



# Aleo snarkOS Implementation and Consensus Mechanism Review

Aleo Systems

Version 1.0 – February 5, 2024

©2024 – NCC Group

Prepared by NCC Group Security Services, Inc. for Aleo Systems Inc. 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**  
Elena Bakos Lang  
Paul Bottinelli  
Kevin Henry  
Eric Schorn

**Prepared For**  
Collin Chin  
Raymond Chu  
Howard Wu

# 1 Executive Summary

---

## Synopsis

In November 2023, Aleo engaged NCC Group's Cryptography Services team to perform a review of the consensus mechanism implemented by snarkOS: "a decentralized operating system for zero-knowledge applications [that] forms the backbone of Aleo network, which verifies transactions and stores the encrypted state applications in a publicly verifiable manner." The consensus mechanism is based on a partially synchronous version of the Bullshark Byzantine Fault Tolerance (BFT) protocol, which uses a directed acyclic graph (DAG) to order updates. The review was performed remotely by four consultants, over a total of 25 person-days of effort. A retest was performed in January 2024.

## Scope

The review targeted the *snarkOS* repository (tagged branch [testnet3-audit-ncc / commit 1ab8c4c](#)). The main focus of the review was on the subdirectories *snarkOS/node/bft* and *snarkOS/node/consensus/src*, with the rest of the *snarkOS/node* being of secondary focus. The review is supplemented by the following resources relating the underlying BFT DAG and Aleo's approach:

- [Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus](#)
- [Bullshark: The Partially Synchronous Version](#)
- The Aleo Whitepaper (draft)

A prior review by NCC Group examined portions of *snarkVM*, which provides data structures and utilities that are leveraged by *snarkOS*. Some findings in this report reference *snarkVM* code as a result.

Retesting was performed via individual pull requests targeting each individual finding as applicable.

## Limitations

Several findings reference data structures or functions in *snarkVM*, however, a complete formal review of *snarkVM* is not included in this report.

## Key Findings

Notable findings from the assessment include:

- [Finding "Validators May Update Round and Trigger Garbage Collection Upon Receipt of Maliciously Crafted Certificates"](#), which details a situation in which a node can be induced to advance their current round number based on unvalidated data, thereby triggering garbage collection on data required to advance the consensus state.
- [Finding "Timestamp Calculation Does Not Provide Byzantine Fault Tolerance"](#), which demonstrates how a set of Byzantine validators can maliciously skew the consensus timestamp without exceeding the allowed Byzantine threshold.
- Several findings relating to brittle or late error handling across the codebase.

Additionally, several comments and observations that did not warrant formal findings are documented in the appendix [Engagement Notes](#).

After retesting, NCC Group found that eight (8) of eleven (11) reported findings had been addressed by the team at Aleo, with two (2) remaining findings acknowledged as "Risk Accepted", and one (1) finding marked as "Not Fixed". These unaddressed findings represent security improvement opportunities and not vulnerabilities within the consensus mechanism and may be addressed in the future.



---

## Strategic Recommendations

- Most literature relating to BFT systems models individual participants, a threshold of which may be Byzantine. In adapting BFT algorithms to a stake-based model, careful consideration is required to ensure that the underlying assumptions remain valid. Decisions made with respect to a number of participants (e.g., a number of active network connections) must be considered in a manner that acknowledges the stake held by these participants.
- Ensure that detectable errors are handled as early as possible within functions in a robust manner.
- The *snarkOS* repository includes a fairly comprehensive set of tests. However, a significant proportion of these tests exercise more than one functionality; only few tests are true *unit tests*. While this is somewhat expected in complex systems such as the consensus protocol implemented in *snarkOS* (since it requires non-trivial amount of set-up code before performing any operation), writing unit tests exercising single functions would be beneficial, for example to facilitate reviewers' efforts at dynamically testing individual portions of the code.



## 2 Dashboard

### Target Data

Name	snarkOS
Type	Blockchain Library
Platforms	Rust
Environment	Local

### Engagement Data





Type	Cryptography / Security Assessment
Method	Code-assisted
Dates	2023-11-09 to 2023-12-01
Consultants	4
Level of Effort	25

### Targets








snarkOS <https://github.com/AleoHQ/snarkOS/tree/testnet3-audit-ncc>

“a decentralized operating system for zero-knowledge applications [that] forms the backbone of Aleo network, which verifies transactions and stores the encrypted state applications in a publicly-verifiable manner.”

### Finding Breakdown

Critical issues	0	
High issues	1	
Medium issues	1	
Low issues	8	
Informational issues	1	
<b>Total issues</b>	<b>11</b>	

### Category Breakdown






Access Controls	1	
Cryptography	1	
Data Exposure	1	
Data Validation	5	
Denial of Service	1	
Other	1	
Session Management	1	








---

## Component Breakdown

---

snarkOS cli	2	
snarkOS node	6	
snarkOS node, snarkVM batch-certificate	1	
snarkVM ledger	1	
snarkVM subdag	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
Validators May Update Round and Trigger Garbage Collection Upon Receipt of Maliciously Crafted Certificates	Fixed	FA9	High
Timestamp Calculation Does Not Provide Byzantine Fault Tolerance	Fixed	EVJ	Medium
Late/Distant Validation of Block Request Range	Risk Accepted	JR7	Low
Brittle Error Handling of <code>NoiseState::Handshake</code>	Risk Accepted	Y4N	Low
Secret Key Stored in Plaintext File, Missing Permission Checks	Fixed	K4R	Low
Event <code>FromBytes</code> Tolerates Extraneous Input	Fixed	UJF	Low
CLI Input Private Keys and Sensitive Values Not Zeroized	Fixed	7VR	Low
Missing Validation of Transmission Response	Fixed	UPV	Low
Potentially Dangerous Handling of Duplicate Signatures on Certificates	Fixed	76W	Low
Weak Malicious Peer Handling	Not Fixed	LA4	Low
Leader Election Process Does Not Match Whitepaper	Fixed	VFD	Info



## 4 Finding Details

High

# Validators May Update Round and Trigger Garbage Collection Upon Receipt of Maliciously Crafted Certificates

Overall Risk High

Impact High

Exploitability High

Finding ID NCC-E009544-FA9

Component snarkOS node

Category Data Validation

Status Fixed

### Impact

Invalid certificates may cause a node to update its local round number, potentially triggering garbage collection and preventing the node from accepting other valid batch proposals or certificates. This effectively constitutes a denial-of-service attack and prevents the validator (and their stake) from contributing to the consensus computation.

### Description

The function `process_batch_certificate_from_peer()` takes as input a `BatchCertificate` received from a peer, stores a valid certificate locally, and updates to the next round if all received certificates so far reach quorum. As part of this process, the function ensures that the local state is consistent with the state described in the `BatchCertificate` and updates it as needed, which is done in the `sync_with_certificate_from_peer()` function. This function first syncs the local state based on the provided `BatchHeader` by calling `sync_with_batch_header()`, before calling `insert_certificate()` to validate the certificate and add it to local storage:

```
1266 // If the peer is ahead, use the batch header to sync up to the peer.
1267 let missing_transmissions = self.sync_with_batch_header_from_peer(peer_ip, batch_h
↳ eader).await?;
1268
1269 // Check if the certificate needs to be stored.
1270 if !self.storage.contains_certificate(certificate.id()) {
1271 // Store the batch certificate.
1272 self.storage.insert_certificate(certificate.clone(), missing_transmissions)?;
1273 debug!("Stored a batch certificate for round {batch_round} from '{peer_ip}'");
1274 // If a BFT sender was provided, send the round and certificate to the BFT.
1275 if let Some(bft_sender) = self.bft_sender.get() {
1276 // Send the certificate to the BFT.
1277 if let Err(e) =
↳ bft_sender.send_primary_certificate_to_bft(certificate).await {
1278 warn!("Failed to update the BFT DAG from sync: {e}");
1279 return Err(e);
1280 };
1281 }
1282 }
```

Figure 1: `node/bft/src/primary.rs`



In particular, the function `sync_with_batch_header_from_peer()` updates the local round if the node has fallen behind:

```
1300 // Check if our primary should move to the next round.
1301 // TODO (howardwu): Re-evaluate whether we need to guard this to increment after
↳ quorum threshold is reached.
1302 let is_behind_schedule = batch_round > self.current_round();
1303 // Check if our primary is far behind the peer.
1304 let is_peer_far_in_future = batch_round > self.current_round() +
↳ self.storage.max_gc_rounds();
1305 // If our primary is far behind the peer, update our committee to the batch round.
1306 if is_behind_schedule || is_peer_far_in_future {
1307 // If the batch round is greater than the current committee round, update the
↳ committee.
1308 self.try_increment_to_the_next_round(batch_round).await?;
1309 }
```

Figure 2: `node/bft/src/primary.rs`

However, note that the bulk of the certificate validation does not occur until the call to `insert_certificate()` that occurs *after* syncing with the header, via a call to `check_certificate()`:

```
495 pub fn insert_certificate(
496     &self,
497     certificate: BatchCertificate<N>,
498     transmissions: HashMap<TransmissionID<N>, Transmission<N>>,
499 ) -> Result<()> {
500     // Ensure the certificate round is above the GC round.
501     ensure!(certificate.round() > self.gc_round(), "Certificate round is at or below
↳ the GC round");
502     // Ensure the certificate and its transmissions are valid.
503     let missing_transmissions = self.check_certificate(&certificate, transmissions)?;
504     // Insert the certificate into storage.
505     self.insert_certificate_atomic(certificate, missing_transmissions);
506     Ok(())
507 }
```

Figure 3: `node/bft/src/helpers/storage.rs`

As a consequence, the current local round number (and local garbage collected round) may update based on the information contained within an invalid certificate, which may cause a node to reject otherwise valid proposals or certificates in the future. In particular, the current round for the node may be fast-forwarded up to `max_gc_rounds` in the future, thereby preventing it from reaching the availability threshold or quorum threshold to advance its state based on the current consensus information.

Note that the first action taken after advancing the local round number is to fetch the missing certificates from the peer, which may enable this attack to be executed recursively using a chain of maliciously crafted certificates. It should be emphasized that a single malicious peer can craft a certificate that will pass initial validation, although it will not pass the availability or quorum thresholds.

As a separate comment, the `sync_with_batch_header()` function distinguishes between peers that have fallen behind, and peers that have fallen behind by more than





`max_gc_rounds`, suggesting that the committee will be updated to the provided round in either case:

```
1300 // Check if our primary should move to the next round.
1301 // TODO (howardwu): Re-evaluate whether we need to guard this to increment after
    ↳ quorum threshold is reached.
1302 let is_behind_schedule = batch_round > self.current_round();
1303 // Check if our primary is far behind the peer.
1304 let is_peer_far_in_future = batch_round > self.current_round() +
    ↳ self.storage.max_gc_rounds();
1305 // If our primary is far behind the peer, update our committee to the batch round.
1306 if is_behind_schedule || is_peer_far_in_future {
1307     // If the batch round is greater than the current committee round, update the
    ↳ committee.
1308     self.try_increment_to_the_next_round(batch_round).await?;
1309 }
```

Figure 4: `node/bft/src/primary.rs`

However, the `try_increment_to_the_next_round()` only updates to the current round if the node is less than `max_gc_rounds()` behind, incrementing by a single round otherwise:

```
1112 // If the next round is within GC range, then iterate to the penultimate round.
1113 if self.current_round() + self.storage.max_gc_rounds() >= next_round {
1114     let mut fast_forward_round = self.current_round();
1115     // Iterate until the penultimate round is reached.
1116     while fast_forward_round < next_round.saturating_sub(1) {
1117         // Update to the next round in storage.
1118         fast_forward_round =
1119             ↳ self.storage.increment_to_next_round(fast_forward_round)?;
1120         // Clear the proposed batch.
1121         *self.proposed_batch.write() = None;
1122     }
1123 }
1124 // Retrieve the current round.
1125 let current_round = self.current_round();
1126 // Attempt to advance to the next round.
1127 if current_round < next_round {
1128     // If a BFT sender was provided, send the current round to the BFT.
1129     let is_ready = if let Some(bft_sender) = self.bft_sender.get() {
1130         match bft_sender.send_primary_round_to_bft(current_round).await {
1131             Ok(is_ready) => is_ready,
1132             Err(e) => {
1133                 warn!("Failed to update the BFT to the next round - {e}");
1134                 return Err(e);
1135             }
1136         }
1137     }
1138     // Otherwise, handle the Narwhal case.
1139     else {
1140         // Update to the next round in storage.
1141         self.storage.increment_to_next_round(current_round)?;
1142         // Set 'is_ready' to 'true'.
1143         true
1144     };
};
```

Figure 5: `node/bft/src/primary.rs`



It is unclear whether this behavior is intentional. In particular, a case where an honest peer is more than `max_gc_rounds` rounds in the future implies that a quorum of honest peers has surpassed the garbage collection threshold relative to the node's local round and will no longer be storing the necessary information for the node to catch up. This suggests that the check `if is_behind_schedule || is_peer_far_in_future` may be incorrect and the behavior should differ based on the value of `is_peer_far_in_future`.

## Recommendation

Revisit the TODO item suggesting it is not safe to advance the round until a quorum is reached and consider what validation can be performed prior to this step. Advancing rounds one-by-one based on the availability threshold or quorum threshold may be safer but would require a large number of certificates to be fetched and processed in sequence. If a node is out of sync by more than `max_gc_rounds` rounds, then an alternative approach may be necessary, but only when a quorum of validators agree on the round number.

Additionally, determine the correct behavior for a node that has fallen far behind, and update the `sync_with_batch_header()` and `try_increment_to_the_next_round()` functions accordingly.

## Location

- [snarkOS/blob/node/bft/src/primary.rs](#)
- [snarkOS/blob/node/bft/src/helpers/storage.rs](#)

## Retest Results

### 2024-01-15 – Fixed

NCC Group reviewed [pull request 2897](#) (commit [3f9083f](#) at the time of retest; not yet merged) which updated the behavior of `sync_with_batch_header_from_peer()` to separate the cases of being behind schedule (within garbage collection range) and being far behind schedule (larger than the garbage collection range). In the latter case, a newly added cache is used to ensure that recovery from Byzantine-triggered garbage collection is possible.

```
1327 // If our primary is behind shedule, update our committee to the batch round.
1328 if is_behind_schedule {
1329     // If the batch round is greater than the current committee round, update the
1330     // ↳ committee.
1331     self.try_increment_to_the_next_round(batch_round).await?;
1332 // If our peer is far ahead, check if a quorum of peers is ahead and consider
1333 // ↳ updating our committee.
1334 } else if is_peer_far_in_future {
1335     // Get the highest round seen from a quorum of the current committee
1336     let committee = self.ledger.get_committee_for_round(self.current_round())?;
1337     let round_with_quorum =
1338     (*self.batch_round_cache.write()).update(batch_round, batch_header.author()
1339     // ↳ , &committee?);
1340     let is_quorum_far_in_future = round_with_quorum > self.current_round() +
1341     // ↳ self.storage.max_gc_rounds();
1342 // If our primary is far behind a quorum of peers, update our committee to the
1343 // ↳ round_with_quorum.
1344 if is_quorum_far_in_future {
1345     self.try_increment_to_the_next_round(round_with_quorum).await?;
1346 }
1347 }
```

Figure 6: Revised approach in `node/bft/src/primary.rs`



---

The highlighted `update()` function has been added to perform the following:

```
99     /// Update based on a new (round, address) pair seen in the wild. This does two things:
100    /// - If the round is higher than a previous one from this address, set it in
      ↳ highest_rounds
101    /// - Keep incrementing `last_highest_round_with_quorum` as long as it passes a stake-
      ↳ weighted quorum
102    /// We ignore the case where tomorrow's stake-weighted quorum round is *lower* than
      ↳ the current one
103    pub fn update(&mut self, round: u64, validator_address: Address<N>, committee:
      ↳ &Committee<N>) -> Result<u64> {
```

*Figure 7: New functionality in [node/bft/src/helpers/cache\\_round.rs](#)*

This function implements the recommendation that the round should only be advanced one-by-one as long as consensus is met at each step. As this approach is substantially slower than the previous approach, it is only applied in situations where a peer claims to be far in the future.

The implemented changes are aligned with the recommendations made above, and this finding has been marked “Fixed”.

# Timestamp Calculation Does Not Provide Byzantine Fault Tolerance

Overall Risk Medium

Impact Medium

Exploitability Medium

Finding ID NCC-E009544-EVJ

Component snarkVM subdag

Category Data Validation

Status Fixed

## Impact

The BFT mechanism assumes that a two-thirds majority of the stake is held by honest validators, but parts of the implementation require that the *number* of honest validators be a majority. As a result, a Byzantine validator may influence the consensus timestamp in a manner not reflected by their stake.

## Description

The Byzantine Fault Tolerance (BFT) mechanism ensures that the system can progress and achieve consensus provided that fewer than one third of the validator committee behave in a Byzantine manner (e.g., arbitrarily deviate from the protocol). In general settings, each validator is weighted equally, and the Byzantine threshold refers to *the number of Byzantine validators* in the committee. Because the Aleo blockchain uses a proof-of-stake mechanism, the Byzantine threshold is viewed in terms of the *amount staked to Byzantine validators*, where consensus should be achieved if the stake held by Byzantine validators is less than one third of the total stake. Reframing Bullshark, the underlying consensus mechanism, in terms of stake rather than validators, requires careful consideration to ensure that Byzantine validators cannot exert undue influence on the resulting block chain.

The consensus mechanism ensures that a given anchor block satisfies an availability threshold or quorum threshold, which ensures that a sufficient number of validators and a sufficient amount of stake is backing the update. However, part of the computation of this anchor block involves consensus on the timestamp to be included in the block. The expected process is described in a draft of *The Aleo Whitepaper*:

The timestamp<sub>i</sub> in block<sub>i</sub> is computed as the median timestamp of the batch certificates. Validators ensure the timestamp<sub>i</sub> is greater than timestamp<sub>{i-1}</sub> from block<sub>{i-1}</sub>.

It follows from the honest majority assumption that if  $n = 3f + 1$  then the median timestamp is safe from corruption as it is derived from at least  $(n - f)$  batch certificates. Thus, in the worst case scenario, at least 1/2 of the reported timestamps are honest and the median cannot be corrupted.

---

This description is written in a language where the parameter  $f$  denotes the *number of Byzantine validators* and not the amount staked by these validators. The implementation aligns with this description, where the timestamp of an anchor block is computed as:

```
123     /// Returns the timestamp of the anchor round, defined as the median timestamp of the
124     ↪ leader certificate.
125     pub fn timestamp(&self) -> i64 {
126         // Retrieve the median timestamp from the leader certificate.
127         self.leader_certificate().median_timestamp()
128     }
```

Figure 8: [snarkVM/ledger/narwhal/subdag/src/lib.rs](#)

where `median_timestamp()` returns the median, *without consideration of individual stake*:

```
129     /// Returns the median timestamp of the batch ID from the committee.
130     pub fn median_timestamp(&self) -> i64 {
131         let mut timestamps = self.timestamps().chain(
132         ↪ [self.batch_header.timestamp()].into_iter()).collect::<Vec<_>>();
133         timestamps.sort_unstable();
134         timestamps[timestamps.len() / 2]
135     }
```

Figure 9: [snarkVM/ledger/narwhal/batch-certificate/src/lib.rs](#)

While it follows that a majority of the stake is held by honest validators, it does not follow that a majority of the timestamps are honest, *which directly contradicts the conclusion above*:

Thus, in the worst case scenario, at least 1/2 of the reported timestamps are honest and the median cannot be corrupted.

As an alternative, the timestamp calculation could be updated to leverage the [weighted median](#), where the weight of each timestamp is proportional to the stake of the validator. This will return the value that sits at the 50th percentile based on proportion of total stake. Such a measure appears to meet the necessary correctness criteria when applied to a stake-based approach.

## Recommendation

Consider adopting the weighted median instead of the median when computing timestamps.

## Location

- [snarkVM/ledger/narwhal/subdag/src/lib.rs](#)
- [snarkVM/ledger/narwhal/batch-certificate/src/lib.rs](#)

## Retest Results

### 2024-01-11 – Fixed

NCC Group reviewed [pull request 2223](#) (merged in [aa5e85e](#)), which updates the `timestamp()` function to return the weighted median (by stake) for a given committee. This effectively implements the recommendation above and ensures that the resulting timestamp is not substantially skewed by Byzantine behavior. Therefore, this finding is considered “Fixed”.



# Late/Distant Validation of Block Request Range

Overall Risk	Low	Finding ID	NCC-E009544-JR7
Impact	High	Component	snarkOS node
Exploitability	Undetermined	Category	Data Validation
		Status	Risk Accepted

## Impact

Late and/or distant validation of deserialized range values may result in oversights involving A) unexpected downstream behavior arising from (for example) an overflowing subtraction of unsigned integers in a block count calculation, B) enabling traffic amplification attacks arising from an overly large specified range, and/or C) corruption of internal data structures.

## Description

The `get_blocks()` function implemented within the `routes.rs` source file deserializes both a starting block height and an ending block height, and then *immediately* validates that the heights are strictly increasing and together specify a reasonable range. **This function is excerpted below and provides a positive example** of early (highlighted) validation checks.

```

104 // GET /testnet3/blocks?start={start_height}&end={end_height}
105 pub(crate) async fn get_blocks(
106     State(rest): State<Self>,
107     Query(block_range): Query<BlockRange>,
108 ) -> Result<ErasedJson, RestError> {
109     let start_height = block_range.start;
110     let end_height = block_range.end;
111
112     const MAX_BLOCK_RANGE: u32 = 50;
113
114     // Ensure the end height is greater than the start height.
115     if start_height > end_height {
116         return Err(RestError("Invalid block range".to_string()));
117     }
118
119     // Ensure the block range is bounded.
120     if end_height - start_height > MAX_BLOCK_RANGE {
121         return Err(RestError(format!(
122             "Cannot request more than {MAX_BLOCK_RANGE} blocks per call (requested {})",
123             end_height - start_height
124         )));
125     }
126     ...

```

Figure 10: `snarkOS/node/rest/src/routes.rs`

However, the `read_le()` function that implements corresponding functionality within the `block_request.rs` source file does not perform these validation checks. **This function is excerpted below and is the subject of this finding.** Note that the checks are not performed within the nearby `new()` function either.

```

53 impl FromBytes for BlockRequest {
54     fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
55         let start_height = u32::read_le(&mut reader)?;
56         let end_height = u32::read_le(&mut reader)?;

```

```

57
58     Ok(Self::new(start_height, end_height))
59 }
60 }

```

Figure 11: [snarkOS/node/bft/events/src/block\\_request.rs](#)

The usage of `BlockRequest` in the `inbound()` function implemented in the [gateway.rs](#) source file mitigates this as shown below.

```

53 Event::BlockRequest(block_request) => {
54     let BlockRequest { start_height, end_height } = block_request;
55
56     // Ensure the block request is well-formed.
57     if start_height >= end_height {
58         bail!("Block request from '{peer_ip}' has an invalid range ({start_height}..
59             ↳ {end_height}")
60     }
61     // Ensure that the block request is within the allowed bounds.
62     if end_height - start_height > DataBlocks::<N>::MAXIMUM_NUMBER_OF_BLOCKS as u32 {
63         bail!("Block request from '{peer_ip}' has an excessive range ({start_height}..
64             ↳ {end_height}")
65     }
66     ...

```

Figure 12: [snarkOS/node/bft/src/gateway.rs](#)

However, this validation is performed well beyond the point where this could be first detected (in the initial deserialization or `new()` functions). Future evolution of the code base may involve usage of `BlockRequest` instances that are not as well protected.

Similarly (and separately), the `BatchPropose` struct contains both a `batch_round` and `batch_header`, where the latter item also contains a `batch_round`. These two instances of `batch_round` could be validated to align during deserialization, but they are not.

## Recommendation

Consider validating information that crosses a trust boundary as close to the point of deserialization as possible. In the `read_le()` function excerpted above, implement early validation that `start_height` is less than `end_height` and that the two values specify a range of reasonable size. Validate the instances of `batch_round` at deserialization.

## Location

- Positive examples: `get_blocks()` within [snarkOS/node/rest/src/routes.rs](#)
- `read_le()` within [snarkOS/node/bft/events/src/block\\_request.rs](#)
- Mitigation [snarkOS/node/bft/src/gateway.rs](#)
- `BatchPropose` deserialization [snarkOS/node/bft/events/src/batch\\_propose.rs](#)

## Retest Results

### 2024-01-11 – Not Fixed

See client response.

## Client Response

The `BlockRequest` does not need to have a bound in the constructor. Because they are node/consensus dependent, the validation will be done by the recipient node.



## Brittle Error Handling of `NoiseState::Handshake`

Overall Risk Low  
Impact High  
Exploitability None

Finding ID NCC-E009544-Y4N  
Component snarkOS node  
Category Denial of Service  
Status Risk Accepted

### Impact

An attacker able to trigger an error condition leading to a panic may cause a denial of service.

### Description

The `encode()` function implemented in the `codec.rs` source file contains logic that encodes Events or Bytes subject to the current state of the Noise codec. Normal operation falls within the `NoiseState::Handshake` and `NoiseState::PostHandshake` conditions as partially excerpted below.

```
fn encode(&mut self, message_or_bytes: EventOrBytes<N>, dst: &mut BytesMut) -> Result<(), Self::Error> {
    let ciphertext = match self.noise_state {
        NoiseState::Handshake(ref mut noise) => {
            ... logic snipped...
        }
        NoiseState::PostHandshake(ref mut noise) => {
            ...logic snipped...
        }
        NoiseState::Failed => unreachable!("Noise handshake failed to encode"),
    };

    // Encode the resulting ciphertext using the length-delimited codec.
    self.codec.encode(ciphertext.freeze(), dst)
}
```

Figure 13: `snarkOS/node/bft/events/src/helpers/codec.rs`

In the above code snippet, the `encode()` function returns a (highlighted) `Result<(), Self::Error>` that allows calling logic to handle errors gracefully. However, the (highlighted) `NoiseState::Failed` condition utilizes the `unreachable!()` macro which results in an uncontrolled panic rather than utilizing the `Result` error path.

Codecs can fail from a variety of unexpected (and even intermittent) conditions. The well-structured `into_post_handshake_state()` function implemented on lines 150-170 of the same source file correctly handles and returns known positive results, while leaving a final non-positive return of `NoiseState::Failed` for everything else. If an external attacker is able to trigger this `NoiseState::Failed` condition in the above code, it may result in a denial of service.



---

It was understood from the kick-off call that the Noise codec is not yet utilized in production (likely due to intermittent failures). Thus, the exploitability of this finding has been rated “Undetermined” and the overall severity “Low”.

From a larger perspective, the code base was found to exhibit some inconsistencies in the way errors were handled. Any error conditions that can be triggered directly-or-indirectly immediately-or-later from the external environment should be handled gracefully and avoid a panic. Due to the complexity of this determination, it is preferable to err on the side of caution and gracefully handle errors wherever possible. That said, note that there are many instances of `unwrap()` which are paired with adjacent comments explaining their usage; this is a positive aspect of the code.

## Recommendation

Adapt the highlighted instance of the `NoiseState::Failed` condition to utilize the `Result` error path.

Consider prioritizing a defense-in-depth review of error handling in the consensus logic.

## Location

[snarkOS/node/bft/events/src/helpers/codec.rs](#)

## Retest Results

### 2024-01-12 – Not Tested

This finding was reported for completeness but relates to unused code. No updates or changes were made as a result and the exploitability of this finding is rated at “None”. As the code is unreachable in the current environment, this finding has been marked as “Risk Accepted”. Should this code be re-visited in the future, the recommendations made here remain valid.

## Client Response

During the November 16 status call, the developers noted that the Noise codec is not yet wired in and that the error arrangement may be as intended. As a result, the exploitability of this finding has been adjusted to “None” but the finding maintained for completeness.



# Secret Key Stored in Plaintext File, Missing Permission Checks

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E009544-K4R

Component snarkOS cli

Category Access Controls

Status Fixed

## Impact

A secret key stored in a group- or public-accessible plaintext file is subject to leakage, malicious use and/or (potentially) unbonding/slashing. Missing permission checks allow unrecognized key exposure to continue.

## Description

SnarkOS provides a permissionless and scalable network for ZK powered smart contracts. On a staked node, protecting the private keys is of paramount importance for continued participation, particularly as lower-skilled operators eventually join the network. Currently, the documentation and code allow private keys to be stored in world-readable plaintext files which increases the risk of leakage.

The code repository [README.md](#) file excerpted below indicates the ability to start the network with a private key stored in an accessible plaintext file.

```

USAGE:
  snarkos start [OPTIONS]

OPTIONS:
  ...
  --private-key <PRIVATE_KEY>          Specify the node's account private key
  --private-key-file <PRIVATE_KEY_FILE> Specify the path to a file containing the
  ↳ node's account private key
  ...

```

Figure 14: [snarkOS/README.md](#)

The corresponding `parse_private_key()` function implemented within the [start.rs](#) source file is used to parse the private key file, and is excerpted below.

```

209 /// Read the private key directly from an argument or from a filesystem location,
210 /// returning the Aleo account.
211 fn parse_private_key<N: Network>(&self) -> Result<Account<N>> {
212     match self.dev {
213         None => match (&self.private_key, &self.private_key_file) {
214             // Parse the private key directly.
215             (Some(private_key), None) => Account::from_str(private_key.trim()),
216             // Parse the private key from a file.
217             (None, Some(path)) => Account::from_str(std::fs::read_to_string(path)?.trim()),
218             // Ensure the private key is provided to the CLI, except for clients or nodes
219             ↳ in development mode.
220             (None, None) => match self.client {
221                 true => Account::new(&mut rand::thread_rng()),
222                 ...

```

Figure 15: [snarkOS/cli/src/commands/start.rs](#)

---

The highlighted line above does not check file permissions (necessary to issue a warning to the user) nor issue any warning related to overly permissive file permissions.

There are a variety of ways an attacker can exfiltrate these plaintext files. Physical removal of the disk and mounting on a system with different constraints (such as Kali Linux) may obviate any set permissions. Acquisition of any backups at any point in their lifecycle may also result in exposure. A Docker image may be configured (or exploited) to offer shell access. An attacker able to compromise the owner account or otherwise obtain root privileges on a running system will have straightforward access.

Note that this private key could be considered similar to an SSH key which is typically passphrase-protected. However, passphrase protection limits the ability of unattended booting of applications. Nonetheless, SSH user safety is improved by warnings on finding keys in files or directories with loose permissions<sup>1</sup>. The `.ssh` directory is required to have `0o700` (`drwx-----`) permissions and the private key file is required to have `0o600` (`-rw-----`) permissions, though arguably `0o400` would be sufficient. These permissions prevent access by other users or group members.

## Recommendation

There are several approaches to improve the security of secret keys, including:

- While environment variables are not ideal, they are typically less persistent and accessible than plaintext files. This technique is already incorporated into the `run-prover.sh` and `run-validator.sh` scripts, and could become the single recommended approach.
- A Docker node could contain a stored public key corresponding to a trusted external entity, and accept a properly signed value injected by that entity at startup/configuration time.
- Adapt the prover and validator to be run inside a Docker container and utilize provided mechanisms<sup>2</sup> for secret management. Kubernetes provides similar capabilities<sup>3</sup>.
- Utilize a built-for-purpose third-party component such as Hashicorp Vault<sup>4</sup>, AWS KMS<sup>5</sup> or Google's Secret Manager<sup>6</sup>.

If plaintext files must be used, implement a check within `parse_private_key()` for `0o700` permissions on the directory and `0o600` permissions on the file (the same as SSH).

As the product moves toward production, consider the overall lifecycle of key management<sup>7</sup> including the potential for key rotation.

## Location

`parse_private_key()` within `snarkOS/cli/src/commands/start.rs`

## Retest Results

### 2024-01-12 – Fixed

NCC Group reviewed [pull request 2998](#) (merged in [7a11bce](#)), which added a function `check_permissions()` to enforce that the folder permissions are `0o700` permissions on the key file are `0o600` (read and write for owner only) before proceeding and uses this function

1. <https://superuser.com/questions/215504/permissions-on-private-key-in-ssh-folder>
2. <https://docs.docker.com/engine/swarm/secrets/>
3. <https://kubernetes.io/docs/concepts/configuration/secret/>
4. <https://www.vaultproject.io/use-cases/secrets-management>
5. <https://aws.amazon.com/kms/>
6. <https://cloud.google.com/secret-manager>
7. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf>



---

in the affected code above. This change is consistent with the recommendations above, and as such, this finding is considered "Fixed".



# Event FromBytes Tolerates Extraneous Input

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E009544-UJF

Component snarkOS node

Category Data Validation

Status Fixed

## Impact

Tolerating extraneous bytes during deserialization may preclude the detection of malicious probing behavior and/or misaligned serialization-deserialization code between communicating parties.

## Description

The `from_le` function implemented on the `Event` struct tolerates left over bytes after the deserialization of events. In other words, the condition where there are still bytes remaining in the `reader` when execution approaches line 217 below is not detected or reported. Note that the `read_le` function below currently returns a (highlighted) `Result`.

```

192 impl<N: Network> FromBytes for Event<N> {
193     fn read_le<R: io::Read>(mut reader: R) -> io::Result<Self> {
194         // Read the event ID.
195         let id = u16::read_le(&mut reader).map_err(|_| error("Unknown event ID"));
196
197         // Deserialize the data field.
198         let event = match id {
199             0 => Self::BatchPropose(BatchPropose::read_le(reader)?),
200             1 => Self::BatchSignature(BatchSignature::read_le(reader)?),
201             2 => Self::BatchCertified(BatchCertified::read_le(reader)?),
202             3 => Self::BlockRequest(BlockRequest::read_le(reader)?),
203             4 => Self::BlockResponse(BlockResponse::read_le(reader)?),
204             5 => Self::CertificateRequest(CertificateRequest::read_le(reader)?),
205             6 => Self::CertificateResponse(CertificateResponse::read_le(reader)?),
206             7 => Self::ChallengeRequest(ChallengeRequest::read_le(reader)?),
207             8 => Self::ChallengeResponse(ChallengeResponse::read_le(reader)?),
208             9 => Self::Disconnect(Disconnect::read_le(reader)?),
209             10 => Self::PrimaryPing(PrimaryPing::read_le(reader)?),
210             11 => Self::TransmissionRequest(TransmissionRequest::read_le(reader)?),
211             12 => Self::TransmissionResponse(TransmissionResponse::read_le(reader)?),
212             13 => Self::ValidatorsRequest(ValidatorsRequest::read_le(reader)?),
213             14 => Self::ValidatorsResponse(ValidatorsResponse::read_le(reader)?),
214             15 => Self::WorkerPing(WorkerPing::read_le(reader)?),
215             16.. => return Err(error("Unknown event ID {id}")),
216         };
217
218         Ok(event)
219     }
220 }

```

Figure 16: `snarkOS/node/bft/events/src/lib.rs`

The condition of leftover input can occur during malicious probing or if there are implementation issues between communicating parties. From a defense-in-depth perspective, it is worthwhile detecting and reporting this condition.

---

## Recommendation

Incorporate a test for extraneous bytes after event deserialization by (for example) replacing line 218 with the following (and adding the `&mut` prefix to the prior 0..15 `reader` instances).

```
let mut overflow = [0u8; 1];
let err_finish = reader.read(&mut overflow);
if err_finish.is_err() | (err_finish.unwrap() == 0) {
    Ok(event)
} else {
    Err(error("Extraneous bytes in reader"))
}
```

## Location

- [snarkOS/node/bft/events/src/lib.rs](#)

## Retest Results

### 2024-01-11 – Fixed

NCC Group reviewed changes as part of [pull request 2994](#) (merged in [20f1d63](#)) which added checks to the `FromBytes()` function for both `Event` and `Message` that returns an error if there are leftover bytes after deserialization. As a result, this finding is considered “Fixed”.



# CLI Input Private Keys and Sensitive Values Not Zeroized

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E009544-7VR

Component snarkOS cli

Category Data Exposure

Status Fixed

## Impact

Failure to zeroize sensitive values may allow them to leak to other processes on the same system.

## Description

Various commands provided by the *snarkOS CLI* accept a private key as one of their input parameters. For example, the `Decrypt` struct wraps parameters for decryption:

```

26 // Decrypts a record ciphertext.
27 #[derive(Debug, Parser)]
28 pub struct Decrypt {
29     // The record ciphertext to decrypt.
30     #[clap(short, long)]
31     pub ciphertext: String,
32     // The view key used to decrypt the record ciphertext.
33     #[clap(short, long)]
34     pub view_key: String,
35 }

```

Figure 17: *cli/src/commands/developer/decrypt.rs*

The above is used to construct a `ViewKey`:

```

31 // The account view key used to decrypt records and ciphertext.
32 #[derive(Copy, Clone, Debug, PartialEq, Eq, Hash, Zeroize)]
33 pub struct ViewKey<N: Network>(Scalar<N>);

```

Figure 18: *snarkVM/console/account/src/view\_key/mod.rs*

It was observed that internal structs holding private keys, such as `ViewKey` and `PrivateKey`, derive the `Zeroize` trait to ensure that they are zeroized on drop. However, the structs in *CLI* which prepare these structs *do not* derive the `Zeroize` trait. For consistency, and to protect against memory-related attacks, it is recommended to consistently leverage the `Zeroize` trait for all sensitive values.

## Recommendation

Add `Zeroize` to the list of derived traits for `Account`, `Decrypt`, `Deploy`, `Execute`, `Scan`, and `TransferPrivate` structs.

## Location

- *cli/src/commands/developer/\**
- *cli/src/commands/account.rs*



---

## Retest Results

### 2024-01-11 – Fixed

NCC Group reviewed changes introduced as part of [pull request 2982](#) (merged in [1db968e](#)) that adds the `zeroize` crate and either implements or derives a function to zeroize the identified structs on drop. As a result, this finding is considered “Fixed”.





# Missing Validation of Transmission Response

Overall Risk	Low	Finding ID	NCC-E009544-UPV
Impact	Low	Component	snarkOS node
Exploitability	Medium	Category	Data Validation
		Status	Fixed

## Impact

An adversary able to respond to transmission requests is able to corrupt the pending queue, potentially impacting timeout logic.

## Description

The `worker.rs` source file implements the `send_transmission_request()` function to send a transmission request to a remote peer, the `send_transmission_response()` function for that remote peer to reply, and the `finish_transmission_request()` function for the originator to handle the peer's reply. The latter function is excerpted below with a `TODO` comment highlighted that notes a missing validation on the reply.

```

394 /// Handles the incoming transmission response.
395 /// This method ensures the transmission response is well-formed and matches the
    ↳ transmission ID.
396 fn finish_transmission_request(&self, peer_ip: SocketAddr, response:
    ↳ TransmissionResponse<N>) {
397     let TransmissionResponse { transmission_id, transmission } = response;
398     // Check if the peer IP exists in the pending queue for the given transmission ID.
399     let exists = self.pending.get(transmission_id).unwrap_or_default().contains(&peer_ip);
400     // If the peer IP exists, finish the pending request.
401     if exists {
402         // TODO: Validate the transmission.
403         // TODO (howardwu): Deserialize the transmission, and ensure it matches the
            ↳ transmission ID.
404         // Note: This is difficult for testing and example purposes, since those
            ↳ transmissions are fake.
405         // Remove the transmission ID from the pending queue.
406         self.pending.remove(transmission_id, Some(transmission));
407     }
408 }

```

Figure 19: `snarkOS/node/bft/src/worker.rs`

A honest remote peer may receive a transmission request and reply with a `TransmissionResponse` as declared in the `transmission_response.rs` source file as excerpted below.

```

18 pub struct TransmissionResponse<N: Network> {
19     pub transmission_id: TransmissionID<N>,
20     pub transmission: Transmission<N>,
21 }

```

Figure 20: `snarkOS/node/bft/events/src/transmission_response.rs`

However, a malicious peer may respond with the correct `transmission_id` but with different and malicious contents of `transmission`. As a result, the `remove()` function on line 406 above is called with unexpected/mismatched values. The remove function will indeed remove the correct pending item based on information within `transmission_id`, but may then

---

send a notification to an unexpected/different callback (based on information within `transmission`).

It appears that in this context, callbacks only relate to timeout functionality so the impact is limited. Further, the `TODO` comment suggests this is a known issue being worked on. For these reasons, the overall severity of this finding is marked 'Low' and it is presented to refresh awareness and maintain prioritization.

## Recommendation

The contents of the `TransmissionResponse` struct includes replicated `transmission_id` information that should be validated when crossing a trust boundary. The adjacent comment indicates test challenges that may be addressed by an instance of `#[cfg(test)]` or `cfg!(test)`<sup>8</sup>.

## Location

- The `finish_transmission_request()` function in `snarkOS/node/bft/src/worker.rs`
- The `TransmissionResponse` struct in `snarkOS/node/bft/events/src/transmission_response.rs`

## Retest Results

### 2023-11-15 – Fixed

[Pull Request 2487](#) (merged in [8de3abd](#)) adds validation of the transmission response fields via the `ensure_transmission_id_matches()` function implemented in the `ledger.rs` source file as recommended. Therefore, this finding is considered "Fixed".

---

8. <https://doc.rust-lang.org/std/macro.cfg.html>



# Potentially Dangerous Handling of Duplicate Signatures on Certificates

Overall Risk Low

Impact Low

Exploitability Medium

Finding ID NCC-E009544-76W

Component snarkOS node, snarkVM  
batch-certificate

Category Cryptography

Status Fixed

## Impact

A signer may sign a certificate multiple times, and the author of a batch may additionally sign their own certificate. Such behavior does not affect consensus but may lead to divergent behavior in future implementations / clients.

## Description

This finding highlights potentially dangerous assumptions about the validity of incoming data, and highlights potential missing validation checks on said data. No ability to violate consensus was identified, but the current approach may be exploitable to induce artificial workloads on validators via redundant certificate validation or increased storage requirements.

A certificate consists of a batch header and a set of signatures, where the batch header contains a signature from the author of the batch. For consensus purposes, the set of signers includes both the author and the other peers that signed the certificate. The following code is used to deserialize a certificate:

```

46     else if version == 2 {
47         // Read the batch header.
48         let batch_header = BatchHeader::read_le(&mut reader?);
49         // Read the number of signatures.
50         let num_signatures = u16::read_le(&mut reader?);
51         // Read the signatures.
52         let mut signatures = IndexSet::with_capacity(num_signatures as usize);
53         for _ in 0..num_signatures {
54             // Read the signature.
55             let signature = Signature::read_le(&mut reader?);
56             // Insert the signature.
57             signatures.insert(signature);
58         }
59         // Return the batch certificate.
60         Self::from(batch_header, signatures).map_err(error)
61     }

```

Figure 21: [ledger/narwhal/batch-certificate/src/bytes.rs](#)

The resulting signatures are individually verified before being passed back to the caller:

```

93     /// Initializes a new batch certificate.
94     pub fn from(batch_header: BatchHeader<N>, signatures: IndexSet<Signature<N>>) ->
95     ↪ Result<Self> {
96         // Verify the signatures are valid.
97         for signature in &signatures {
98             if !signature.verify(&signature.to_address(), &[batch_header.batch_id()]) {

```

```

98         bail!("Invalid batch certificate signature")
99     }
100 }
101 // Return the batch certificate.
102 Self::from_unchecked(batch_header, signatures)
103 }

```

Figure 22: [ledger/narwhal/batch-certificate/src/lib.rs](#)

Within SnarkOS, the logic is primarily concerned with the signers themselves, and not the individual signatures, where a `HashSet` is used to represent the set of signers, and the signer address is derived from the signature, e.g.:

```

95 // Returns the signers.
96 pub fn signers(&self) -> HashSet<Address<N>> {
97     self.signatures.iter().chain(Some(self.batch_header.signature())).map(
98         ↪ Signature::to_address).collect()
99 }

```

Figure 23: [node/bft/src/helpers/proposal.rs](#)

Similarly, when checking the validity of a certificate, a complete set of the signers is constructed:

```

460 // Initialize a set of the signers.
461 let mut signers = HashSet::with_capacity(certificate.signatures().len() + 1);
462 // Append the batch author.
463 signers.insert(certificate.author());
464
465 // Iterate over the signatures.
466 for signature in certificate.signatures() {
467     // Retrieve the signer.
468     let signer = signature.to_address();
469     // Ensure the signer is in the committee.
470     if !previous_committee.is_committee_member(signer) {
471         bail!("Signer {signer} is not in the committee for round {round} {gc_log}")
472     }
473     // Append the signer.
474     signers.insert(signer);
475 }

```

Figure 24: [node/bft/src/helpers/storage.rs](#)

The use of a `HashSet` ensures that each signer only appears once in the list of signers and is similarly only included once for the purposes of availability or quorum threshold computation. However, the use of a `HashSet` causes duplicate signatures for the same signer to be silently ignored. There are several remarks that can be made about the above approach:

1. Duplicate signatures within the serialized data are tolerated, but silently dropped during deserialization due to the use of an `IndexSet`. One could check if `signatures.insert(signature)` returns `false` or if the resulting length of the set is equal to `num_signatures` and throw a warning or error if this is not the case.
2. Duplicate signers within the serialized data are tolerated and will only be counted once for threshold purposes but will have each individual signature validated.
3. The author of a certificate may add a signature to the certificate as well. The function `from_unchecked` ensures at least one signature is present, which is validated in `from()`. It follows that a single validator may produce a certificate on their own which



---

cryptographically validates within SnarkOS. This *does not* allow the author to be counted twice for threshold purposes.

4. The usage of sets may introduce implied assumptions about the order in which data is processed. While the current usage appears to be safe, future functionality or alternative clients may not make the same assumptions about signature processing.

As a concrete example, the function `signers()` excerpted above computes over `self.signatures.iter().chain(Some(self.batch_header.signature()))`, which considers the list of signatures followed by the author. In contrast, the `check_certificate()` function includes the author first, followed by the remaining signers. The same logic applied to signatures instead of signers would result in a different set of signatures being validated, which may introduce unintended or incorrect behavior. This does not affect consensus, as the individual signatures must all be valid. But it does suggest that two different implementations may compute over two distinct sets of data based on how the above situations are handled.

As a final consideration, the function `add_signature()`, used to add a signature to a batch, will explicitly reject a duplicate signer, and the calling function will throw an error before attempting to add a signature for the current primary (i.e., the author). The observations above can be summarized by noting that these same constraints are not enforced on incoming data.

## Recommendation

Consider stronger consistency checks on certificates, such as ensuring each signature is from a unique signer and not from the author.

## Location

- [node/bft/src/helpers/proposal.rs](#)
- [node/bft/src/helpers/storage.rs](#)
- [ledger/narwhal/batch-certificate/src/bytes.rs](#)
- [ledger/narwhal/batch-certificate/src/lib.rs](#)

## Retest Results

### 2024-01-11 – Fixed

NCC Group reviewed changes as part of [pull request 2298](#) (merged in [b062f9e](#)) which add additional checks to the set of signers. In particular, the use of an `IndexSet` has been changed to `HashSet` during deserialization, an explicit check to ensure the author is not among the list of signers has been added, and the number of signatures in the final `HashSet` is compared against the expected number of signatures to ensure that no duplicates are present. These changes collectively implement the recommendations made above, and this finding is therefore considered “Fixed”.



# Weak Malicious Peer Handling

Overall Risk	Low	Finding ID	NCC-E009544-LA4
Impact	Low	Component	snarkOS node
Exploitability	Medium	Category	Session Management
		Status	Not Fixed

## Impact

Nodes actively disconnect from peers when malicious behavior is detected but do not track which peers have acted maliciously in the past. At each network heartbeat, the peer may actively reconnect to a malicious peer.

## Description

At various points in the `Primary` structure, a proposal, batch, or certificate is checked for consistency, and malicious behavior is detected and handled. For example:

```

620 // Retrieve the signer.
621 let signer = spawn_blocking!(Ok(signature.to_address()))?;
622 // Ensure the batch signature is signed by the validator.
623 if self.gateway.resolver().get_address(peer_ip).map_or(true, |address| address !=
↳ signer) {
624     // Proceed to disconnect the validator.
625     self.gateway.disconnect(peer_ip);
626     bail!("Malicious peer - batch signature is from a different validator ({{signer}})");
627 }
628 // Ensure the batch signature is not from the current primary.
629 if self.gateway.account().address() == signer {
630     bail!("Invalid peer - received a batch signature from myself ({{signer}})");
631 }

```

Figure 25: [node/bft/src/primary.rs](#)

Here, a peer that sends a batch signature that does not match their address is assumed to be malicious and a disconnect is triggered. However, no proactive action is taken to prevent this peer from reconnecting in the future. In many cases, a reconnection will be automatically established at each network heartbeat. For example, if the malicious peer appears on the trusted validator list, then a reconnection will be triggered at every heartbeat as part of `handle_trusted_validators()`. If the malicious node is not a trusted validator and the number of connected validators is below `MIN_CONNECTED_VALIDATORS`, then the function `handle_min_connected_validators()` will request a list of validator IPs from a random connected validator, which will in turn trigger a connection attempt to each validator in the `ValidatorsResponse` message.

The implemented behavior when encountering malicious behavior appears to be at odds with the implemented approach for maintaining an active connection to at least `MIN_CONNECTED_VALIDATORS`. If malicious behavior warrants an active disconnection from a peer, then it may be necessary to actively prevent reconnection for some well-defined period of time. A list of recent malicious peers could be maintained, and reconnection to any peer not on this list could be prioritized over suspected malicious peers. Care must be taken to ensure that any response to misbehavior cannot be triggered by a malicious peer attempting to frame an honest peer for misbehavior.

---

It was noted that a `restricted_peers` list does exist as part of the router, but its use is limited to tracking connection failures above the `MAXIMUM_CONNECTION_FAILURES` threshold.

At a system level, it was noted that although peers locally observe potentially malicious behavior, there does not appear to be any disincentive to such behavior, aside from a temporary disconnection. Ethereum, for example, introduces penalties for observed malicious behavior, [where the stake of malicious nodes is slashed when misbehavior occurs](#).

### Recommendation

- Track which peers have triggered a disconnection for potential malicious activity and prioritize connections to validators that do not appear on this list.
- Carefully consider any changes relating to misbehavior response to ensure that honest peers cannot be mistakenly or maliciously identified and punished for misbehavior.

### Location

- `node/bft/src/primary.rs`

### Retest Results

**2024-01-16 – Not Fixed**

See client response.

### Client Response

Won't address for now. Will focus on tackling peer/validator reputation scores and malicious behavior in the future.



# Leader Election Process Does Not Match Whitepaper

Overall Risk Informational

Impact None

Exploitability None

Finding ID NCC-E009544-VFD

Component snarkVM ledger

Category Other

Status Fixed

## Impact

Discrepancies between documentation and implementation may represent unintended behavior or introduce confusion.

## Description

To support the review, a draft of *The Aleo Whitepaper* was provided, which specifies the following regarding leader election:

The leader election algorithm takes into account the current round number, the number of validators, and the amount of stake bonded to each validator.

The corresponding implementation of this in *SnarkVM* does not include the number of validators, but does include the starting round:

```

160 pub fn get_leader(&self, current_round: u64) -> Result<Address<N>> {
161     // Ensure the current round is at least the starting round.
162     ensure!(current_round >= self.starting_round, "Current round must be at least the
↳ starting round");
163     // Retrieve the total stake of the committee.
164     let total_stake = self.total_stake();
165     // Construct the round seed.
166     let seed = [self.starting_round, current_round, total_stake].map(Field::from_u64);
167     ...

```

Figure 26: *snarkVM/ledger/committee/src/lib.rs*

This `seed` is later used to select the leader at random, weighted by their total stake. The consensus mechanism and reward payout does not rely on the unpredictability of this value, which suggests that both the implemented and documented approaches are appropriate.

## Recommendation

Validate that the implemented approach is the intended approach and ensure that the documentation and implementation are aligned prior to publication.

## Location

- *snarkVM/ledger/committee/src/lib.rs*

## Retest Results

2024-01-11 – Fixed

This finding highlighted a discrepancy between the implementation and a *draft* paper explaining the implemented approach. Aleo has confirmed that the implemented approach is correct and that the paper will be updated accordingly. As no code changes are required as a result, this finding is considered “Fixed”.



---

## Client Response

The intention is the same, the whitepaper will be updated to have more clear verbiage. The difference is that the implementation does not use the “number of validators”, but just the validator set itself.



## 5 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.



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



## 6 Engagement Notes

This informational section contains a number of notes and observations on the design and implementation of snarkOS.

### General Notes and Concerns on the Topic of Staked Validators in a Dynamic Committee Setting

The consensus mechanism underpinning Aleo's snarkOS implementations is a generic DAG BFT protocol (Bullshark<sup>9</sup> and Narwhal<sup>10</sup>), which was converted to a proof-of-stake system and additionally modified to support a dynamic committee. While no straightforward incompatibilities seem to result from these modifications, this system does blur the lines between stakes and validators. BFT protocols such as the one used in *snarkOS* are commonly proven to function correctly as long as the number of Byzantine nodes does not exceed 1/3 of the total number of nodes in the system. That is, given a total number of participants  $N = 3f + 1$ , the system behaves normally as long as at most  $f$  participants are Byzantine. Translating this concept to a proof-of-stake system implies that for the system to proceed, given a total amount of stake equal to  $N$ , validators amounting to at least  $2f + 1$  stake should be in consensus. These types of checks are performed in a few areas in the code base, but ultimately resolve to the function `is_quorum_threshold_reached()` in the ledger component of *snarkVM*, in `snarkVM/ledger/committee/src/lib.rs` and provided below for reference. This function sums up the combined stakes of the given set of addresses and checks whether that amount is enough to reach the quorum threshold,  $2f + 1$ .

```
115 // Returns `true` if the combined stake for the given addresses reaches the quorum
    ↳ threshold.
116 // This method takes in a `HashSet` to guarantee that the given addresses are unique.
117 pub fn is_quorum_threshold_reached(&self, addresses: &HashSet<Address<N>>) -> bool {
118     let mut stake = 0u64;
119     // Compute the combined stake for the given addresses.
120     for address in addresses {
121         // Accumulate the stake, checking for overflow.
122         stake = stake.saturating_add(self.get_stake(*address));
123     }
124     // Return whether the combined stake reaches the quorum threshold.
125     stake >= self.quorum_threshold()
126 }
```

Figure 27: `ledger/committee/src/lib.rs`

A positive example of this process happens in the function `propose_batch()` in the file `node/bft/src/primary.rs`, where a batch will not be proposed until the quorum threshold has been reached.

```
340 // If quorum threshold is not reached, return early.
341 if !committee.is_quorum_threshold_reached(&connected_validators) {
342     debug!(
343         "Primary is safely skipping a batch proposal {}",
344         "(please connect to more validators)".dimmed()
345     );
346     trace!("Primary is connected to {} validators", connected_validators.len() - 1);
347     return Ok(());
348 }
```

Figure 28: `node/bft/src/primary.rs`

9. <https://arxiv.org/abs/2209.05633>

10. <https://arxiv.org/pdf/2105.11827.pdf>



---

In order to assess the security of the modifications to the consensus protocol, it is crucial to identify areas in the design *and in the implementation* of the protocol where important processes hinge upon the *number* of validators when they should instead rely on the stakes of these validators. One such issue was described in [finding "Timestamp Calculation Does Not Provide Byzantine Fault Tolerance"](#), where it was observed that the timestamp computation was not weighted by the respective stakes of the validators, but it was equally weighted among them.

Another area where this distinction appears to be unclear is in the underlying communication protocol, as implemented by the `Gateway` in `node/bft/src/gateway.rs`. The gateway is in charge of maintaining communication with other validators in the system. The gateway also defines an upper bound and a lower bound on the number of validators that it maintains connections with, as can be observed in the following variables:

```
85  /// The minimum number of validators to maintain a connection to.
86  const MIN_CONNECTED_VALIDATORS: usize = 175;
87  /// The maximum number of validators to send in a validators response event.
88  const MAX_VALIDATORS_TO_SEND: usize = 200;
```

Figure 29: `node/bft/src/gateway.rs`

The gateway frequently ensures it is connected to a minimum number of validators, by way of the `handle_min_connected_validators()` function called during the heartbeat process of the gateway, see below.

```
880 /// This function sends a `ValidatorsRequest` to a random validator,
881 /// if the number of connected validators is less than the minimum.
882 fn handle_min_connected_validators(&self) {
883     // If the number of connected validators is less than the minimum, send a
884     ↳ `ValidatorsRequest`.
885     if self.number_of_connected_peers() < MIN_CONNECTED_VALIDATORS {
886         // Retrieve the connected validators.
887         let validators = self.connected_peers().read().clone();
888         // If there are no validator IPs to connect to, return early.
```

Figure 30: `node/bft/src/gateway.rs`

*Theoretically*, being connected to a minimum number of validators does not ensure that their combined stake is enough to reach the quorum threshold. As such, a validator could essentially be connected to a majority of Byzantine nodes (in terms of stake), in a form of Eclipse attack<sup>11</sup>, which could break some of the guarantees of the system.

The NCC Group team further encourages Aleo to perform a pass throughout the code base to try and identify areas where potential assumption discrepancies between the number of validators and their stake occur.

### Instability of the System at Genesis

Genesis of the system starts with 4 validators, each with an equal amount of stake. However, the system does not reward validators whose stake is larger than 25% of the total stake, in an apparent effort to incentivize greater stake distribution and larger number of validators.

```
26 /// Returns the updated stakers reflecting the staking rewards for the given committee and
27 ↳ block reward.
28 /// The staking reward is defined as: `block_reward * stake / total_stake`.
29 ///
```

---

11. <https://research.nccgroup.com/2023/06/02/how-to-spot-and-prevent-an-eclipse-attack/>



```

29 /// This method ensures that stakers who are bonded to validators with more than **25%**
30 /// of the total stake will not receive a staking reward. In addition, this method
31 /// ensures stakers who have less than 10 credit are not eligible for a staking reward.
32 ///
33 /// The choice of 25% is to ensure at least 4 validators are operational at any given time,
34 /// since our security model adheres to 3f+1, where f=1. As such, we tolerate Byzantine
    ↳ behavior
35 /// up to 33% of the total stake.
36 pub fn staking_rewards<N: Network>(
37     stakers: &IndexMap<Address<N>, (Address<N>, u64)>,
38     committee: &Committee<N>,
39     block_reward: u64,
40 ) -> IndexMap<Address<N>, (Address<N>, u64)> {
41     // If the list of stakers is empty, there is no stake, or the block reward is 0, return
    ↳ the stakers.
42     if stakers.is_empty() || committee.total_stake() == 0 || block_reward == 0 {
43         return stakers.clone();
44     }
45
46     // Compute the updated stakers.
47     cfg_iter!(stakers)
48         .map(|(staker, (validator, stake))| {
49         // If the validator has more than 25% of the total stake, skip the staker.
50         if committee.get_stake(*validator) > committee.total_stake().saturating_div(4) {
51             trace!("Validator {validator} has more than 25% of the total stake -
    ↳ skipping {staker}");
52             return (*staker, (*validator, *stake));
53         }

```

Figure 31: [synthesizer/src/vm/helpers/rewards.rs](#)

In a genesis state with 4 validators, each possessing exactly 25% of the total stake, rewards will still be paid out to these validators. However, as soon as the stake of any single validator is increased, this validator will necessarily possess more than 25% of the total stake and will cease to receive rewards. Should the network continue to progress in this state, the stake of the remaining validators will increase over time until they each surpass 25% of the total stake, at which point they will cease to be paid rewards and the situation inverts. From this point on, the set of validators earning rewards will oscillate as they cross and then subsequently fall behind this threshold.

While it is understood that a network with only 4 validators may not be considered resilient or trustworthy, the above behavior may be considered unintuitive, or even undesirable. It also suggests that in corner cases, there may be incentive for “malicious staking” of tokens. For example, at the genesis state, a malicious validator could lower their stake slightly such that the remaining validators each possess more than 25% of the stake. This action does not increase the rewards for the malicious node but may nevertheless be seen as a successful attack when relative rewards are compared.

For correctness, a minimum of 4 validators are necessary, however the above observations (as well as many similar variants on the above), suggest that for stability it may be desirable for the genesis state to consist of at least 5 or more validators such that no validator is required to forego rewards during the initial blocks of the network. It may also be beneficial to consider under what scenarios an attack involving “malicious staking” might allow an attacker to disrupt or undermine trust in the system.



## Unpruned Restricted Peer List in Router

The `InnerRouter` structure defined in `node/router/src/lib.rs`, which is responsible for routing messages between nodes, defines and maintains a number of lists of peers. Among these lists, the `restricted_peers` keeps track of misbehaving peers and maps their address to their latest offending action.

```
86  /// The set of restricted peer IPs.
87  restricted_peers: RwLock<IndexMap<SocketAddr, Instant>>,
```

Figure 32: `node/router/src/lib.rs`

The NCC Group team noted that this list could grow arbitrarily and did not seem to get pruned regularly. The only place where a peer may be removed from that list happens in the function `insert_connected_peer()`, see the highlighted line in the code excerpt below.

```
/// Inserts the given peer into the connected peers.
pub fn insert_connected_peer(&self, peer: Peer<N>, peer_addr: SocketAddr) {
    let peer_ip = peer.ip();
    // ...
    // Remove this peer from the restricted peers, if it exists.
    self.restricted_peers.write().remove(&peer_ip);
}
```

Figure 33: `node/router/src/lib.rs`

In order to check if a peer is restricted, which is performed when trying to connect to a new peer, the function `is_restricted()` fetches the entry for the given IP address, and declares the peer restricted if they have been inserted in the list less than `RADIO_SILENCE_IN_SECS` seconds ago (currently set to 2.5 minutes), see below.

```
283  /// Returns `true` if the given IP is restricted.
284  pub fn is_restricted(&self, ip: &SocketAddr) -> bool {
285      self.restricted_peers
286          .read()
287          .get(ip)
288          .map(|time| time.elapsed().as_secs() < Self::RADIO_SILENCE_IN_SECS)
289          .unwrap_or(false)
290  }
```

Figure 34: `node/router/src/lib.rs`

Thus, the list will eventually contain a large number of stale “restricted” peers, but no longer considered *restricted*. It would be advisable to add a frequent pruning step to that list, which would remove entries that are no longer considered restricted. This would prevent the list from growing arbitrarily, and avoid the redundant work of repeatedly checking whether a given peer is restricted based on the `RADIO_SILENCE_IN_SECS` delay.

## Unpruned Connecting Peer List in Router

In the same file, it seems that the `connecting_peer` list is not properly pruned once we have successfully connected to a *trusted* peer. In the function `connect()`, a tentative peer is added to the `connecting_peers` list in the function call `check_connection_attempt()` (first line highlighted below). If the connection is unsuccessful, the peer is removed from the `connecting_peers` list. However, if the connection is successful, it seems that the peer will not be removed from that list.

```
136  pub fn connect(&self, peer_ip: SocketAddr) -> Option<JoinHandle<bool>> {
137      // Return early if the attempt is against the protocol rules.
138      if let Err(forbidden_message) = self.check_connection_attempt(peer_ip) {
```



```

139     warn!("{forbidden_message}");
140     return None;
141 }
142
143 let router = self.clone();
144 Some(tokio::spawn(async move {
145     // Attempt to connect to the candidate peer.
146     match router.tcp.connect(peer_ip).await {
147         // Remove the peer from the candidate peers.
148         Ok(()) => {
149             router.remove_candidate_peer(peer_ip);
150             true
151         }
152         // If the connection was not allowed, log the error.
153         Err(error) => {
154             router.connecting_peers.lock().remove(&peer_ip);
155             warn!("Unable to connect to '{peer_ip}' - {error}");
156             false
157         }
158     }
159 })
160 }

```

Figure 35: *node/router/src/lib.rs*

In the case where the connection is performed as part of the handshake process, the peer will eventually be removed from that list. However, there are instances where the `connect()` function is called outside of the handshake, such as in the `handle_trusted_peers()` function in *node/router/src/heartbeat.rs*. In that case, it appears that the trusted peer will not be properly removed from the `connecting_peer` list.

### Static Lists of Trusted Peers

Still in the file *node/router/src/lib.rs*, the router also maintains a list of trusted peers which is provided upon instantiation of that object. That list is essentially static; once peers have been added to it upon creation, they won't ever be removed and will always be considered trusted. The same can be said about the bootstrap peers, located in the function `bootstrap_peers()`.

```

372 /// Returns the list of bootstrap peers.
373 pub fn bootstrap_peers(&self) -> Vec<SocketAddr> {
374     if cfg!(feature = "test") || self.is_dev {
375         vec![]
376     } else {
377         vec![
378             SocketAddr::from_str("35.224.50.150:4133").unwrap(),
379             SocketAddr::from_str("35.227.159.141:4133").unwrap(),
380             SocketAddr::from_str("34.139.203.87:4133").unwrap(),
381             SocketAddr::from_str("34.150.221.166:4133").unwrap(),
382         ]
383     }
384 }

```

Figure 36: *node/router/src/lib.rs*

In the event that one of these trusted peers becomes untrusted, due to an unforeseen change such as their IP address getting re-allocated, the router does not have any mechanism in place to remove trust in them.





## Late Detection of Disconnected Peer

The function `process_batch_signature_from_peer()` performs several checks for potential malicious behavior prior to acquiring a write lock, followed by a check for whether or not the batch has previously been seen, which is eventually followed by a check that the signer is a currently connected peer:

```
654     let Some(signer) = self.gateway.resolver().get_address(peer_ip) else {
655         bail!("Signature is from a disconnected validator");
656     };
```

Figure 37: `node/bft/src/primary.rs`

In comparison, `process_batch_certificate_from_peer()` combines several of these peer checks and terminates early if the signer is disconnected:

```
711     // Ensure the batch certificate is from the validator.
712     match self.gateway.resolver().get_address(peer_ip) {
713         // If the peer is a validator, then ensure the batch certificate is from the
714         // ↳ validator.
715         Some(address) => {
716             if address != author {
717                 // Proceed to disconnect the validator.
718                 self.gateway.disconnect(peer_ip);
719                 bail!("Malicious peer - batch certificate from a different validator
720                 ↳ ({author})");
721             }
722         }
723     }
724     None => bail!("Batch certificate from a disconnected validator"),
725 }
```

Figure 38: `node/bft/src/primary.rs`

It may be beneficial to align these two approaches such that detectable errors are handled as early as possible, particularly before entering a critical section of the code.

## Minor Error in Code Comment

The comment below refers to a `timestamp` instead of a `batch_id`, which appears to be a reference to the deprecated V1 `BatchSignature` format.

```
617     // Retrieve the signature and timestamp.
618     let BatchSignature { batch_id, signature } = batch_signature;
```

Figure 39: `node/bft/src/primary.rs`

## Outdated Reference to candidate peers in Code Comment

In the file `gateway.rs`, the function `remove_connected_peer()` is preceded by a comment referring to a list of `candidate peers`. The gateway currently only maintains lists of `connected_peers` and `connecting_peers`; `candidate peers` seems to be an outdated reference.

```
448     /// Removes the connected peer and adds them to the candidate peers.
449     fn remove_connected_peer(&self, peer_ip: SocketAddr) {
```

Figure 40: `node/bft/src/gateway.rs`



## Unclear Repeated Hash Function Call in `assign_to_worker`

In `node/bft/src/helpers/partition.rs`, the function `assign_to_worker()` is used to assign an operation to a given `worker`, in order to distribute the work equally among processes. To do so, it computes a hash of the transmission ID and reduces it to a value in the range of the number of workers, see highlighted lines below.

```
38 /// Returns the worker ID for the given transmission ID.
39 pub fn assign_to_worker<N: Network>(transmission_id: impl Into<TransmissionID<N>>,
↳ num_workers: u8) -> Result<u8> {
40     // If there is only one worker, return it.
41     if num_workers == 1 {
42         return Ok(0);
43     }
44     // Hash the transmission ID to a u128.
45     let hash = sha256d_to_u128(&transmission_id.into().to_bytes_le()?);
46     // Convert the hash to a worker ID.
47     let worker_id = (hash % num_workers as u128) as u8;
48     // Return the worker ID.
49     Ok(worker_id)
50 }
```

Figure 41: `node/bft/src/helpers/partition.rs`

The hash function call `sha256d_to_u128()` on line 45 performs a double SHA256 operation: it hashes the transmission ID and hashes the resulting digest again. It is unclear what benefits are obtained with this second hash function call. Performing a single hash function call and incorporating some randomness would presumably achieve similar results, with the additional advantage of being less predictable by adversaries.