet_peers = [  
278     "dnsseed.z.cash:8233",  
279     "dnsseed.str4d.xyz:8233",  
280     "mainnet.seeder.zfnd.org:8233",  
281     "mainnet.is.yolo.money:8233",  
282 ]
```

The use of a variety of trusted introduction points with distinctive, easily recognized names is considered a best practice. However, taken together, these endpoints do represent a degree of centralization in the network infrastructure. While users technically *can* change or extend this list, it is doubtful that many of them *will*.

Additionally, changes to the default list would not be propagated to any user who has made local modifications. One could imagine a case where one of the default mainnet peers is compromised, loses trust, and is removed from the list, but peers who edited their configuration to extend this list fail to receive this change and continue connecting to a malicious peer.

Scenarios such as this one could be mitigated by decomposing the default mainnet peer list into two lists: one, the “officially sanctioned” default list, which users technically *could* edit but are not expected to, and another, which may default to an empty list, which is intended to contain any additional peers a user may wish to use by default. This would simplify peer list maintenance, leading to better user experience and maintainability. It could also act as a subtle encouragement for peers to diversify their set of introduction points, which would be good for network health and resilience (as long as users vet any new introduction points to their satisfaction out-of-band).

---

9. See `init_checkpoint_list()` from the `zebra-consensus` router for details. Note that this reference is ahead of the audited commit.



---

As a further improvement, the degree of centralization could be reduced by following a practice commonly seen in other peer-to-peer systems, e.g., torrent clients, where the client will attempt, on startup, to reconnect to whoever it was connected to before it last exited the network. The initial mainnet peers can be used as well, and should probably be preferred<sup>10</sup>, but they are no longer treated as an exclusive source of truth<sup>11</sup> regarding the peer-to-peer network.

**Update:** The Zcash Foundation team has added more details regarding this note in <https://github.com/ZcashFoundation/zebra/issues/6675>. In addition, this note is partially addressed by periodically storing the latest peers from the address book to disk, and adding them to the configured DNS seed peers on Zebra startup. See <https://github.com/ZcashFoundation/zebra/pull/6739> for details.

---

10. When considering attacks at the level of the routing overlay, we can assume that malicious peers will only refer us to other malicious peers, whereas honest peers may refer us to either a malicious or an honest peer. Thus, even if we trust our introduction point, this trust is not fully transitive, and our trust in other peers should be inversely correlated to the number of routing hops required to reach them - although of course this level of trust cannot be precisely quantified in general. What we *can* say in general is that even if our introduction points are not exclusively trusted, they are likely the *most* trusted peers we have.

11. Slight caveat: the default peers likely - though not necessarily, if we've modified the peer list - gave us our original introduction to these cached peers (either directly or by introducing their introducers, etc). Therefore there is still a great deal of trust placed in these introduction points; the improvement here is that we have to extend this trust for a shorter time window.



## 6 Additional Code Changes

---

The following code changes were added to the Zebra codebase after the [audit tag](#) was created. All the changes that resulted from this report's findings and notes in the [Implementation Review Notes](#) have been examined and documented in their corresponding sections<sup>12</sup>. The following notes are added for completeness:

**fix(state): Fix minute-long delays in block verification after a chain fork:** refactors the `zebra-state`'s non-finalized chains' implementation. It primarily simplifies the note commitment trees' maintenance logic and improves chain forking's performance. Some unit tests were broken by this change and fixing them is left as `TODO`s.

**cleanup(consensus): Remove unused impl ZcashDeserialize for Height:** removes the unused `ZcashSerialize` implementation for the `Height` type to prevent misuse.

**fix(mempool): Re-verify transactions that were verified at a different tip height:** ensures that a newly downloaded and verified transaction is added to the mempool, only-if the current best tip is at the same height as the tip height that was used for transaction validation.

**fix(consensus): Check that Zebra's state contains the social consensus chain on startup:** introduces a checkpoint verification task to be run at `Zcashd` start-up. This task ensures that the finalized chain which is fetched from peers includes the socially accepted checkpoints and is not an invalid fork of the chain. This change improves a new node's start-up performance by pruning invalid chains earlier.

**fix(sync): Pause new downloads when Zebra reaches the lookahead limit:** ensures that the block syncer (downloader) pauses new downloads after the lookahead limit is reached. With this measure, the downloader allows the verifier and state services to catch up, and prevents downloading too many far ahead and possibly invalid blocks.

**fix(rpc): Shut down the RPC server properly when Zebra shuts down:** makes a new thread that handles the RPC server's shutdown. This measure prevents the encompassing tokio task from panicking.

**refactor(state): Make implementation of block consensus rules clearer:** renames some internal functions to make their functionality more clear.

**Fix Halborn network security disclosures:** limits memory usage that can lead to denial-of-service attacks, e.g., limits the number of inventory hashes that are sent to the peer, limits the number of inventory hashes that are stored, limits the number of peers that are tracked for an inventory hash, limits the size of the `VersionMessage` that is stored for a peer, and limits the log size on disk.

**fix(net): Limit the number of leftover nonces in the self-connection nonce set:** 1. limits the time interval between inbound connections as well as outbound connections. 2. removes leftover nonces that were not cleaned-up from the state due to their corresponding handshakes' failure.

**fix(net): Allow each initial peer to send one inbound request before disconnecting any peers:** increases the maximum number of concurrent inbound connections to match the target peer set. A lower inbound connections limit lead to connections being dropped at startup.

---

12. These issues have been tracked by the project's issue tracker in [Epic: Improvements from Zebra Audit](#).



---

**fix(ci): Avoid inbound service overloads and fix failing tests:** fixes some frequently failing tests and tweaks some constants. Moves the Inbound service's startup higher in the sequence of tasks that are created at *Zebra* startup.

**security(state): Limit the number of non-finalized chains tracked by Zebra:** prunes the non-finalized chains that are tracked in `zebra-state` by removing chains with the smallest amount of work.

**fix(rpc): Omit transactions with transparent coinbase spends that are immature at the next block height from block templates:** ensures that the `getblocktemplate` RPC method does not construct blocks with spent transparent coinbase transactions that are immature.

**fix(net): Avoid a rare panic when a connection is dropped:** prevents a panic that could be caused by a one-shot sender not receiving its response before it was dropped.



# 7 Finding Field Definitions

---

The following sections describe the risk rating and category assigned to issues NCC Group identified.

## Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

## Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

## Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

## Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	





Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
<b>Low</b>	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
<b>Access Controls</b>	Related to authorization of users, and assessment of rights.
<b>Auditing and Logging</b>	Related to auditing of actions, or logging of problems.
<b>Authentication</b>	Related to the identification of users.
<b>Configuration</b>	Related to security configurations of servers, devices, or software.
<b>Cryptography</b>	Related to mathematical protections for data.
<b>Data Exposure</b>	Related to unintended exposure of sensitive information.
<b>Data Validation</b>	Related to improper reliance on the structure or values of data.
<b>Denial of Service</b>	Related to causing system failure.
<b>Error Reporting</b>	Related to the reporting of error conditions in a secure fashion.
<b>Patching</b>	Related to keeping software up to date.
<b>Session Management</b>	Related to the identification of authenticated users.
<b>Timing</b>	Related to race conditions, locking, or order of operations.

