Espresso

Security Review

Cantina Managed review by:

Guido Vranken, Lead Security Researcher

0xRajeev, Lead Security Researcher

Phaze, Security Researcher

January 10, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Espresso Network is a shared source of truth that provides strong confirmations on transaction ordering and data across chains. The confirmations provided by HotShot are additive, meaning that rollups can keep giving their users pre-confirmations using their own existing sequencer.

From Dec 11th to Dec 23nd the Cantina team conducted a review of gsc and espresso-monorepo on commit hashes 7149813b and b96d65c2 respectively. The team identified a total of **23** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	3	3	0
Low Risk	12	10	2
Gas Optimizations	0	0	0
Informational	7	5	2
Total	**23**	**19**	**4**

3 Findings

3.1 GSC

3.1.1 Medium Risk

3.1.1.1 Anyone with access to the machine hardware can tamper with keystore or configuration files **Severity:** Medium Risk

Context: nitro-espresso.manifest#L16

Description: Anyone with access to the machine hardware can tamper with the L1 keystore or configuration files related to Batch Poster.

From the gramine docs, `allowed-files` "*are allowed to be created or opened in the enclave unconditionally. In other words, allowed files, directories and devices can be opened for reading/writing and can be created if they do not exist already. Allowed files are not cryptographically hashed and are thus not protected.*" This has an associated warning that: "*It is insecure to allow files containing code or critical information; developers must not allow files blindly! Instead, use trusted or encrypted files.*"

However, `nitro-espresso.manifest` has `sgx.allowed_files = ["file:/home/user/.arbitrum", "file:/home/user/kzg10-aztec20-srs-1048584.bin", "file:/config", "file:/l1keystore"]`. The `l1keystore` will include the signing keys among others and the `config` may include critical configuration parameters that affect the liveness and security of the system.

Impact: High, because anyone with access to the machine hardware can tamper with the keystore or configuration to affect liveness and security guarantees of the Batch Poster.

Likelihood: Low, because access to machine hardware is presumably guarded with the highest level of OpSec best practices.

Recommendation: Consider using `trusted` or `encrypted` files instead of `allowed` files for anything that has critical information as noted in the Gramine documentation.

Espresso: `file:/home/user/.arbitrum` does not contain important information, it only contains the cache of state of batch poster. Even if this state is deleted, batch poster can re-sync the entire state from L1. Also Batch poster writes to this folder and trusted files cannot be used for this. The rest we will add to the trusted files

See PR 3.

Cantina Managed: `file:/l1keystore` will be moved to a remotely hosted third-party KMS solution and `file:/home/user/.arbitrum` doesn't need increased protection.

But in PR 3, nitro-espresso.manifest#L11-L14, we see:

```
sgx.allowed_files = ["file:/home/user/.arbitrum", "file:/home/user/kzg10-aztec20-srs-1048584.bin",
↪  "file:/config"]
```

where

```
"file:/home/user/kzg10-aztec20-srs-1048584.bin"
```

has a `uri` and `sha256` under `sgx.trusted_files`.

Shouldn't:

1. `file:/home/user/kzg10-aztec20-srs-1048584.bin` be removed from `sgx.allowed_files` list? Not sure if `sgx.trusted_files` overrides that or the categorization is based on the initial listing.
2. `file:/config` also have a `uri` and `sha256` under `sgx.trusted_files` and be removed from `sgx.allowed_files` list?

Espresso: We are adding the config folder in CI here sgx-poster.yml#L65. The reason for doing it in CI is because each chain will have a different config, so we didnt want to hard code it.

We updated the manifest and deleted these files from the manifest

Cantina Managed: Fixed.

3.1.2 Low Risk

3.1.2.1 Enabling the `flock` system call in Gramine Shielded Containers manifest is risky **Severity:** Low Risk

Context: nitro-espresso.manifest#L8

Description: Given the use of Gramine Shielded Containers (GSC) to execute Batch Poster inside an Intel SGX enclave, the corresponding manifest file `nitro-espresso.manifest` specifies some configuration parameters to override the default values.

`sys.experimental__enable_flock = true` enables the use of the `flock` system call in Gramine. However, the GSC Manifest documentation has a warning that says:

```
This syscall is still under development and may contain security vulnerabilities. This is temporary; the
↪  syscall will be enabled by default in the future after thorough validation and this syntax will be removed
↪  then.
```

This warning suggests that enabling the use of `flock` system call is currently risky. However, the level of impact/likelihood is underdetermined at this point.

Recommendation: Consider if `flock` enabling is required and any implications from this documented risk.

Espresso: It seems like flock is needed by geth - thats the reason why flashbots also added it PR 21.

Cantina Managed: Acknowledged.

3.1.3 Informational

3.1.3.1 Unnecessary or restrictive manifest parameters may be refactored **Severity:** Informational

Context: nitro-espresso.manifest#L4, nitro-espresso.manifest#L5-L6

Description: Given the use of Gramine Shielded Containers (GSC) to execute Batch Poster inside an Intel SGX enclave, the corresponding manifest file `nitro-espresso.manifest` specifies some configuration parameters to override the default values.

However, some of the parameters are unnecessary or restrictive as documented:

1. `sgx.nonpie_binary` is not specified in manifest-syntax.html, which indicates that this is either unsupported or unnecessary.
2. From the gramine docs: "*When this feature is enabled, it is not necessary to specify `sgx.enclave_size` (Gramine will automatically set it to 1TB which should be enough for any application). However if `sgx.enclave_size` is specified, this explicit value will take precedence*." However, the manifest has `sgx.edmm_enable = true` and `sgx.enclave_size = "16G"`, which reduces the enclave size from 1TB to 16GB and is unnecessarily restrictive.

Recommendation: Consider if the above manifest parameters can be changed/removed safely.

Espresso: PR 3.

Cantina Managed: Reviewed that PR 3 resolves the issue as recommended by removing `sgx.nonpie_binary` and `sgx.enclave_size = "16G"`.

3.2 Nitro & Contracts

3.2.1 High Risk

3.2.1.1 Overly flexible configuration could allow TEE attestation bypass **Severity:** High Risk

Context: batch_poster.go#L184-L185

Description: The batch poster's configuration system is too flexible, allowing modification of critical security parameters that could potentially enable malicious actors to bypass TEE attestation guarantees. Most critically, empty values for `LightClientAddress` and `HotShotUrl` completely bypass Espresso validation while still allowing the TEE to produce valid attestation proofs.

Several critical configuration parameters can be modified to point to potentially malicious endpoints or alter security-critical behavior:

1. Network Endpoints:
    - `LightClientAddress`: Can be configured to point to an invalid light client contract, potentially allowing verification of invalid HotShot consensus states.
    - `HotShotUrl`: Can be set to any endpoint, potentially allowing the batch poster to receive manipulated consensus data.
    - Critically, setting either of these to empty strings completely bypasses Espresso validation while still producing valid TEE attestations.
    - `ParentChainConfig.connection`: Allows configuration of arbitrary L1 endpoints.
2. Security Parameters:
    - `UseEscapeHatch`: Can be enabled to bypass HotShot consensus verification.
    - `EspressoSwitchDelayThreshold`: Can be set to artificially low values to trigger escape hatch behavior.

The batch poster currently receives the same data that has been submitted by the transaction streamer and does not critically depend on the results from HotShot consensus. However, the flexibility in configuration could enable several attack vectors if it did:

1. Complete Validation Bypass: Empty values for `LightClientAddress` and `HotShotUrl` disable Espresso validation while maintaining valid TEE attestation.
2. Consensus Bypass: A malicious actor could configure endpoints to receive manipulated consensus data while still generating valid TEE attestations.
3. Verification Bypass: Invalid light client addresses could allow verification of incorrect consensus states.
4. Escape Hatch Abuse: Low threshold values could force the system to bypass consensus verification.
5. Chain Data Manipulation: Arbitrary L1 endpoints could provide manipulated chain data.

Impact: The impact could be severe if the batch poster's TEE attestation is used as a security guarantee while the configuration remains flexible. An attacker could potentially:

1. Generate valid TEE attestations for malicious batches.
2. Bypass consensus verification while maintaining apparent validity.
3. Manipulate the sequencing process through controlled endpoints.

Likelihood: The likelihood (as measured in the ease of exploitation) is medium to high for the batch poster if the TEE attestation is used as a standalone security guarantee, as the flexible configuration provides multiple vectors for potential exploitation.

Recommendation: Add stricter configuration validation and ensure that critical endpoint parameters remain unchanged. The entry point script could be modified to receive a default configuration and to only allow a smaller subset of parameters to be modified.

Espresso: Worked on it in PR 3.

Cantina Managed: Fixed. The new GSC manifest file adds the poster_config.json as a trusted file. The config file is hardcoded and passed in when executing the nitro binary in the docker image's entrypoint. There currently does not seem to be a way to run the GSC image while bypassing the entrypoint script or executing the nitro binary with different arguments, as long as the `--insecure-args` option is not used when building the image.

From the the docs:

> Using Gramine's trusted command line arguments Most executables aren't designed to run with attacker-controlled arguments. Allowing an attacker to control executable arguments can break the security of the resulting enclave. gsc build uses the existing Docker image's entrypoint and cmd fields to identify the trusted arguments. These arguments are stored in trusted_argv. This file is only generated when --insecure-args is not specified. As a result any arguments

specified during docker run are ignored. To be able to provide arguments at runtime, the image build has to enable this via the option --insecure-args.

3.2.2 Medium Risk

3.2.2.1 Pending transaction positions not properly tracked when messages are skipped Severity: Medium Risk

Context: transaction_streamer.go#L1602-L1671

Description: In the `submitEspressoTransactions()` function, when messages are skipped (due to `msg.Message` being nil), the code incorrectly updates the pending transaction positions by using the message count from the payload builder rather than tracking which positions were actually processed. This can lead to messages being lost or incorrectly marked as processed.

In the `submitEspressoTransactions()` function, there is a mismatch between how nil messages are handled during message collection and how the pending positions are updated. When messages are nil (`msg.Message == nil`), they are skipped in the message collection loop but the position tracking still assumes all messages were processed.

The issue occurs in these steps:

1. The function collects messages but skips nil ones.
2. After submission, it slices `pendingTxnsPos[:msgCnt]` to track completed messages.
3. This assumes all positions up to `msgCnt` had valid messages, which may not be `true` if some were `nil`.

This can lead to incorrectly marking skipped nil messages as submitted when they weren't actually included in the payload.

Impact: The impact is medium. While this doesn't lead to direct security vulnerabilities, it could cause messages to be lost from the system's tracking, potentially affecting the reliability and correctness of the message processing system.

Likelihood: The likelihood is medium. This issue would only manifest when messages are skipped (`msg.Message` is nil), which may be a relatively rare occurrence. However, when it does occur, the issue will consistently cause tracking problems.

Proof of Concept:

```
// pendingTxnsPos = [0, 1, 2, 3]
// msg at position 1 is nil
// msgs will contain messages from positions [0, 2, 3]
// msgCnt = 3
// Code will incorrectly mark positions [0, 1, 2] as submitted
// Position 1 was never actually processed but is marked as submitted
```

Recommendation: Track which positions actually had messages included in the submission:

```
  func (s *TransactionStreamer) submitEspressoTransactions(ctx context.Context) time.Duration {
      pendingTxnsPos, err := s.getEspressoPendingTxnsPos()
      if err != nil {
          return s.espressoTxnsPollingInterval
      }

      if len(pendingTxnsPos) > 0 {
          msgs := []arbostypes.L1IncomingMessage{}
+         var includedPos []arbutil.MessageIndex
          for _, pos := range pendingTxnsPos {
              msg, err := s.GetMessage(pos)
              if err != nil {
                  log.Error("failed to get espresso submitted pos", "err", err)
                  return s.espressoTxnsPollingInterval
              }
              if msg.Message != nil {
                  msgs = append(msgs, *msg.Message)
+                 includedPos = append(includedPos, pos)
              }
          }
          payload, msgCnt := buildHotShotPayload(&msgs)
```

```
          if msgCnt == 0 {
              log.Error("failed to build the hotshot transaction: a large message has exceeded the size limit")
              return s.espressoTxnsPollingInterval
          }

          // ... submit transaction ...

          s.espressoTxnsStateInsertionMutex.Lock()
          defer s.espressoTxnsStateInsertionMutex.Unlock()

          batch := s.db.NewBatch()
-         submittedPos := pendingTxnsPos[:msgCnt]
+         submittedPos := includedPos[:msgCnt]
          err = s.setEspressoSubmittedPos(batch, submittedPos)
          if err != nil {
              log.Error("failed to set the submitted txn pos", "err", err)
              return s.espressoTxnsPollingInterval
          }
-         pendingTxnsPos = pendingTxnsPos[msgCnt:]
+         pendingTxnsPos = includedPos[msgCnt:]
          err = s.setEspressoPendingTxnsPos(batch, pendingTxnsPos)
          // ... rest of function ...
```

This ensures that only positions of actually included messages are marked as submitted.

Espresso: We found out some potential bugs and we decided to post all the messages (not the user transactions anymore) to hotshot and add a signature into the payload.

This our approach now:

```
fetcher := func(pos arbutil.MessageIndex) ([]byte, error) {
    msg, err := s.GetMessage(pos)
    if err != nil {
        return nil, err
    }
    b, err := rlp.EncodeToBytes(msg)
    if err != nil {
        return nil, err
    }
    return b, nil
}

payload, msgCnt := buildRawHotShotPayload(pendingTxnsPos, fetcher, s.espressoMaxTransactionSize)
if msgCnt == 0 {
    log.Error("failed to build the hotshot transaction: a large message has exceeded the size limit or failed
    ↪  to get a message from storage", "size", s.espressoMaxTransactionSize)
    return s.espressoTxnsPollingInterval
}

payload, err = signHotShotPayload(payload, s.getAttestationQuote)
if err != nil {
    log.Error("failed to sign the hotshot payload", "err", err)
    return s.espressoTxnsPollingInterval
}
```

The `buildRawHotShotPayload` function is like this:

```
func buildRawHotShotPayload(
    msgPositions []arbutil.MessageIndex,
    msgFetcher func(arbutil.MessageIndex) ([]byte, error),
    maxSize uint64,
) ([]byte, int) {

    payload := []byte{}
    msgCnt := 0

    for _, p := range msgPositions {
        msgBytes, err := msgFetcher(p)
        if err != nil {
            log.Warn("failed to fetch the message", "pos", p)
            break
        }

        sizeBuf := make([]byte, LEN_SIZE)
        positionBuf := make([]byte, INDEX_SIZE)

        if len(payload)+len(sizeBuf)+len(msgBytes)+len(positionBuf)+MAX_ATTESTATION_QUOTE_SIZE > int(maxSize) {
            break
        }
        binary.BigEndian.PutUint64(sizeBuf, uint64(len(msgBytes)))
        binary.BigEndian.PutUint64(positionBuf, uint64(p))

        // Add the submitted txn position and the size of the message along with the message
        payload = append(payload, positionBuf...)
        payload = append(payload, sizeBuf...)
        payload = append(payload, msgBytes...)
        msgCnt += 1
    }
    return payload, msgCnt
}
```

So we don't skip `nil` messages now. I think this can resolve this issue. This was implemented in PR 385.

Cantina Managed: Fixed.

3.2.2.2 Missing validation of transactions from the Espresso network **Severity:** Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The current integration with the Espresso network lacks proper transaction validation in the EspressoFinalityNode, allowing potential malicious transactions to be processed until they are eventually rejected by the L1 sequencing contract. While this does not compromise the final ordering or validity of transactions (due to TEE verification on L1), it creates unnecessary overhead for nodes processing these unvalidated transactions.

The `EspressoFinalityNode.createBlock()` function processes transactions from the Espresso network without validating their authenticity or origin. The only validation performed is checking if transactions can be properly deserialized:

```
for _, tx := range arbTxns.Transactions {
    var out types.Transaction
    // signature from the data poster is the first 65 bytes of a transaction
    tx = tx[65:]
    if err := out.UnmarshalBinary(tx); err != nil {
        log.Warn("malformed tx found")
        continue
    }
    txes = append(txes, &out)
}
```

This means that any actor can submit transactions to the Espresso network, which will be processed by nodes until they are ultimately rejected at the L1 sequencing contract level where TEE proofs are properly verified.

Impact: The impact is medium. While the final sequencing security is maintained through L1 TEE verification, the lack of transaction validation means:

1. Node operators cannot distinguish between valid and invalid transactions before L1 confirmation.

2. Nodes waste resources processing potentially invalid transactions.
3. Network bandwidth is consumed by transactions that will eventually be rejected.

Likelihood: The likelihood is medium. The current design makes it straightforward for anyone to submit transactions to the network. However, an actual attack requires preparation and various conditions.

Recommendation: Consider implementing transaction validation that includes TEE verification proofs at the Espresso network level. This would allow nodes to reject invalid transactions early, reducing unnecessary resource consumption.

The current integration should be reevaluated to determine if the added complexity provides sufficient benefits given that the critical security properties are ultimately enforced through L1 TEE verification.

Espresso: Due to a vulnerability that was discovered internally we decided we will exclude espresso finality node from the audit scope because the implementation is incomplete, in PR 385.

Cantina Managed: The payload now contains the signature/attestation. The `EspressoFinalityNode` is missing functionality to verify the signature, however, as it requires further modifications it was moved out of scope for this review.

3.2.3 Low Risk

3.2.3.1 Automata's `V3QuoteVerifier` which is a critical dependency of `EspressoTEEVerifier` has not undergone a security review **Severity:** Low Risk

Context: EspressoTEEVerifier.sol#L6

Description: Automata's `V3QuoteVerifier`, which is a critical dependency of `EspressoTEEVerifier` for verifying TEE quotes, has not undergone a security review and therefore may potentially contain vulnerabilities that affect Espresso's dependency on it.

`EspressoTEEVerifier` uses Automata's library function `quoteVerifier.verifyQuote()` to verify a quote from Batch Poster running inside the TEE and attest on-chain. Automata's `V3QuoteVerifier` was assumed to have undergone a security review which is not the case as confirmed by communication from the Automata team.

Impact: Undetermined, because although Automata's unaudited `V3QuoteVerifier` may contain vulnerabilities that affect Espresso's dependency on it, we cannot determine the number/nature of such potential vulnerabilities. We are considering this as "Very Low" impact for now assuming the best-case scenario of no serious vulnerabilities in Automata's `V3QuoteVerifier`.

Likelihood: High, because Automata's `V3QuoteVerifier` has been integrated for planned production use by Espresso by assuming that it has undergone a security review.

Recommendation: Consider getting a security review of Automata's `V3QuoteVerifier` before using it in production.

Espresso: Acknowledged.

Cantina Managed: Espresso noted this issue and plans to wait for Automata's ongoing security review of their `V3QuoteVerifier` to finish before using it in production.

3.2.3.2 `EspressoTEEVerifier` owner can reset `mrEnclave` or `mrSigner` to unexpected values and break chain liveness **Severity:** Low Risk

Context: EspressoTEEVerifier.sol#L135-L147

Description: Setting `mrEnclave` or `mrSigner` to unexpected values in `EspressoTEEVerifier.setMrEnclave()` and `EspressoTEEVerifier.setMrSigner()` will cause verification to revert and break chain liveness thereafter.

`EspressoTEEVerifier.setMrEnclave()` and `EspressoTEEVerifier.setMrSigner()` allow the contract `owner` to reset values of `mrEnclave` or `mrSigner` to different ones from those at the time of deployment. This is presumably to allow `EspressoTEEVerifier` to work after either the:

1. TEE hosting the Batch Poster has been changed for some reason, which changes the Enclave Identity captured in `MRENCLAVE` or ...

2. The Sealing Identity used for enclave data protection has been changed for some reason, which changes the value captured in `MRSIGNER`.

Changing these values to unexpected values in `EspressoTEEVerifier` such that they are different from the ones in the TEE hosting the Batch Poster will cause verification to revert and break chain liveness thereafter. This is because of the following check in `EspressoTEEVerifier.verify()`:

```
// Check that mrEnclave and mrSigner match
if (localReport.mrEnclave != mrEnclave || localReport.mrSigner != mrSigner) {
    revert InvalidMREnclaveOrSigner();
}
```

Impact: Medium, because the liveness is broken but not integrity. The mismatch in values can be corrected either in the TEE or in `EspressoTEEVerifier`.

Likelihood: Low, because this requires the contract owner to accidentally reset their values to incorrect ones or assumes owner compromise.

Recommendation: Consider:

1. Emitting events in `EspressoTEEVerifier.setMrEnclave()` and `EspressoTEEVerifier.setMrSigner()` for transparency and monitoring.
2. Using a reasonable multisig for the owner address.
3. Using Solady's two-step ownership transfer instead of `transferOwnership()` to prevent accidental transfers to incorrect new owners.
4. Putting this function behind a reasonable timelock.

Espresso: We will work on 1, 2, 3.

For 4, we will skip because Arbitrum currently doesnt call their owner only functions behind a timelock, see SequencerInbox.sol#L847-L859.

So we will keep the same pattern.

Addressed PR 37.

Cantina Managed: Reviewed that PR 37 addresses Recommendation (1) by adding appropriate events and (3) by replacing `solady/auth/Ownable.sol` with `openzeppelin/contracts/access/Ownable2Step.sol`. Acknowledge (4).

3.2.3.3 Malicious rollup owner can arbitrarily update `espressoTEEVerifier` to break chain integrity or liveness **Severity:** Low Risk

Context: SequencerInbox.sol#L861-L864

Description: Malicious rollup owner can arbitrarily reset `espressoTEEVerifier` in `SequencerInbox.setEspressoTEEVerifier()` to break chain integrity by allowing invalid batch poster data with an always passing `espressoTEEVerifier.verify()` or liveness by an always reverting `espressoTEEVerifier.verify()`.

`SequencerInbox` initializes `espressoTEEVerifier` but allows updating it via `setEspressoTEEVerifier()` callable by `onlyRollupOwner`. The rollup owner can maliciously/accidentally update it to a contract that allows invalid batch poster data with an always passing `espressoTEEVerifier.verify()` to break protocol integrity or always reverting `espressoTEEVerifier.verify()` to break protocol liveness.

Impact: High, because this can be used to break either protocol integrity or liveness.

Likelihood: Very low, because this requires the rollup owner to maliciously/accidentally update `espressoTEEVerifier` and the assumption is that the rollup owner account has the highest level of operational security given the potential impact.

Recommendation: Consider:

1. Emitting an indexed event in `SequencerInbox.setEspressoTEEVerifier()` which includes the `_espressoTEEVerifier` address for transparency and monitoring, instead of emitting `OwnerFunctionCalled(6)`.
2. Using a reasonable multisig for the rollup owner address.

3. Using two-step ownership transfer to prevent accidental transfers to incorrect new owners.
4. Putting this function behind a reasonable timelock.

Espresso:

1. We are following the pattern of how Arbitrum sets their batch poster and other addresses context - SequencerInbox.sol#L856.
2. Thanks, Caldera the RAAS provider will handle this.
3. We are following the patter of how Arbitrum does it currently (SequencerInbox.sol#L856). They dont have timelocks and two-step ownership pattern in their code and we will mimic the behaviour. Just to make sure we dont break any functionality
4. Same as point 3.

Addressed in PR 37.

Cantina Managed: Reviewed that PR 37 addresses Recommendation (3) by replacing `solady/auth/Ownable.sol` with `openzeppelin/contracts/access/Ownable2Step.sol`. Acknowledge the rest.

3.2.3.4 Missing mutex protection for pending transaction position updates **Severity:** Low Risk

Context: transaction_streamer.go#L1588-L1594

Description: The `SubmitEspressoTransactionPos()` function modifies shared state (pending transaction positions) without acquiring the `espressoTxnsStateInsertionMutex`. This creates a potential data race condition when multiple goroutines attempt to submit transaction positions simultaneously.

By contrast, the `submitEspressoTransactions()` function correctly protects its shared state modifications with the mutex:

```
s.espressoTxnsStateInsertionMutex.Lock()
defer s.espressoTxnsStateInsertionMutex.Unlock()
```

Recommendation: Add mutex protection in `SubmitEspressoTransactionPos`:

```
  func (s *TransactionStreamer) SubmitEspressoTransactionPos(pos arbutil.MessageIndex, batch ethdb.Batch) error
  ↪ {
+     s.espressoTxnsStateInsertionMutex.Lock()
+     defer s.espressoTxnsStateInsertionMutex.Unlock()

      pendingTxnsPos, err := s.getEspressoPendingTxnsPos()
      if err != nil {
          return err
      }
      if pendingTxnsPos == nil {
          pendingTxnsPos = []arbutil.MessageIndex{pos}
      } else {
          pendingTxnsPos = append(pendingTxnsPos, pos)
      }
      err = s.setEspressoPendingTxnsPos(batch, pendingTxnsPos)
      if err != nil {
          log.Error("failed to set the pending txns", "err", err)
          return err
      }
      err = batch.Write()
```

Espresso: Addressed in PR 407.

Cantina Managed: Fixed.

3.2.3.5 Payload size limit check happens after message addition **Severity:** Low Risk

Context: transaction_streamer.go#L1836-L1854

Description: The `buildHotShotPayload()` function incorrectly checks the payload size limit only after appending each message, rather than verifying if adding the next message would exceed the limit. This means the function can create payloads larger than `ESPRESSO_TRANSACTION_SIZE_LIMIT` (10KiB), since it only breaks the loop after already exceeding the limit.

There are several important size limits in the system:

- Espresso network maximum block/transaction size: 1MB.
- L2Message size limit: 256KiB (`MaxL2MessageSize = 256 * 1024`).
- Default batch poster config `MaxSize`: 100KB (`MaxSize = 100000`).
- HotShot payload size limit: 10KiB (`ESPRESSO_TRANSACTION_SIZE_LIMIT = 10 * 1024`).

With these limits, a single L2 message could be up to 256KiB, while the function's intended limit is only 10KiB. The current implementation could allow a payload to grow beyond the intended 10KiB limit, potentially approaching the L2 message size limit.

For each message, the function:

1. Checks if current payload size exceeds limit.
2. If not, adds 8 bytes for size and the message bytes.
3. Increments message count.

However, it should check if adding the next message would exceed the limit before including it.

Additionally, due to the interaction between size limits:

1. The batch poster's `addSegment` function uses `testForOverflow` to reject messages that would exceed `MaxSize`.
2. A large message (up to 256KiB) that doesn't fit in the current batch will block processing of subsequent messages.

Recommendation: First, consider whether a transaction size limit is needed at all:

1. Every L2 message must be processed for the system to make progress.
2. The L2 message size limit (256KiB) is already enforced elsewhere.
3. The Espresso network's block size limit (1MB) is larger than the L2 message limit.
4. Introducing an artificial limit smaller than these system constraints could potentially cause unnecessary processing delays or system halts.

Alternatively, if the limit must be maintained, fix the size checking:

```
  func buildHotShotPayload(msgs *[]arbostypes.L1IncomingMessage) (espressoTypes.Bytes, int) {
      payload := []byte{}
      msgCnt := 0
      sizeBuf := make([]byte, 8)

      for _, msg := range *msgs {
-         if len(payload) >= ESPRESSO_TRANSACTION_SIZE_LIMIT {
+         nextSize := len(payload) + 8 + len(msg.L2msg)  // 8 bytes for size + message bytes
+         if nextSize > ESPRESSO_TRANSACTION_SIZE_LIMIT {
              break
          }
          msgByte := msg.L2msg
          binary.BigEndian.PutUint64(sizeBuf, uint64(len(msgByte)))
          payload = append(payload, sizeBuf...)
          payload = append(payload, msgByte...)
          msgCnt += 1
      }
      return payload, msgCnt
  }
```

Espresso: Fixed in PR 385.

Cantina Managed: Fixed.

3.2.3.6 Skip verification positions mechanism is redundant and logically inconsistent Severity: Low Risk

Context: batch_poster.go#L565-L617

Description: The skip verification position mechanism in the `checkEspressoValidation` function has logical inconsistencies:

1. During network downtime:
    - The escape hatch is activated (`HotshotDown = true`).
    - Skip verification positions are set.
    - However, messages are already skipped by the `HotshotDown` check, making skip positions redundant.
2. After network recovery:
    - The system continues to skip messages based on skip positions when the network is back up.
    - This is incorrect because these messages should now be processed through the recovered Espresso network.
    - Skip positions from the downtime period should not affect validation after network recovery.

Recommendation: Remove the skip verification position mechanism and rely solely on the `HotshotDown` flag:

```
 func (b *BatchPoster) checkEspressoValidation(
     msg *arbostypes.MessageWithMetadata,
 ) error {
     if b.streamer.espressoClient == nil && b.streamer.lightClientReader == nil {
         return nil
     }
     if msg.Message.Header.Kind != arbostypes.L1MessageType_L2Message {
         return nil
     }
     kind := msg.Message.L2msg[0]
     if kind != arbos.L2MessageKind_Batch && kind != arbos.L2MessageKind_SignedTx {
         return nil
     }

     lastConfirmed, err := b.streamer.getLastConfirmedPos()
     if err != nil {
         log.Error("failed call to get last confirmed pos", "err", err)
         return err
     }

     if lastConfirmed != nil && b.building.msgCount <= *lastConfirmed {
         return nil
     }

-    if b.streamer.UseEscapeHatch {
-        skip, err := b.streamer.getSkipVerificationPos()
-        if err != nil {
-            log.Error("failed call to get skip verification pos", "err", err)
-            return err
-        }
-
-        if skip != nil {
-            if b.building.msgCount <= *skip {
-                log.Warn("skipped espresso verification due to hotshot failure", "pos", b.building.msgCount)
-                return nil
-            }
-        }
-    }

     if b.streamer.HotshotDown && b.streamer.UseEscapeHatch {
         log.Warn("skipped espresso verification due to hotshot failure", "pos", b.building.msgCount)
         return nil
     }

     return fmt.Errorf("%w (height: %d)", EspressoFetchMerkleRootErr, b.building.msgCount)
 }
```

This simplifies the logic:

- During downtime: Skip validation based on `HotshotDown` flag.
- After recovery: Process all messages through the Espresso network as intended.

Espresso: Fixed in PR 407.

Cantina Managed: Fixed.

3.2.3.7 Multiple unsafe unwrap operations could cause node panics **Severity:** Low Risk

Context: lib.rs#L26-L115

Description: The verification helper functions contain numerous `.unwrap()` calls on operations that could fail, potentially causing node panics. In particular:

1. String parsing operations:

```
let block_comm_str = std::str::from_utf8(block_comm_bytes).unwrap();
let txn_comm_str = std::str::from_utf8(tx_comm_bytes).unwrap();
```

2. Deserialization operations:

```
let proof: Proof = serde_json::from_slice(proof_bytes).unwrap();
let header: Header = serde_json::from_slice(header_bytes).unwrap();
let tagged = TaggedBase64::parse(&commit_str).unwrap();
```

   These could panic if:

   - Input isn't valid UTF-8.
   - `TaggedBase64` parsing fails.
   - JSON deserialization fails.

3. Type conversions:

```
let namespace: u32 = namespace.try_into().unwrap();
let commit: VidCommitment = tagged.try_into().unwrap();
```

   Could panic if the u64 value doesn't fit in u32

4. Merkle proof operations:

```
let proved_comm = proof.elem().unwrap().clone();
BlockMerkleTree::verify(block_comm.digest(), header.height(), proof)
    .unwrap()
    .unwrap();
```

These unwrap operations can cause the node to panic if any operation fails, rather than gracefully handling the error by returning `false`.

Recommendation: Replace unwrap operations with error handling that defaults to returning `false` rather than panicking.

Espresso: Addressed in PR 407.

Cantina Managed: Fixed.

3.2.3.8 Unsafe memory operations could cause undefined behavior **Severity:** Low Risk

Context: lib.rs#L78-L115

Description: The code contains two unsafe memory operations that make dangerous assumptions about memory layout and alignment:

1. `std::mem::transmute` is used to convert raw bytes into an `NsTable`:

```
let ns_table: NsTable = unsafe { std::mem::transmute(ns_table_bytes.to_vec()) };
```

This is unsafe because:

- The struct does not have `#[repr(C)]` to guarantee memory layout.
- No alignment checks are performed.
- The operation assumes the byte layout matches the struct exactly.

2. `std::slice::from_raw_parts` is used to create slices from raw pointers:

```
let ns_table_bytes = unsafe { std::slice::from_raw_parts(ns_table_ptr, ns_table_len) };
```

   This assumes proper pointer alignment and valid memory ranges.

Recommendation:

1. Replace transmute with proper deserialization:

```
- let ns_table: NsTable = unsafe { std::mem::transmute(ns_table_bytes.to_vec()) };
+ let ns_table = NsTable {
+     bytes: ns_table_bytes.to_vec(),
+ };
```

2. Add safety checks around raw pointer operations:

```
fn slice_from_raw_parts<'a>(ptr: *const u8, len: usize) -> Option<&'a [u8]> {
    if ptr.is_null() {
        return None;
    }
    if !ptr.is_aligned() {
        return None;
    }
    // Check if the range overflows
    if usize::MAX - (ptr as usize) < len {
        return None;
    }
    Some(unsafe { std::slice::from_raw_parts(ptr, len) })
}
```

3. Consider adding proper memory layout guarantees to `NsTable`:

```
- [repr(C)]
- [derive(Clone, Debug, Deserialize, Eq, Hash, PartialEq, Serialize)]
pub struct NsTable {
    #[serde(with = "base64_bytes")]
    pub(crate) bytes: Vec<u8>,
}
```

Espresso: Previously, constructing an `NsTable` from bytes was not feasible because the `bytes` field was private, forcing us to rely on an unsafe approach.

Although `espresso-types` has since introduced functionality to address this, their recent updates include several untested changes, such as a new header version. Moreover, the new method is not yet available on the Espresso mainnet. Given these factors, we believe it is safer to continue using the current implementation.

Addressed in PR 407.

Cantina Managed: Fixed.

3.2.3.9 Ineffective address validation check **Severity:** Low Risk

Context: deploy.go#L102-L105

Description: The code contains an ineffective address validation check:

```
esperssoTEEVerifierAddress := common.HexToAddress(*espressoTEEVerifierAddressString)
if !common.IsHexAddress(esperssoTEEVerifierAddress.String()) {
    panic("specified espressoTEEVerifier address is invalid")
}
```

This check is problematic because `HexToAddress()` always returns a valid address by zero-padding or truncating input. Therefore, `IsHexAddress(address.String())` should always return true.

Recommendation: Validate the original input string before conversion:

```
- esperssoTEEVerifierAddress := common.HexToAddress(*espressoTEEVerifierAddressString)
- if !common.IsHexAddress(esperssoTEEVerifierAddress.String()) {
+ if !common.IsHexAddress(*espressoTEEVerifierAddressString) {
    panic("specified espressoTEEVerifier address is invalid")
  }
+ esperssoTEEVerifierAddress := common.HexToAddress(*espressoTEEVerifierAddressString)
```

Espresso: Addressed in PR 407.

Cantina Managed: Fixed.

3.2.3.10 HotShot payload lacks message position information for recovery **Severity:** Low Risk

Context: transaction_streamer.go#L1838-L1854

Description: The current implementation of `buildHotShotPayload` only includes message content without position information in the payload. When a node crashes and restarts, or when other nodes try to sync from HotShot consensus data, they cannot determine the original message positions.

The current payload format is:

```
[length_1][message_1][length_2][message_2]...
```

Recommendation: Modify the function to include position information in the payload.

The new payload format becomes:

```
[position_1][length_1][message_1][position_2][length_2][message_2]...
```

This change ensures nodes can properly reconstruct message ordering during recovery or sync operations.

Espresso: Fixed in PR 385.

Cantina Managed: Fixed.

3.2.3.11 Unclear units and behavior for Espresso switch delay threshold **Severity:** Low Risk

Context: batch_poster.go#L189-L190

Description: The `EspressoSwitchDelayThreshold` parameter is unclear in several issues:

1. The unit is block numbers but this isn't documented, unlike other time-based parameters that clearly use `time.Duration`.
2. The parameter name doesn't indicate it represents block lag.
3. It's not clear that exceeding the threshold triggers the escape hatch.
4. The default value of 350 blocks has no clear justification.

```
// Current definition
EspressoSwitchDelayThreshold uint64 `koanf:"espresso-switch-delay-threshold"`

// Used in
func (l *LightClientReader) IsHotShotLive(delayThreshold uint64) (bool, error)
```

The `IsHotShotLive` function returns false when current block number - latest state update block number > threshold, but this behavior isn't obvious from the parameter name.

Recommendation:

1. Rename the parameter to clarify units and purpose:

```
- EspressoSwitchDelayThreshold uint64 `koanf:"espresso-switch-delay-threshold"`
+ MaxBlockLagBeforeEscapeHatch uint64 `koanf:"max-block-lag-before-escape-hatch"`
```

2. Add documentation explaining the behavior:

```
// MaxBlockLagBeforeEscapeHatch specifies the maximum number of L1 blocks that HotShot
// state updates can lag behind before triggering the escape hatch. If the difference
// between the current L1 block number and the latest state update's block number
// exceeds this value, the escape hatch will be activated.
// Default: 350 blocks (~1 hour at 12s block time)
MaxBlockLagBeforeEscapeHatch uint64 `koanf:"max-block-lag-before-escape-hatch"`
```

3. Consider using a more descriptive function name:

```
- func (l *LightClientReader) IsHotShotLive(delayThreshold uint64) (bool, error)
+ func (l *LightClientReader) IsHotShotLiveWithinBlockLag(maxBlockLag uint64) (bool, error)
```

This makes it clearer that the parameter represents a block count threshold and that exceeding it indicates an unhealthy state.

Espresso: Fixed in PR 407.

Cantina Managed: Fixed.

3.2.4 Informational

3.2.4.1 Code quality improvements can enhance readability Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There are some code quality aspects that can be improved to enhance readability. These are itemized in the Recommendation section below.

Recommendation: Consider:

1. Removing the duplicate import of `V3QuoteVerifier` in L5 and L18 of `EspressoTEEVerifier.sol`.
2. In `offset = HEADER_LENGTH + ENCLAVE_REPORT_LENGTH` on L67 of `EspressoTEEVerifier.sol`, the `offset` is technically the end of the array slice and should be named as such.
3. In `parseQuoteHeader()`, the computations of `attestationKeyType`, `qeSvn`, `pceSvn` and `qeVendorId` can be skipped to directly compute those values while initializing `Header` as done for `teeType` and other fields.
4. `parseEnclaveReport()` can be imported from Automata's `QuoteVerifierBase.sol` instead of reimplementing in `EspressoTEEVerifier.sol`.

Espresso: Fixed in PR 37.

Cantina Managed: Reviewed that PR 37 addresses Recommendations (1), (2) and (3).

3.2.4.2 Configurable SGX attestation file paths may be unnecessarily flexible Severity: Informational

Context: batch_poster.go#L186-L187

Description: The batch poster allows configuration of the user report data and quote file paths through configuration parameters rather than using hardcoded SGX paths. Intel SGX uses specific files `/dev/attestation/user_report_data` and `/dev/attestation/quote` for generating attestation quotes, which are critical for proving that code is running in a genuine SGX enclave with the expected measurements.

When generating an attestation quote, the process works as follows: First, the application writes a 64-byte user report data value (in this case, a Keccak256 hash of the batch data padded to 64 bytes) to `/dev/attestation/user_report_data`. The SGX enclave then uses the CPU's EREPORT instruction to generate a hardware-signed report containing this user data. Next, the Quoting Enclave converts this into a remotely verifiable quote by signing it with a device-specific EPID attestation key. Finally, this quote is made available by the SGX runtime at `/dev/attestation/quote` for the application to read and submit along with the batch.

While making these paths configurable does not appear to introduce immediate security vulnerabilities, it represents unnecessary flexibility that could be simplified by hardcoding the standard SGX paths:

1. Writing to a path other than `/dev/attestation/user_report_data` will not generate a valid quote since only the real SGX device file triggers quote generation.
2. Even if an attacker reads a valid quote from a path other than `/dev/attestation/quote`, it would not pass on-chain verification since the smart contract verifies that the quote's user report data matches the hash of the submitted batch data.
3. The MRENCLAVE measurement verified on-chain ensures the program binary cannot be modified.

Recommendation: Consider hardcoding the SGX attestation file paths to their standard locations:

```
- UserDataAttestationFile string `koanf:"user-data-attestation-file"`
- QuoteFile               string `koanf:"quote-file"`
+ const UserDataAttestationFile = "/dev/attestation/user_report_data"
+ const QuoteFile = "/dev/attestation/quote"
```

This would simplify the configuration while maintaining the same security properties.

Espresso: Configurable files allow us to run tests in non SGX environment and as a reason we will keep them as is for now.

Cantina Managed: Acknowledged.

3.2.4.3 Escape hatch activation fails when Espresso network is unreachable **Severity:** Informational

Context: *(No context files were provided by the reviewer)*

Description: The escape hatch mechanism, designed to bypass Espresso verification when the network is down, can fail to activate properly when the Espresso network is unreachable. This occurs because the activation process requires a successful call to the Espresso network, potentially creating a deadlock condition where batch processing remains halted.

The issue occurs in the `toggleEscapeHatch` function when the following sequence happens:

1. The light client reader detects Espresso network is not live (`IsHotShotLive` returns false).
2. The function sets `s.HotshotDown = true`.
3. It then attempts to call `FetchTransactionByHash` on the Espresso network.
4. This call fails since the network is down, causing the function to return an error.
5. The caller (`espressoSwitch`) backs off for `retryRate` seconds.
6. On the next attempt, since `s.HotshotDown` is true, `toggleEscapeHatch` returns early without setting the skip verification positions.
7. This creates a deadlock where:
   - Batch processing is halted waiting for Espresso verification.
   - The escape hatch appears activated (`HotshotDown = true`).
   - But no messages can be processed because skip positions were never set.

It must be stated that in the code's current logic, skip positions are ignored entirely when the network is down. Therefore, this is only an issue if they were relevant.

Recommendation: Modify the escape hatch activation to proceed even when the Espresso network is unreachable.

Espresso: Fixed in PR 407.

Cantina Managed: Fixed.

3.2.4.4 Code quality improvements **Severity:** Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation:

1. Typos and Naming Inconsistencies:

```
- func (s *TransactionStreamer) setSkipVerifiactionPos(batch ethdb.KeyValueWriter, pos
↪  *arbutil.MessageIndex) error {
+ func (s *TransactionStreamer) setSkipVerificationPos(batch ethdb.KeyValueWriter, pos
↪  *arbutil.MessageIndex) error {

- var EspressoFetchMerkleRootErr = errors.New("failed to fetch the espresso merkle roof")
+ var EspressoFetchMerkleRootErr = errors.New("failed to fetch the espresso merkle root")

- []byte = []byte("_espressoPendingTxnsPositions")
+ []byte = []byte("_espressoPendingTxnsPos")
```

2. Error Handling Improvements: Missing error handling:

```
- espresso, _ := b.streamer.isEspressoMode()
+ espresso, err := b.streamer.isEspressoMode()
+ if err != nil {
+     return err
+ }
```

   Unhandled errors during message processing:

```
msg, err := b.streamer.GetMessage(p)
if err != nil {
    log.Error("error getting message from streamer", "error", err)
-     break
+     return err
}
```

3. Code Organization: Move constant to appropriate location:

```
+ // At top of file with other constants
+ const ESPRESSO_TRANSACTION_SIZE_LIMIT = 10 * 1024

// Remove from end of file
- const ESPRESSO_TRANSACTION_SIZE_LIMIT int = 10 * 1024
```

4. Function Optimization: Optimize `checkEspressoValidation()` to validate only message-specific state:

```
- func (b *BatchPoster) checkEspressoValidation(msg *arbostypes.MessageWithMetadata) error {
+ func (b *BatchPoster) checkEspressoValidation(msg *arbostypes.MessageWithMetadata) error {
-     // Global state checks
-     lastConfirmed, err := b.streamer.getLastConfirmedPos()
    // Only check message-specific properties
    if msg.Message.Header.Kind != arbostypes.L1MessageType_L2Message {
        return nil
    }
    // ... rest of message validation
}
```

5. Database Operation Improvements: Move batch creation inside function:

```
- err = b.streamer.SubmitEspressoTransactionPos(pos, b.streamer.db.NewBatch())
+ err = b.streamer.SubmitEspressoTransactionPos(pos)

func (s *TransactionStreamer) SubmitEspressoTransactionPos(
-     pos arbutil.MessageIndex, batch ethdb.Batch
+     pos arbutil.MessageIndex
) error {
+     batch := s.db.NewBatch()
    // ... rest of function
}
```

6. Documentation Updates: Update function binding documentation:

```
// oldSequencerBatchPostMethodName uses automatically generated solidity function
// binding with selector 8f111f3c for "addSequencerL2BatchFromOrigin1"
const oldSequencerBatchPostMethodName = "addSequencerL2BatchFromOrigin1"
```

7. Misleading Function Names: Rename `submitEspressoTransactionPos` to `enqueuePendingTransaction` or similar to better reflect its actual behavior of enqueueing transactions rather than submitting

them.

8. Error Reset Logic: Add missing error handler reset:

```
resetAllEphemeralErrs := func() {
    commonEphemeralErrorHandler.Reset()
    exceedMaxMempoolSizeEphemeralErrorHandler.Reset()
    storageRaceEphemeralErrorHandler.Reset()
    normalGasEstimationFailedEphemeralErrorHandler.Reset()
    accumulatorNotFoundEphemeralErrorHandler.Reset()
    espressoEphemeralErrorHandler.Reset()
+     espressoMerkleProofEphemeralErrorHandler.Reset()
}
```

9. Return Value Simplification: Remove unnecessary return value handling between `espressoSwitch` and `submitEspressoTransactions`:

```
func (s *TransactionStreamer) espressoSwitch(ctx context.Context, ignored struct{}) time.Duration {
    // ... other code ...
    if shouldSubmit {
-         return s.submitEspressoTransactions(ctx)
+         s.submitEspressoTransactions(ctx)
+         return s.espressoTxnsPollingInterval
    }
}

- func (s *TransactionStreamer) submitEspressoTransactions(ctx context.Context) time.Duration {
+ func (s *TransactionStreamer) submitEspressoTransactions(ctx context.Context) {
    pendingTxnsPos, err := s.getEspressoPendingTxnsPos()
    if err != nil {
-         return s.espressoTxnsPollingInterval
+         return
    }
    // ... other error checks ...
    if err != nil {
-         return s.espressoTxnsPollingInterval
+         return
    }
-     return s.espressoTxnsPollingInterval
}
```

10. Conditional Batch Creation: Move batch creation inside conditional block:

```
- batch := s.db.NewBatch()
if s.UseEscapeHatch {
+     batch := s.db.NewBatch()
    // ... use batch
+     err = batch.Write()
+     if err != nil {
+         return err
+     }
}
- err = batch.Write()
- if err != nil {
-     return err
- }
```

Espresso: Fixed in PR 407.

Cantina Managed: Fixed.

3.2.4.5 ChainID should be validated against maximum allowed namespace size **Severity:** Informational

Context: transaction_streamer.go#L117-L142

Description: The transaction streamer needs to validate that the chain ID is within Espresso's namespace limits. According to Espresso documentation, namespaces are internally represented as `u32`, meaning the maximum allowed value is `2^32 - 1` (4,294,967,295). Currently, there is no validation of the chain ID when initializing the transaction streamer, which could lead to transaction submission failures if the chain ID exceeds this limit.

Recommendation: Add validation of the chain ID during transaction streamer initialization.

Espresso: Fixed in PR 407.

Cantina Managed: Fixed.

3.2.4.6 Hotshot consensus ordering by namespace is ignored by batch poster **Severity:** Informational

Context: *(No context files were provided by the reviewer)*

Description: Transaction streamer does not respect HotShot consensus ordering within namespaces.

In the transaction streamer's `pollSubmittedTransactionForFinality()` function, when verifying finalized transactions, the batch poster only validates that its own submitted transactions were included in a block through namespace proof verification. The function does not consider or respect the transaction ordering determined by the HotShot consensus within a single namespace/chainID.

The batch poster operates as a single-party broadcaster by:

1. Only retrieving its own previously submitted messages.
2. Rebuilding the payload to verify inclusion.
3. Enforcing its preferred transaction ordering without considering alternative inputs or the consensus-determined order.

While HotShot consensus does enforce ordering between different namespaces, the ordering inside a particular namespace will likely be supported in the future in the path of decentralization of the sequencer.

Recommendation: This is a known design limitation of the current architecture that will be addressed as part of the sequencer decentralization roadmap. No immediate action is required until the decentralized sequencer infrastructure is implemented.

Espresso: Acknowledged. The ordering doesnt matter for the batch poster because it has its own ordering denoted by `seqNum` (batch_poster.go#L1095) which further means it will not matter if the same message is sent twice to the espresso network because it only cares that it is included once. However, the ordering will matter for the client who is listening to hotshot for transactions but again because of `msgPosition` (espresso_utils.go#L18) the client can build its payload based on the `msgPosition` which increments sequentially and it will have to make sure that it includes each msgPosition only once.

Cantina Managed: This issue we're trying to highlight here is that any other ordering than the expected ordering is ignored. I.e. if the HotShot consensus were to produce some other ordering for some reason and include additional transactions with a different ordering that the batch poster didn't submit, then these would be simply ignored, because the batch poster only listens to whether its own submitted hash has been finalized.

Espresso: Yes, the behaviour is that we ignore other transactions because if a hacker submits a transaction to hotshot and batch poster listens to all transactions finalized by Hotshot. The batch poster could potentially include transactions from the hacker as well which the batch poster doesnt want to include.

Let me know if this makes sense!

Cantina Managed: Yes, that does make sense. We think the point we're trying to emphasize is that this approach effectively bypasses any form of consensus or joint decision-making. The HotShot network is not really being used for reaching consensus. Instead, the batch poster operates more like a single-party broadcaster, imposing its own decision without considering any alternative inputs.