arbitrary./execution

MilkomedaM1

MILKOMEDA ROLLUP SECURITY ASSESSMENT

October 14, 2022

Prepared For:
Rob Kornacki
Prepared By:
Chris Masden, Ian Bridges

Changelog:
September 2, 2022 *Initial report delivered*
October 14, 2022 *Final report delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's (AE) security assessment of Milkomeda's Algorand rollup service. The rollup operates as a Layer 2 EVM service that uses Algorand Layer 1. Algorand was chosen as the chain to use due to its immediate transaction finality. Algorand does not support forking so once a transaction is added to the blockchain, it will always exist in the chain; there are no block reorgs. The rollup service consists of five core components: Observer, Batcher, TxQueue, Web Server, and EVM Rollup Node. All of these were included in the audit except for the EVM Rollup Node. The purpose of the Batcher service is to collect transactions and then batch them to Algorand mainnet. This helps reduce the cost to users and allows the use of Algorand as the ledger used to maintain state in the EVM node. The main goal of the Observer is to read data from the Algorand blockchain and relay any applicable transactions to the EVM node. The TxQueue service provides functionality to add, remove, and sort transactions in the Redis database. The Web Server runs a JSON RPC endpoint that allows signed transactions to be submitted via the `eth_sendRawTransaction` method.

Two AE engineers conducted this review over a 3-week period, from August 8, 2022 to August 26, 2022. The audited commit for the protocol contracts was `267be194add56e911bc898556c151b90fa41f3c1` in the `main` branch of the `dcSpark/milkomeda-rollup` repository. This repository was private at the time of the engagement, so hyperlinks may not work for readers without access. The TypeScript files in the `src/` directory were in scope for this audit. The complete list of files is located in Appendix B.

The team performed a detailed, manual review of the codebase with a focus on the Observer, Batcher, TxQueue, and Web Server services. A focus was placed on actions a malicious user could take to either modify an EVM transaction or to prevent a user or group of users from accessing and using the rollup service. In addition to manual review, the team used SONARQUBE and semgrep for automated static analysis.

Our assessment uncovered issues ranging in severity from critical to note (informational). One of the critical findings concerns a lack of authentication on the Redis database used by the rollup service. If there were a public route to the database, the database could be compromised by a malicious actor and could be used to prevent operation of the rollup service. Another critical finding revolved around the `getTransactionToBatch` function. A malicious actor could insert a large transaction with a high gas price, which would then prevent legitimate users from using the rollup service. This is because the large transaction would be prioritized (because of the high gas price) and the large transaction would fill up the size limit of the batch, preventing other transactions from being added to the current batch.

FIX REVIEW UPDATE

FIX REVIEW PROCESS

After receiving fixes for the findings shared with Milkomeda, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the *Update* section for a finding. For findings that the Milkomeda team chose not to address, the team's rationale is included in the update.

Milkomeda has fixed or acknowledged all major issues identified in the engagement. The fix for the C02 issue involved a major refactoring of the code base. The refactor correctly addressed the issue identified in C02. All the modified behaviors of batcher services as a result of the refactor could not be reviewed in depth within the fix review timeframe. The full breakdown of fixes can be found in the Fixes Summary section.

AUDIT OBJECTIVES

AE had the following high-level goals for the engagement:

- Verify replay protections
- Verify the security of the rollup service
- Verify the reliability of the rollup service

OBSERVATIONS

The transaction finality in Algorand simplifies the process and method of an L2 rollup service. This reduces code complexity and reduces the chance of an undiscovered bug. Two of the main external dependencies of the rollup are the SQLite3 and Redis databases. It is essential to be very careful with databases in regards to sanitizing user-controlled inputs and the authentication schema.

SYSTEM OVERVIEW

All of the rollup services described below operate in Docker containers and are controlled and configured via the `docker-compose` command.

BATCHER

The Batcher service continually runs and queries a Redis database for transactions to batch. The Batcher then collects and bundles the transactions and submits them to the Algorand blockchain.

OBSERVER

Once a transaction bundle has been included in a block it is the job of the Observer to parse those transactions from the Algorand blockchain and send them to the EVM node.

TXQUEUE

A service that primarily interfaces with the Redis database to submit or retrieve transactions. This service contains the functionality required for interaction with the Batcher, Observer, Web Server, and the Redis database.

WEB SERVER

A web server that exposes a way for rollup users to submit a signed transaction via the `eth_sendRawTransaction` method. The Web Server is then responsible for interacting with the TxQueue service to add the transaction to the Redis database to later be included in a batch on Algorand.

EVM NODE

The EVM Node was not in scope for the audit. The role of the EVM Node is to provide a mechanism to allow execution of EVM bytecode using the Algorand network.

VULNERABILITY STATISTICS

Severity	Count
Critical	3
High	0
Medium	0
Low	2
Note	8

FIXES SUMMARY

Finding	Severity	Status
C01	Critical	Fixed in pull request #66
C02	Critical	Fixed in pull request #57
C03	Critical	Fixed
L01	Low	Acknowledged
L02	Low	Acknowledged
N01	Note	Acknowledged
N02	Note	Fixed
N03	Note	Acknowledged
N04	Note	Fixed in pull request #68
N05	Note	Fixed in pull request #69
N06	Note	Fixed in pull request #70
N07	Note	Acknowledged
N08	Note	Acknowledged

FINDINGS

CRITICAL SEVERITY

[C01] ROLLUP SERVICE DENIAL-OF-SERVICE (DOS)

The `getTransactionsToBatch` function in `txQueue.tx` is responsible for collecting a set of Ethereum transactions from the Redis database for batching. This function contains a check to ensure that the size of the transactions added to a batch does not go over a configurable limit (`maxBytes`):

```
public async getTransactionsToBatch(
    maxBytes: number,
    deleteUsedTxs: Boolean = false
): Promise<[Buffer[], string[]]> {

...

if (cumulativeSize + best.rawTx.length > maxBytes) {
    break;
}

...
```

When the TxQueue service runs across a transaction that does not fit in the current batch, it stops adding transactions to the batch and treats it as full. The transactions that were not included in the current batch are left to be included in future batches.

An attacker could exploit this behavior to perform a Denial-of-Service attack on the rollup service by preventing the Batcher from posting transactions to the Algorand blockchain. Executing the attack requires creating a transaction whose size is larger than `maxBytes`. A malicious user can create a transaction large enough to fail the `maxBytes` check by populating the `data` field of the transaction with random bytes.

Once the attacker has a transaction of sufficiently large size, they will need to ensure that it is the first transaction selected by TxQueue for the next batch. The transaction queue is built by pulling one transaction for each account in the Redis database. If an account has multiple transactions waiting to be batched, the Redis database will return the transaction with the lowest nonce that comes after the nonce of the last batched transaction.

Once the initial set of batchable transactions is created, TxQueue selects transactions from that set based on the ratio of gas price to size of the transaction (in bytes).

Given how TxQueue selects transactions for batches, the attacker can increase the likelihood of their malicious transaction being selected as the first transaction in a batch by ensuring that:

1. The attacker account has no outstanding transactions waiting to be batched
2. The transaction has a high gas price-to-transaction size ratio

The attacker can guarantee that the first condition is met by sending the transaction from an account that has never sent a transaction to the rollup node. The attacker can meet the second requirement by setting a high gas price for their transaction:

```
const myString = 'A'.repeat(299880); // tx size = 600,002 bytes, maxBytes =
600,000 bytes
const encoded = Buffer.from(myString).toString('hex');
const tx = {
    from: rollupAccounts.validator1,
    to: rollupAccounts.member1,
    value: toWei("1"),
    gasPrice: toWei("600003", "Gwei"), // size of transaction in bytes + 1
    data: encoded,
};
```

When TxQueue attempts to collect the set of transactions for the next batch, it will first check whether the malicious transaction will fit in the current, empty batch. Since the transaction is larger than `maxBytes`, it will fail the size check and exit the `while` loop. When the `while` loop terminates in this way, `getTx` is never called a second time. The second call to `getTx` is responsible for incrementing the `number` stored in `txsInProgress`. TxQueue uses the `number` stored in `txsInProgress` to track the index of the next transaction in the Redis database for batching. Since this value is reset before each batching round, TxQueue will pull the malicious transaction from the database again the next time the Batcher requests a batch of transactions.

After the `while` loop terminates, TxQueue will return an empty set of transactions to the Batcher. Since the batch is empty, the Batcher will not post anything to the Algorand blockchain. Unless another user submits a transaction with a higher gas price-to-transaction size ratio from an account that has no transactions waiting to be batched, the next time the Batcher requests a new set of batchable transactions, it will receive another empty set. This occurs because TxQueue will fail to fit the malicious transaction into the requested batch. Since the Batcher cannot post transactions to Algorand, the Observer will never receive transactions to send to the EVM node.

RECOMMENDATION

Consider adding a check to `validateTransaction` that checks whether the length of the transaction is greater than the maximum batch size. If the transaction is too large to fit into a single batch, reject the transaction and return an error to the user. This will prevent an oversized transaction from being added to the Redis database.

UPDATE

Fixed in pull request #66 (commit hash `d7cce1ff80d073b7c8c79fc671af895364015408`). Milkomeda fixed the issue by inserting a transaction size limit of 50000 bytes.

[C02] BATCHER FUNDS CAN BE DRAINED BY ANYONE

The Batcher service retrieves transactions via the TxQueue service to be batched from the Redis database. The Redis database can be populated by any account sending a signed transaction via the `eth_sendRawTransaction` method. TxQueue validates a transaction for inclusion into a batch of transactions written to the Algorand blockchain by performing various checks on the transaction format in the `validateTransaction` function. There is information in the transaction that `validateTransaction` is not able to verify because it relies on the account's state on the EVM Node. One piece of information that it is not able to verify is the nonce of an account. This means that if a signed transaction with an incorrect nonce is sent to the Batcher, that transaction will be included into a batch and written to the Algorand blockchain.

The Observer service will correctly invalidate the transaction in the `validateEvmTxn` function and will correctly not relay the transaction to the EVM node.

This means that an attacker can insert invalid transactions and have the Batcher include them into transaction batches to the Algorand blockchain, which results in the Batcher paying transaction fees without any penalty to the user who submitted the invalid transactions. The intention is that the fees collected on the rollup service will be more than the fees paid by the Batcher when submitting blocks to the Algorand blockchain. The transaction fees on the EVM node are supposed to pay for any fees incurred by the Batcher, but with an invalid transaction no fees are collected on the node.

This can allow an attacker to perform the following steps and drain the Batcher service of ALGO:

1. Create a signed transaction with an invalid nonce.
2. Send the transaction to the Redis database via the `eth_sendRawTransaction` endpoint.
3. The Batcher will then pull the transaction out of the database and include it into an Algorand transaction batch (costing the Batcher ALGO).
4. Observer picks up the transaction and does not post it to the EVM node because it contains an invalid nonce.
5. Repeat.

RECOMMENDATION

Consider adding functionality to the TxQueue that ensures a submitted transaction has a correct nonce and the account has at least the required amount to pay the gas fee so that the Batcher can recoup transaction fees.

UPDATE

Fixed in pull request #57 (commit hash `99cf31832430b4c3360aee3947d07a586b500b6c`). Milkomeda has performed a significant refactor of the code base and now simulates EVM transactions and removes those that fail validation before including them in a new batch.

[C03] NO AUTHENTICATION ENABLED ON REDIS DATABASE

The Milkomeda protocol stores transactions in a Redis database that the Batcher service uses to batch transactions to Algorand. The database plays a key role in the rollup system and if that database can be manipulated it can prevent users from using the rollup service. There is no authentication in place on the database. The only thing needed to connect and modify database entries is a routable path to the docker container the database runs in. The malicious actor can then remove a specific user's transactions or remove all transactions in the Redis database.

RECOMMENDATION

Consider adding authentication to the Redis database. The Redis documentation provides guidance on how to enable authentication.

UPDATE

Fixed, as recommended. Milkomeda's statement for this issue:

> *"We are ready to have authentication set up on Redis in production deployments. No change in rollup code is needed (redisUrl connection string can include username and password)."*

LOW SEVERITY

[L01] INABILITY TO RELIABLY CANCEL TRANSACTIONS

In the Ethereum network, a user can cancel a pending transaction by sending another transaction with the same nonce, but with a higher gas price to make it more likely to be selected by miners over the previous transaction, which becomes invalid once the replacement transaction has been included in a block. The implementation of the Milkomeda rollup allows users to submit transactions with duplicate nonces, but there is not a way for a user to prioritize a transaction with the same nonce over a different transaction with the same nonce by setting a higher gas price.

The Redis `zRange` function used in `getTx` will sort transactions first by nonce and then lexicographically. This means that given two transactions with the same nonce, the transaction with the lower lexicographical ordering will be returned first by the Redis database.

The following code demonstrates this:

```
...

signedTx.gasPrice = toWei("200", "Gwei");
const response = await postSignedTx(signedTx, 150);

...

// Send a transaction with the same nonce, but higher gas price to attempt to
cancel the previous transaction
signedTx.gasPrice = toWei("250", "Gwei");
const response2 = await postSignedTx(signed2, 151);

...

let firstTx = await txQueue.getTx(rollupAccounts.validator1, false)
// TxQueue picks up the first transaction with a lower gas price instead of
the second one
expect(firstTx).to.equal(signed)
```

The only way to effectively cancel an existing transaction is to send a transaction with the same nonce, but whose raw bytes would be ordered lower by the Redis database than the existing transaction.

RECOMMENDATION

Consider encoding all information needed to select a transaction for batching into the Redis database. Specifically, consider adding a new field for each transaction that tracks whether a transaction has been batched. The query for pulling transactions from the Redis database could then be updated to pull the transaction with the lowest nonce that has not been added to a batch. If multiple transactions are present with the same nonce, TxQueue could select the one with the highest gas price to transaction size ratio.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for this issue:

> *"These are good improvement points, we haven't followed through so far as changing the queue structure requires some care. Longer term we are moving all of txQueue/L2 mempool to Quorum node in multibatcher scenario."*

[L02] IMPROPER HANDLING OF TRANSACTIONS WITH SAME NONCE

The TxQueue service implemented in `txQueue.ts` allows users to submit transactions with nonces that collide with nonces already present in the Redis database.

TxQueue tracks the index for the next transaction to retrieve from the Redis database in the `txInProgress` mapping. This value is incremented each time a transaction is successfully pulled from the queue and placed into a batch. TxQueue assumes that each transaction in the Redis database will have a unique nonce that is one more than the nonce of the previous transaction. Given this assumption, the value in `txInProgress` can be used as an index into the Redis database, which will produce the transaction with the next nonce in the sequence.

However, TxQueue does not enforce this assumption. A user can submit a transaction with a nonce that has already been used in a transaction. If a user submits a transaction with a nonce of zero from an account whose current nonce is greater than zero, that transaction will be put at the front of the list when the Redis database is queried for transactions for that account. The next time `getTx` is called, `txInProgress` will no longer point to the transaction with the next lowest nonce.

If a user were to manually submit a transaction with a nonce lower than the next expected nonce, they could delay their own legitimate transactions from being added to batches.

Although this is possible, the given scenario only happens if the invalid transaction with a lower than expected nonce is submitted within the same batch round as the valid transaction. This is because the `deleteTxsInProgress` function is called after each batch round, which will reset the index counter for an account in the Redis database after each batch round.

RECOMMENDATION

Consider encoding all information needed to select a transaction for batching into the Redis database. Specifically, consider adding a new field for each transaction that tracks whether a transaction has been batched. The query for pulling transactions from the Redis database could then be updated to pull the transaction with the lowest nonce that has not been added to a batch. If multiple transactions are present with the same nonce, TxQueue could select the one with the highest gas price to transaction size ratio.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for this issue:

> *"These are good improvement points, we haven't followed through so far as changing the queue structure requires some care. Longer term we are moving all of txQueue/L2 mempool to Quorum node in multibatcher scenario."*

NOTE SEVERITY

[N01] DISCREPANCY IN BATCH SIZE LIMITS

The Batcher sets the maximum batch size `MAX_BATCH_BYTES` in the `createBatch` function to `1024 * 15 * (2^31)`. However, Algorand's maximum block size is `1000000 bytes`. The documentation in the `config.ts` file states that all transactions in a batch must be confirmed within the round `N + batchPostInterval`, where `N` is the round where the corresponding batch proposal was confirmed and `batchPostInterval` is the number of rounds after `N` when all transactions in the batch must be confirmed. This is enforced in the `getBatch` function The default `batchPostInterval` is set to `6`. This implies that the actual maximum batch size should be `6000000 bytes`, which is far less than the value stored in `MAX_BATCH_BYTES`.

RECOMMENDATION

Consider modifying the batch size check in the `createBatch` function to match the maximum batch size calculated in the `getBatch` function. Updating the check ensures that any processing of transactions in batches between these two function calls does not increase the size of the batch beyond the expected batch size limitation.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for this issue:

> *"It's a bit tricky to make a really good check on batch size. We'd need to consider the non-note portion of Algorand transactions that house the batch contents for example. In any case `expectedAvailableBlockBytes` from config will be set pretty conservatively, as we consider other network users competing for block space."*

[N02] FAULTY ERROR CHECKING IN TEST CODE

The `"end user constructs & signs rollup transactions and submits them to our batcher"` test in `end2end.test.ts` sends 5 transactions and ensures those transactions were posted successfully by verifying that HTTP response status code `200 OK` is returned for each transaction. However, both valid and invalid transactions will be posted with a `200 OK` status code. Invalid transactions will contain an additional error message in the response data. Therefore, checking the HTTP status code alone is not sufficient to determine if a transaction was valid.

RECOMMENDATION

Consider checking the data of the HTTP response when submitting a transaction for any errors.

UPDATE

Fixed. Previously in the audited code, the `index.ts` file would forward HTTP POST requests to the Quorum node. The Quorum web service would then return an HTTP status code of `200 OK` when an invalid transaction was submitted. As of commit hash `3327cd2f78b5316a4b3f53125e1bddd0d623dba4`, the route was changed and now calls the `receiveTransaction` function, which correctly returns an HTTP status code of `400 BAD REQUEST`. This now results in appropriately failing the test.

[N03] HTTP USAGE IN THE GOQUORUM NODE

The GoQuorum node currently uses an HTTP connection that allows users to submit transactions to be batched and used in the EVM Rollup. An HTTP connection is not secure because request and response data is sent as plain text. Data sent through an HTTP connection can be susceptible to a Man in the Middle attack. While an attacker cannot currently manipulate intercepted data to his benefit (the data being sent to the web server is a signed transaction) it is possible in the future that privileged information may be sent that could be manipulated to a users detriment. HTTPS is the secure version of HTTP that encrypts all data being transmitted. The use of HTTPS is strongly recommended to protect data during transmission over public networks.

RECOMMENDATION

Consider adding the Quorum Enterprise Security Plugin to the GoQuorum node to enable connections over HTTPS.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for this issue:

> *"All batcher services will be deployed on the same machine and the communication will be over localhost so MITM attack is not a concern."*

[N04] MAGIC VALUES IN BATCHER.TS

The following code block declares variables using magic numbers that are not easily comprehensible or derived:

```
export const MAX_TXN_NOTE_BYTES = 1024;
export const MAX_CHUNK_BYTES = 15 * MAX_TXN_NOTE_BYTES;
export const HASH_SIZE_BYTES = 32;
export const PROOF_ELEM_BYTES = HASH_SIZE_BYTES + 1; // 1 for side identifier
export const MAX_BATCH_BYTES =
    MAX_CHUNK_BYTES * 2 ** Math.floor(MAX_TXN_NOTE_BYTES / PROOF_ELEM_BYTES);
```

RECOMMENDATION

Consider adding a comment for each variable explaining where the value came from and include hyperlinks when appropriate.

UPDATE

Fixed in pull request #68 (commit hash `1b2bf250f34b7879ea3585bdaceeab4bbb7474aa`), as recommended.

[N05] MISLEADING POSTGRES COMMENTS

The following lines refer to a future Postgres database upgrade:

- Line 90 in the `observer.ts` file
- Line 274 in the `batcher.ts` file
- Line 281 in the `batcher.ts` file

These comments are misleading, as Milkomeda has stated that they do not plan on changing their database type in the Batcher and Observer.

RECOMMENDATION

For clarity, consider removing all references to Postgres.

UPDATE

Fixed in pull request #69 (commit hash `18d67cced6eb797b8e1290378d5b9f2e776bd194`), as recommended.

[N06] TYPOGRAPHICAL ERRORS

The following files and lines contain typographical errors:

- `README.md`
  - Line 81: there is no url for the indicated hyperlink corresponding to `npm run serve-testnet`.
  - Line 163: the hyperlink `#start-redis` does not link to anything. Consider updating the hyperlink to `#redis`.
  - Line 240: `accound` is a misspelling of `account`. Also, the hyperlink corresponding to `this dapp` returns a `404 status`.
- `types.ts`
  - Line 2: `aplication` should be `application`.
- `txQueue.ts`
  - Line 51: `unsuccessfull` should be `unsuccessful`.
  - Line 67: `usecase` should be `use case`.
  - Line 165: `Transation` should be `Transaction`.
  - Line 187: `prioritised` should be `prioritized`. Consider changing `price` to `prices`.
  - Line 190: `succesful` should be `successful` and `retrived` should be `retrieved`.
  - Line 190: `deletedUsuedTxs()` is referenced as a function in the comment, but is referring to `deleteTxsInProgress()`. Consider changing `deleteUsedTxs()` to `deleteTxsInProgress()`.
- `batcher.ts`
  - Line 273: `neccessary` should be `necessary` and `ony` should be `only`.
  - Line 380: `succesful` should be `successful`.
  - Line 381: `succesful` should be `successful`.
  - Line 485: `batchPostLimit` in the comment refers to the `batchPostInternal` variable. Consider changing `batchPostLimit` to`batchPostInterval`.
  - Line 627: `successfuly` should be `successfully`.
- `config.ts`
  - Line 53: `treshold` should be `threshold`.

RECOMMENDATION

Consider making the suggested changes above to fix the typographical errors.

UPDATE

Fixed in pull request #70 (commit hash `7af95dd1ef2a10e6ce1d6455c1110891c7ea9432`), as recommended.

[N07] OVERLY COMPLEX TRANSACTION CHECKS

The `validateTransaction` function performs a series of checks on transactions before they are added to the Redis database. Three of these checks enforce an upper bound on values encoded in the transaction:

```
// sanity check for extremely large numbers
if (parsedTx.gasLimit.toHexString().length > 66) {
    return Error("Gas limit is very high");
}
if (parsedTx.gasPrice.toHexString().length > 66) {
    return Error("Gas price is very high");
}
if (parsedTx.value.toHexString().length > 66) {
    return Error("Transaction value is very high");
```

It is not clear to a reader why the code converts number values into hex strings and then compares the length of those strings to an upper bound. Additionally, the significance of the upper bound (a string length of 66) is undocumented. The number 66 is used as the upper bound because two of the characters will be `"0x"` which denotes it as a hexadecimal value. The remaining length of 64 characters denotes a 32 byte value (2 characters per byte).

RECOMMENDATION

Consider using the `BigNumber` built-in comparison and equivalence functions to perform these checks against the original number values.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for the issue:

> *"We'd need to add extra dependency to import ethersproject's BigNumber and then construct appropriate value to compare against - current method seems simpler."*

[N08] UNNECESSARY WALLET INITIALIZATION

When the Batcher's `kmd` wallet handle expires, it will initialize a new handle. The handle expiration checks are handled in proposeBatch and postBatchChunks. It is not necessary to initialize a new handle because renewal of handles is possible and recommended in the Algorand documentation if the handle expires.

RECOMMENDATION

Consider renewing expired handles instead of creating new ones to replace them.

UPDATE

Acknowledged, and will not fix. Milkomeda's statement for the issue:

> *"We tried to use `renewWalletHandle` in Algorand JS SDK now and earlier, but it seems not working, thus we must keep the wallet reinitialization."*

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	Definition
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
./src/batcher.ts
./src/config.ts
./src/error.ts
./src/index.ts
./src/observer.ts
./src/server/index.ts
./src/server/routes.ts
./src/services.ts
./src/txQueue.ts
```