

ReserveBlock Core August 2022 Audit

Alex Williams

PhD in Mathematics¹

1 Preface

The findings and recommendations herein were conducted during a consultation and audit of the *ReserveBlock-Core* project during August 2022. During this time, the project has been undergoing significant changes and will continue to do so in the near term. The paper highlights implementation details, areas of concern, potential improvements, and various trade-offs relevant to the project.

Contents

1 Preface	1
2 Relevant Experience	2
3 Testing Methodology	2
4 Startup processes	2
4.1 Long wait to access wallet functionality	3
4.2 Potentially large database size	3
4.3 Thread safety	4
5 Databases	5
5.1 Denormalization	5
5.2 Unfiltered queries	5
5.3 Indices on natural keys	6
5.4 Updating via natural keys	6
5.5 The $n + 1$ problem	6
5.6 Block corruption	7
6 Cryptography	7
6.1 Storage of private keys	7
6.2 Post-Quantum attacks	7
7 Client and server	8
7.1 Empty block exploit	8
7.2 Denial of service	8
7.3 Persistent mempool saturation	9
7.4 Centralized network topology	9
7.5 Instant send tradeoff	10

¹acquired from Texas Tech University

8 Metrics	10
8.1 Metrics in the limit	10
9 Masternodes	11
9.1 Validation complexity	11
9.2 Masternode censorship	11
9.3 Submit time moral hazard	12
10 Adjudicators	12
10.1 FortisPool exploit	12
10.2 Trust is required	12
10.3 Skin in the game	13
10.4 Network partition	13
11 Beacons / NFT relays	14
11.1 Storage concerns	14
12 Smart contracts	14
12.1 Turing completeness	14
13 Disclaimer	15

2 Relevant Experience

I have served in several full-stack lead, solo, and architectural software development roles in various contexts including (but not limited to) distributed systems, third-party integration, single-page SAAS applications, custom algorithm development, data engineering, and parallel programming. Over the past 10 years, several of these projects heavily used C# and the .NET ecosystem.

3 Testing Methodology

Performing a comprehensive consultation and audit requires a deep understanding of (a) how the project should work (i.e. the specification), (b) how and where that specification is implemented in the code, (c) data as it layers from the API, the local in-memory cache, the persistent database, and the various operations enabled by the user interface. To achieve this understanding, I had several lengthy discussions regarding the intended behavior of the project with the core developer, studied the code file-by-file and function-by-function, and crafted custom queries against the API and the database.

The primary methodology of testing used was examining custom logging and studying the associated code. Other methods of testing such as unit testing and integration testing were less extensively used due to their limited ability to uncover unknown issues. As an example, transactions appear to send correctly both from a unit testing and integration testing perspective. However, when studying the code in the broader context of the API control flow, we see scalability issues that need to be addressed. A key takeaway is that while we can shed light on discovered issues, we cannot affirmatively conclude any aspect of the project is devoid of potentially relevant issues.

4 Startup processes

The ReserveBlock code is compatible with all platforms that support *.NET 6.0*. When the code is running for the first time, *LiteDB* databases are automatically installed assuming there are no permission or storage issues. Any time a wallet is started, the following are initiated: a command loop, task schedulers, database validations, a connection to the lead adjudicator, connections to peers, block syncing, the wallet API, and

the *SignalR* server hub(s).

The command loop accepts commands typed by a user. The `/help` command displays a full list of available commands. The scheduled tasks perform database cleanup, get the max block height from all connected peers, and attempt to maintain the connection to peers and the lead adjudicator.

The wallet accepts the following command line options:

1. `testnet` - This will connect the application to the testnet.
2. `enableapi` - This will enable the GUI interface and curl calls to the wallet.
3. `testurl` - This cause the wallet API to port 7777.
4. `privKey` - This will add the account associated with the private key before the wallet loads.

These commands appear to be correctly implemented.

4.1 Long wait to access wallet functionality Wallet functionality such as creating transactions is disabled until all startup processes are finished. As time progresses, the wait time might be a concern.

Observations: The methods `StateTreiSyncService.SyncAccountStateTrei` and `StartupService.CheckForDuplicateBlocks` run at start-up. These are temporary solutions there were put in place to make sure balances are in sync with transactions in a user's block database and there are no duplicate blocks in the database. These checks have been causing the wallet to take noticeably longer to load.

Relevance: The time to load a wallet will continue to increase even if it is synchronized with all existing blocks. Since this effect was noticeable in less than a month, it would be significantly slower over larger timeframes such as a year.

Improvements: The need for the balance sync and block duplication check is believed to be resolved by proper utilization of database transactions and reimplementing block downloading logic.

Suggestions: The severity of these issues suggests their resolution should be verified. To verify users should be provided with an optional sync command. Ideally, the command would output if any balance discrepancies were found with a helpful log that a developer could investigate. A reasonable default would be to automatically run the command until a certain block height has been reached. Also, the sync command should pause the addition of new blocks since updating balances while recalculating them can cause a miscalculation.

4.2 Potentially large database size The block database size could grow over 1 terabyte a year.

Observations: At a block height of 120,000 the block database is a little over 200 MB. The code allows for 1 megabyte per block size.

Relevance: If a small percentage of RBX is staked, the supply of RBX for sale could make validating unprofitable if data storage costs are too high. Also, a large database can be a prohibitive barrier to entry for new validators.

Since most blocks currently only contain a block reward transaction, the current growth rate should be considered a lower bound. At this rate a year of mining will increase the database size by at least 1.75 GB. Since blocks will be allowed to reach 1 MB in size, a worst-case size increase per day is 2.9 GB. Since the

project is still early and the bound estimates have so much variance, it is difficult to estimate the size the blockchain will be.

Improvements: Parallel downloading is now implemented to utilize more peer bandwidth, which in turn speeds up block downloading. Significant changes were made to `BlockDownloadService.GetAllBlocks()` and `BlockValidatorService.ValidateBlocks()` was added to process validation from a new in-memory queue. The queue is expected to resolve the duplicate block problem.

Also, logic was added to opportunistically seek out peers that provide higher bandwidth. The following code implements the opportunistic bandwidth functionality:

```
var PeersWithSamples = Globals.Nodes.Where(x => x.Value.SendingBlockTime > 60000)
    .Select(x => new
    {
        IPAddress = x.Key,
        BandWidth = x.Value.TotalDataSent / ((double)x.Value.SendingBlockTime)
    })
    .OrderBy(x => x.BandWidth)
    .ToArray();

var Length = PeersWithSamples.Length;
if (Length < 3)
    return;

var MedianBandWidth = Length % 2 == 0 ? .5 * (PeersWithSamples[Length / 2 -
1].BandWidth + PeersWithSamples[Length / 2].BandWidth) :
    PeersWithSamples[Length / 2 - 1].BandWidth;

foreach (var peer in PeersWithSamples.Where(x => x.BandWidth < .5 *
    MedianBandWidth))
{
    if(Globals.Nodes.TryRemove(peer.IPAddress, out var node))
        await node.Connection.DisposeAsync();
}
```

Suggestions: It is unclear that any needs to change to increase validating profitability at this time. Download speed is still significantly limited due to the API only allowing to request one block at a time. An aggregate block size table would enable efficient range queries to send several blocks at once.

4.3 Thread safety For convenience and performance, the wallet maintains various data in a global cache during run time. Some of the fields are not implemented in a thread-safe manner.

For convenience and performance, the wallet maintains various data in a global cache during run time. Some of the fields are not thread-safe and some of the fields require other fields to be initialized before they are. In particular, `List` fields such as `TaskAnswerList` and `FortisPool` should be implemented as `ConcurrentBag` instances.

Observations: Some fields (most notably `List<>` fields such as `FortisPool`, `TaskAnswerList`, and `BroadcastedTrxList`) are not implemented with proper locking mechanisms and some of the fields implicitly assume they will load in a certain order without the code providing such a guarantee.

Relevance: Fields that are not thread-safe can cause updates to be ignored and processes to intermittently fail. The bugs associated with these problems are often among the most difficult to detect and troubleshoot.

Improvements: Most static fields were gathered into one file. C# primitives such as `SemaphoreSlim` and `ConcurrentDictionary` were appropriately used for some fields such as `Globals.Memblocks` and `Globals.Nodes`.

Suggestions: A careful and appropriate use of C# primitives such as locks, semaphores, concurrent data structures, and atomic writes could allow the wallet to be thread-safe. Also, the use of a static constructor can guarantee the order that various fields load. As development time permits, all static fields should be changed to use thread-safe primitives.

5 Databases

The information stored in the databases consists of blocks with their transactions, unprocessed transactions (the mempool), a user's public/private key pairs, nonces for public addresses, and IP addresses of wallet nodes. Some of the information is denormalized such as account and wallet balances, block heights, and the most recent block's hash.

The databases are automatically generated from and are compatible with LiteDB. LiteDB will throw an exception if there are simultaneous writes to a table. To alleviate this issue, wrapper code for LiteDB was written to provide safe database modifying functions.

5.1 Denormalization Denormalization consists of both explicit and implicit duplication of data.

Observations: Account and wallet balances are examples of denormalized data in the project. This is clear because they can be computed solely from the information in the block database.

One denormalization bug found was that wallet balances do not properly update if RBX is sent from another wallet with the same `FromAddress`.

Relevance: Denormalization can introduce discrepancies that require proper use of transactions to prevent. Account and wallet balances in particular cannot be trusted if they are not guaranteed to exactly reflect the transactions in the blocks stored in the database.

Improvements: The validation code was modified to properly use transactions to make sure balance updates occur in lockstep with the addition of new blocks.

Suggestions: There are other portions of the code that require the use of transactions to maintain data integrity in cases such as power loss or exceptions being thrown. In particular, code involving unprocessed transactions requires careful attention to make sure they are removed from the mempool precisely when they are successfully crafted into a winning block. LiteDB isolates transactions within the context of managed thread ids. While this provides an intuitive abstraction for developers, future code must not be added that attempts to modify data in the same table from two different threads at the same time to prevent deadlocking.

Denormalization also benefits from maintaining a 'business logic' or repository level of abstraction (as opposed to interleaving raw database queries with non-database code) to maintain invariants and coordinate database operations. This type of organization can make limit the surface area of code that can cause denormalization bugs.

5.2 Unfiltered queries Unfiltered queries consist of pulling all data in a database table into memory instead of asking the database to send a limited number of entries.

Observations: There were database queries that would over time pull a large amount of data into memory. These places could be found in the code by searching for `.FindAll`.

Relevance: Pulling a large amount of data into memory could cause the wallet to become unresponsive and fail to complete operations within the time allocated for them.

Improvements: Now, most queries pull no more rows out of a database than necessary.

Suggestions: A good rule of thumb is to ensure database queries do not request more than 1000 entries at a time unless the query is performed once at start-up to load data into the cache. Whenever it is appropriate to use the `FindOne` command, it should be preferred to `Find` or `FindAll`.

5.3 Indices on natural keys Databases indexes allow for more efficient queries at the expense of a larger database and slower insertions. It is considered best practice to use them for ongoing range and equality-based queries.

Observations: Indices are currently used on blocks and the mempool. However, they are not used for addresses.

Relevance: Not utilizing indices when they are appropriate can significantly harm query performance as tables grow in size.

Improvements: None.

Suggestions: The account database would also benefit from an index on the address column. LiteDB's `EnsureIndex` is present with several query operations. Database calls can be reduced by only calling `EnsureIndex` on wallet start-up.

5.4 Updating via natural keys One antipattern consists of querying a database for an entity and then making a separate query to either insert or update the entity depending on if it was found.

Observations: `GetSeedNodePeers` is one example of this antipattern.

Relevance: This antipattern enables the possibility of duplicate entities.

Improvements: None.

Suggestions: LiteDB exposes the `Upsert` command which inserts data if there is no existing associated entity and otherwise updates the associated entity.

5.5 The $n + 1$ problem The $n + 1$ problem refers to instances where database operations occur in a loop rather than sending a constant number of bulk operations to a database.

Observations: The validation code that updates account balances is a hot path in the code where the $n + 1$ problem is prominent.

Relevance: Communication to external processes such as database manipulations are relatively far slower than interprocess communication. The $n + 1$ problem can significantly limit an application's scalability.

Improvements: None.

Suggestions: LiteDB's `Insert` and `Update` commands have overloads for multiple entities, which could enable a significant reduction of the possible number of database operations while processing blocks, which would alleviate one of several potential transaction bottlenecks.

5.6 Block corruption A node's blocks are corrupted if they differ from the majority of other nodes.

Observations: I wrote code that proved it is feasible to craft blocks with any given previous hash that would be accepted as valid even if it was not considered a winning block (assuming the winning block has not already been added). These crafted blocks could be given to peers that request them for downloading.

Relevance: For each node, it is critical their blocks are the only ones adjudicators considered accepted. Validation is not enough. It is feasible to intentionally corrupt one's peers' blockchains. It is also feasible to unintentionally cause corruption by sharing a corrupted block that has been downloaded.

Improvements: Peer banning logic has been added. However, by itself, this does not fully address either intentional or unintentional block corruption.

Suggestions: User's should be given a command that allows them to revert to a given block height. This functionality might need to be used if an invalid block has been detected since it is possible a previous corrupted block has already been added.

6 Cryptography

The project uses the same elliptic curve cryptography scheme as used by Bitcoin. No faults were found with the correctness of the implementation.

6.1 Storage of private keys It is critical to minimize access to a wallet's private keys.

Observations: The lead developer is working on wallet encryption that mirrors Bitcoin's implementation. By default, 1000 public addresses will be generated so password access will be primarily limited to creating new transactions.

Relevance: If keys are unencrypted, it is easier for someone that has physical access to a wallet device to transfer funds to their personal wallet.

Improvements: None.

Suggestions: Wallets should be encrypted by default. C# provides the `SecureString` class to minimize the length of time-sensitive data should as passwords and private keys are stored in RAM.

6.2 Post-Quantum attacks *Post quantum cryptography* address the fact that several popular cryptography schemes including elliptic curve cryptography can theoretically be easily attacked by a quantum computer.

Observations: None.

Relevance: It is unclear how soon quantum computers will be able to attack the cryptography used in this project.

Improvements: None.

Suggestions: There are four popular *Post quantum cryptography systems*. Two support key exchanges and two support signatures with different sizes and performance tradeoffs. Once higher priorities items are addressed, it would be prudent to investigate which if any of those systems could protect a future version of the blockchain.

7 Client and server

Every wallet is both a client and a server. At present, each wallet's client connects to 4 default server IP addresses via a peer seed endpoint. Clients are pushed new blocks and transactions from the lead adjudicator and potentially these 4 peers. The servers expose several methods to clients. Some of the notable ones are `SendAdjMessageSingle`, `SendAdjMessageAll`, `ReceiveBlock`, `SendBlock`, and `SendTxToMempool`.

7.1 Empty block exploit It is easy to craft blocks that only contain a block reward transaction. If that block is accepted by peer nodes without being distributed as a winning block, then those peers no longer accept blocks from the adjudicator.

Observations: With the current code, it is possible for a masternode to send any type of adjudicator message to all other nodes as if the message was written by the adjudicator via `P2PAdjServer.SendAdjMessageAll`. I tested this with a status message and confirmed the status message was broadcasted to my nodes.

Relevance: In particular, it is possible for a masternode to craft a block with only a `Coinbase_BlkRwd` transaction and send it to `P2PAdjServer.SendAdjMessageAll`. Timed correctly, this would cause the lead adjudicator to broadcast the crafted as the winning block before broadcasting the block that was chosen to win.

Another version of the exploit consists of using `ReceiveBlock` to send crafted empty blocks of the next block height directly to peers. Those peers would in turn rebroadcast the empty block to all their peers and so forth. Empty blocks can also be sent to peers that request blocks while their chain is synchronizing.

These exploits were found simply by examining the API code as would be most exploits of this nature.

Improvements: `P2PAdjServer.SendAdjMessageAll` is now a private method that is not an exposed API method.

Suggestions: Without modifying the consensus algorithm, the most direct solution to prevent the acceptance of blocks not chosen by the adjudicator is to require adjudicators to also sign blocks. However, this solution requires carefully deciding what should happen if two adjudicators partitioned from each other sign and broadcast two different blocks. Network partition attacks could lead to double spending.

7.2 Denial of service A denial of service attack consists of making requests to an API in such a manner that diminishes the service provided by the API to other nodes.

Observations: The code currently has no protection against denial of service attacks.

Relevance: Usage of the API is free. Therefore, the cost to attack the network is low.

Improvements: Recently, a one request per second rate limiting protocol with an exponential delay has been added to client and server methods to the code in the developer branch. Every IP address has a 5 MB restricted buffer before messages are dropped. The core logic is implemented via the following code:

```
var prev = Interlocked.Exchange(ref Lock.LastRequestTime, now);
if (Lock.ConnectionCount > 20)
    Peers.BanPeer(ipAddress, ipAddress + ": Connection count exceeded limit.
    Peer failed to wait for responses before sending new requests.",
    func.Method.Name);
```



```

if (Lock.BufferCost + sizeCost > 5000000)
{
    throw new HubException("Too much buffer usage. Message was dropped.");
}
if (now - prev < 1000)
    Interlocked.Increment(ref Lock.DelayLevel);
else
{
    Interlocked.CompareExchange(ref Lock.DelayLevel, 1, 0);
    Interlocked.Decrement(ref Lock.DelayLevel);
}

```

The exponential penalty will greatly incentivize participating nodes to refrain from spamming their peers with excessive network requests in a small period of time, which should preserve resources for other participating nodes.

Suggestions: To support multiple transactions per second, transactions should be queued into batches that are distributed at most once per second.

7.3 Persistent mempool saturation The mempool is saturated when the flow of incoming transactions exceeds the flow of transactions being included in accepted blocks.

Observations: It is possible that with enough users sending transactions the mempool will have a higher flow of incoming transactions than what the code allows to be processed in a given period.

Relevance: A proven solution would be to allow fees to increase to make persistent mempool saturation reach a market equilibrium until the demand for layer 1 transactions diminishes. That solution will not work for ReserveBlock since they have chosen to make low fees and fast transactions their top priorities.

Improvements: None.

Suggestions: In the context of mempool saturation the flow of new transactions must be reduced. It could be worthwhile to consider dynamically reducing the number of transactions allowed to be sent in bulk as the mempool increases. That will likely resolve the issue. If the saturation remains, then a greater than one-second delay should be used in the denial of service protection.

Transactions that are sending more RBX should be prioritized over transactions that are sending less to minimize the effectiveness of an intentional mempool attack.

7.4 Centralized network topology A network is centralized if there is a small percentage of nodes whose removal would disconnect the network.

Observations: There is currently no finalized decision as to how to best have node traffic rely more on peers than a handful of hard-coded adjudicator nodes.

Relevance: At present, the disconnection of a small number of nodes can take down the network.

Improvements: None.

Suggestions: The transaction and block transmission protocols should balance using hashes when possible, strategically using some redundancy without using too much, being resilient to malicious nodes, having increased robustness with more nodes, and ensuring eventual consistency with exponential convergence. While insight should be drawn from other blockchains such as Bitcoin [GKL15] for efficient transmission, a strategy that achieves round ribbon network configuration [CMS04] will likely lead to the best use of network resources.

7.5 Instant send tradeoff An instant send transaction consists of one that participants have a reason to accept as finalized before a new block has been accepted.

Observations: Existing transactions must be included in an accepted block to be safely considered confirmed. Some blockchains offer an instant send option which might be a requested feature.

Relevance: The advantage of blocks is they enable eventual consistency of transactions. It is okay if different nodes have different transactions in their mempool when blocks are crafted. In contrast, Instant sending requires full consistency to prevent double-spending.

Improvements: None.

Suggestions: Instant spending could be provided in a centralized manner. For example, adjudicators could receive instant spend requests, transmit confirmations to their recipients, and then verify the next winning block contains no transactions from the spending addresses. If the winning block contains another transaction from a spending address, then the adjudicator should notify the respective recipient the instant spend was canceled. Then the adjudicator can wrap the winning block with all remaining instant spend transactions and their signature.

8 Metrics

Metrics are indicators that help people decide if and how they wish to participate in the network.

Observations: The *ReserveBlock Metrics* page states 82 million RBX have been mined out of a lifetime supply of 372 million RBX, 0.0155 RBX have been burned, and there are 3550 active masternodes.

Relevance:

8.1 Metrics in the limit Once masternodes have fully mined the lifetime supply of RBX, they will no longer have any incentive to keep mining. Furthermore, transaction burning will slowly erode the total supply of RBX until no more transactions can be sent.

The maximum possible number of masternodes there will ever be is 372,000. While efficiently coordinating that many nodes are possible with load balancing, the cost is and will likely continue to be significant. It is unclear what if any incentives an adjudicator would have to properly coordinate a large number of nodes.

Improvements: None.

Suggestions: At some point the equations for Masternode compensation and transaction burning will need to change.

9 Masternodes

Masternodes are responsible and rewarded for crafting blocks. While all full nodes are responsible for verifying only valid transactions exist in blocks, those that craft blocks should ensure transaction censorship is infeasible, adjudicators have minimal work while the network scales, and in theory, they vote on changes to the ReserveBlock protocol. These four responsibilities are referred to as security, decentralization, scalability, and democracy.

9.1 Validation complexity Cyclomatic complexity measures the number of linearly independent paths through a program's source code.

Observations: No logical errors were found with block and transaction validation. The functions `BlockValidatorService.ValidateBlock` and `TransactionValidatorService.VerifyTX` respectively have cyclomatic complexities of 89 and 53 as computed by Visual Studio's Code Metrics. The [Wik22a] article states a complexity over 50 is considered "Untestable code, very high risk".

Relevance: The validation code is critical for determining which transactions will be accepted. The high complexity substantially increases the chances of difficult-to-spot bugs, and it made future enhancements more difficult to implement correctly.

Improvements: None.

Suggestions: All high-complexity functions should be refactored and simplified. The code has matured enough that properly abstracting the repeated code segments should be feasible and beneficial.

9.2 Masternode censorship Transaction censorship consists of crafting blocks with certain types of transactions being consistently excluded from accepted blocks.

Observations: I have sent several transactions and have not observed any of them failing. If many transactions are sent at once, it seems to cause nodes to disconnect from the adjudicator. This is understandable since adjudicators rebroadcast transactions to all peers.

Relevance: If a fixed percentage of masternodes p and the lead adjudicator are running unaltered code, then due to the random selection of masternodes the probability any given transaction reaches finality within n blocks exponentially converges to 1. In other words, if the masternode successfully rebroadcasts transactions to all nodes, then even a modest percentage of validators running unmodified code will make transaction censorship infeasible assuming the adjudicator is properly running unmodified code. Therefore, if the mempool is not saturated and the network is well connected, then all valid, broadcasted transactions should eventually be added to accepted blocks. A key remaining issue is to address the scalability associated with sufficient connectivity without sacrificing censorship resistance.

Improvements: None.

Suggestions: To maintain scalability it should not be assumed adjudicators will always rebroadcast transactions to all nodes. If the adjudicator is running unaltered code and the number of nodes seeking to censor a transaction exceeds the number of nodes the transaction is broadcasted to, then there is a chance (likely small) that masternodes could censor the transaction.

It might be worthwhile for nodes to keep track of their belief of other nodes' reputations. With a proper formulation of reputation, peers with a better reputation should be given priority when broadcasting blocks and transactions.

9.3 Submit time moral hazard Validators are incentivized to try to win blocks. Therefore, the project should make sure the rules for winning blocks create appropriate incentives.

Observations: Masternode ties are currently broken by the earliest submission time.

Relevance: This incentivizes masternodes to ignore transactions and disregard the default answer submission flow to answer as soon as a new block is received.

Improvements: None.

Suggestions: Ties can be solved randomly without changing the consensus model.

10 Adjudicators

In the present code, there is a single hard-coded IP address referred to as the lead adjudicator that

1. receives `TaskQuestion` requests,
2. receives transactions,
3. determines winning blocks and broadcasts them,
4. broadcasts transactions.

The lead adjudicator checks every 2 seconds if at least 28 seconds have elapsed since the last time they finished requesting new blocks. After the time has elapsed, a valid winning block is randomly selected among those received from the `TaskQuestion` requests.

10.1 FortisPool exploit The Fortis pool determines which nodes are considered during block submission.

Observations: By inspecting the code, one can see it is possible for any node the lead adjudicator's client has a subscription with to call `ClientCallService.DoFortisPoolWork`.

Relevance: This enables such nodes to set the adjudicator's Fortis pool to be whatever they want. In particular, a node that requests at the right time could force the adjudicator to select them as a winning node.

Improvements: None

Suggestions: The client and server APIs need to be minimized and continually scrutinized to ensure no one is making unintended requests.

10.2 Trust is required When considering processes that are claimed to be random, it is possible they are biased even if the bias is not obvious.

Observations: I compared several thousand runs of a block submission simulation and compared the results with 2 weeks worth of actual block submissions when the active validator count was relatively stable. The distribution of the number of blocks won per address during the period had similar results to the simulation. So, there was no statistical evidence found that suggests the lead adjudicator is not randomly selecting winning blocks.

Relevance: However, there might be other adjudicators, and potentially a different analysis could detect a discrepancy in the randomness of chosen blocks. If an honest adjudicator's node were compromised via a technical attack or gag order, the accepted blocks could be manipulated without their consent.

Improvements: None.

Suggestions: There are consensus algorithms compatible with proof of stake that does not require trusted nodes. For example, a deterministic consensus algorithm using round-robin selection [RG21] would be trustless, and a consensus algorithm that uses a verifiable random function such as Cardano's *Ouroboros Protocol* might require some, but less trust.

10.3 Skin in the game It is not rational to expect people that are not properly incentivized to act in the best interests of the network to do so.

Observations: The RBX whitepaper found at reserveblock.io adjudicators are entirely altruistic.

Relevance: If adjudicators have no legal or financial incentive to run unmodified software in a highly secure environment, then then it could be in their favor to deceptively cheat or short RBX and attack the network. If an adjudicator earns no rewards, they will be subsidizing the network with server costs. This does not align their long-term interest with the rest of the network.

Improvements: None.

Suggestions: Adjudicators should be required to stake a large amount of RBX and should earn some ongoing RBX for their stake.

10.4 Network partition A Network partition would be a situation where one set of nodes is crafting and distributing blocks to each other independently of another set of nodes also crafting and distributing a different set of blocks.

Observations: Most blockchains either implicitly assume network partitions will not happen (e.g. those that claim instant finality) or rely on their consensus mechanism to address the issue if will ever arise. The current RBX codebase assumes network partitions will not occur since there is no mechanism to resolve a block conflict.

Relevance: If more than one node can serve as an adjudicator, then if those nodes lose connection to each other it is possible both will accept and distribute a different set of blocks.

Improvements: None.

Suggestions: There should be a predefined consensus mechanism to determine the priority adjudicators' nodes attempt to connect to and which ones have chosen the correct block in the case of a conflict. The priority of adjudicators can be hard coded in the same way the lead adjudicator is currently hard coded. The same priority can be used to decide which chain wins consensus. However, it might make sense to use the most popular block hash to determine which one wins considering the higher priority adjudicator could have been partitioned into a small network.

11 Beacons / NFT relays

The purpose of beacons is to enable NFT creators to send their associated content to those who purchase the NFT via a smart contract with changing network configurations or communicate in a side channel. Currently, the creator uploads their content onto a beacon's storage device which at a later time can be downloaded by whoever purchased the NFT.

11.1 Storage concerns A beacon serves as a mechanism to allow two peers to share an NFT without either of them hosting a server.

Observations: The original plan for beacons was to store NFTs on their machines.

Relevance: NFTs could take up significant space. People might treat beacons as cloud providers like DropBox or Google Drive. Creators might wish to solely share their content with the purchaser.

Improvements: None.

Suggestions: Beacons can function as a server that allows the wallets of the content provider and the purchaser to ping them whenever they are available. When they are both available, the server can notify both parties in their subsequent ping request to respectively upload and download the data while the beacon simply maintains an in-memory buffer until the process is finished. This data could be end-to-end encrypted which would eliminate any privacy concerns.

12 Smart contracts

The ReserveBlock smart contract functionality is still being developed. Right now, the contracts can illustrate the Trillium language they are based on, but they have no access to external input or output functionality.

12.1 Turing completeness A Turing-complete language is one in which all possible computational algorithms can be expressed.

Observations: It is well-known that if a programming language can simulate the *General recursive functions*, then it is Turing-complete. Since the Trillium language enables implementing primitive functions, unrestricted recursion, composition, and addition, it can simulate any general recursive function and is Turing-complete.

One can test sample Trillium programs in modern web browsers at <https://trillium.rbx.network>. When executing programs, the Trillium IDE displays detected errors. Here is an example of code where the execution time is highly dependent on the input for which no errors are detected.

```
function Ackermann(a : int, b : string) : string
{
    if(a == 0)
        return string(int(b) + 1)
    if(int(b) == 0)
        return Ackermann(a - 1, "1")
    return Ackermann(a - 1, Ackermann(a, string(int(b) - 1)))
}
```

The execution of `Ackermann(3, "3")` takes less than a second. The execution of `Ackermann(5, "5")` continues to run for over a minute before the browser stalls.

Relevance: Ideally, runtime would be proportional to contract length. It is well-known that no algorithm that reliably predicts execution time (or even the termination of) a given program written in a *Turing-complete* language can exist. Therefore, there must be some external mechanism to limit the computational resources of contracts written in Trillium. If such a mechanism is nondeterministic, then nodes can disagree on valid transactions. Ethereum’s virtual machine enables a deterministic economic incentive model to restrict resources used by contracts. Since low fees are a priority for ReserveBlock, there must be explicit contract restrictions in place rather than economic implicit ones.

Improvements: None.

Suggestions: Either contract writers must accept contracts that are not guaranteed to execute or the smart contract language should be restricted in a manner that guarantees all transactions that compile will execute.

If a contract relies on data that is only known at runtime, then it is impossible to predict how long the contract will take to run in general unless every possible runtime value is simulated. This is why Ethereum contracts fail to finalize if their gas runs out. Contract failure has been one source of Ethereum contract exploits.

A well-designed domain-specific language could be an elegant solution that avoids the security bugs [ABC17] discovered and are yet to be discovered in Solidity. Domain-specific languages can be easier to validate, have more predictable run-time behavior, minimize leaky abstractions such as can be found with the <-conventions of existing contracts, and give users flexibility without making it feasible for them to craft exploits.

13 Disclaimer

The lead developer of this project, Aaron Mathis, on behalf of the RBX Foundation contracted me to perform this audit without bias. This audit was in part funded by RBX coins. With respect to any conflicts of interest, I have reviewed and commented on this project as I would on any other project

References

- [ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A survey of attacks on ethereum smart contracts (sok)”. In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186.
- [Car22] Cardano. *Ouroboros Protocol*. <https://cardano.org/ouroboros>. 2022.
- [CMS04] Andrea EF Clementi, Angelo Monti, and Riccardo Silvestri. “Round robin is optimal for fault-tolerant broadcasting on wireless networks”. In: *Journal of Parallel and Distributed Computing* 64.1 (2004), pp. 89–96.
- [Dav22] Mauricio David. *LiteDB*. <https://github.com/mbdavid/LiteDB>. 2022.
- [Fou22] ReserveBlock Foundation. *ReserveBlock Metrics*. <https://rbx.network/metrics>. [Online; accessed 29-August-2022]. 2022.
- [GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. “The bitcoin backbone protocol: Analysis and applications”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 281–310.
- [Mat22] Aaron Mathis. *ReserveBlock-Core*. <https://github.com/ReserveBlockIO/ReserveBlock-Core>. 2022.
- [Mic22a] Microsoft. *.NET 6.0*. <https://dotnet.microsoft.com/en-us/download/dotnet/6.0>. 2022.
- [Mic22b] Microsoft. *SignalR*. <https://github.com/dotnet/aspnetcore/tree/main/src/SignalR>. 2022.

- [Mic22c] Microsoft. *Post quantum cryptography systems*. <https://www.microsoft.com/en-us/research/project/post-quantum-cryptography>. 2022.
- [RG21] Mayank Raikwar and Danilo Gligoroski. “R3V: Robust Round Robin VDF-based Consensus”. In: *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE. 2021, pp. 81–88.
- [Wik22a] Wikipedia. *Cyclomatic-complexity*. https://en.wikipedia.org/wiki/Cyclomatic_complexity. 2022.
- [Wik22b] Wikipedia. *General recursive functions*. https://en.wikipedia.org/wiki/General_recursive_function. 2022.
- [Wik22c] Wikipedia. *Post quantum cryptography*. https://en.wikipedia.org/wiki/Post-quantum_cryptography. 2022.
- [Wik22d] Wikipedia. *Turing-complete*. https://en.wikipedia.org/wiki/Turing_completeness. 2022.

ReserveBlock Core Post Audit Report

Reviewed by Aaron Mathis

Lead Developer on ReserveBlock Blockchain

Contents

1. Preface.....	3
2. The Relevant Experience of the Auditor.....	3
3. Testing Methodology.....	3
4. Startup Processes.....	4
4.1 - Long wait to access wallet functionality.....	4
4.2 - Potentially Large Database Size.....	5
4.3 - Thread Safety.....	6
5. Databases.....	7
5.1 - Denormalization.....	7
5.2 - Unfiltered Queries.....	9
5.3 - Indices on Natural Keys.....	9
5.4 - Updating via Natural Keys.....	10
5.5 - The $n + 1$ Problem.....	10
5.6 - Block Corruption.....	10
6. Cryptography.....	12
6.1 - Storage of Private Keys.....	12
6.2 - Post-Quantum Attacks.....	13
7. Client and Server.....	13
7.1 - Empty Block Exploit.....	13
7.2 - Denial of Service (DoS).....	13
7.3 - Persistent Mempool Saturation.....	14

- 7.4 - Centralized Network Topology 16
- 7.5 - Instant Send Tradeoff 17
- 8. Metrics..... 17
 - 8.1 - Metrics in the Limit..... 17
- 9. Masternodes..... 18
 - 9.1 - Validation Complexity 18
 - 9.2 - Masternode Censorship 19
 - 9.3 - Submit Time Moral Hazard 19
- 10. Adjudicators..... 20
 - 10.1 - FortisPool Exploit 20
 - 10.2 - Trust is Required 20
 - 10.3 - Skin in the game 20
 - 10.4 - Network Partition 21
- 11. Beacon / NFT Relays..... 21
 - 11.1 Storage Concerns 21
- 12. Smart Contracts..... 22
 - 12.1 - Turing Completeness 22
- 13. Disclaimer..... 24
- Conclusion..... 24

1. Preface

This document is a post audit report on the findings from the “ReserveBlock Core August 2022 Audit” known as the “audit”. It is meant as a supplemental document to audit and will describe what steps have been taken to resolve or improve upon the findings. Not all findings in the audit merited a change and some findings in the audit are subjective to philosophical opinion. Both documents are meant to support one another and should be reviewed in tandem.

2. The Relevant Experience of the Auditor

Alex Williams, known as “the auditor” or “auditor”, portrayed and demonstrated to the project through not only work history, but through proficiency in both C# and Blockchain technology. The auditor having worked on Blockchain projects for exchanges, trade engines, and other Blockchain related projects was a driving factor in selection. The second reason was they have a profound understanding of the core language and offered to not only point out findings in the audit, but also to provide legitimate source code where relevant, to resolve open issues. The auditor also demonstrated strong architectural skills and required very little guidance and deployed suggested code in a proficient and timely manner. Overall, the auditor proved to have both the skills and knowledge to provide a thorough report.

3. Testing Methodology

The auditor provided his methods for testing and they were deemed to be both useful and proficient. They revealed a few non thread-safe variables that were local in-memory. The auditor studied the code “file-by-file” and has multiple well-coordinated calls to go over any questions in regards to existing source code for explanation when

requested. The testing measures of the auditor were extremely helpful in revealing areas of optimization, as well as, a few areas of minor exploits.

4. Startup Processes

The startup processes in this section were all noted, with no issues reported. The auditor points out the network DBs are used with the LiteDB engine and the Peer-to-Peer (P2P) protocol is maintained with the help of SignalR.

4.1 - Long wait to access wallet functionality

This was due to chain sync on the state treis files. This was resolved by improving the LiteDB functions and removing conflicting inserts that cause the data to experience denormalization and need a sync tool. Since removing the sync tool wallet startup has returned to a normal state. The sync tool was ran until the update (v2.1) to ensure all databases have a synced and same treis and no denormalization occurs. Also users can run the sync manually if they choose with the /synctreis command.

Running the /synctreis will call the `public static async Task SyncAccountStateTrei()` call. This method will ensure in the event you have a system crash or other issues with settlement due to localized problems (issues with the node/machine not the network) that the treis are in sync. You can find this method in the StateTreiSyncService.cs file. Below is the use of how it will output in the CLI


```
public static async Task SyncTreis()
{
    Console.WriteLine("Beginning Trei Sync");
    await StateTreiSyncService.SyncAccountStateTrei();
    Console.WriteLine("Trei Sync has completed. Please check error log for report on balances updated.");
}
```

```

RBX wallet
ReserveBlock Main Menu
=====
| 1. Genesis Block (Check)
| 2. Create Account
| 2hd. Create HD Wallet
| 3. Restore Account
| 3hd. Restore HD Wallet
| 4. Send Coins
| 5. Get Latest Block
| 6. Transaction History
| 7. Wallet Address(es) Info
| 8. Startup Masternode
| 9. Search Block
| 10. Enable API (Turn On and Off)
| 11. Stop Masternode
| 12. Import Smart Contract (disabled)
| 13. Exit
=====
|type /help for menu options
|type /menu to come back to main area
=====

/synctrei
Beginning Trei Sync
Trei Sync has completed. Please check error log for report on balances updated.

```



```

errorlog.txt - Notepad
File Edit Format View Help
[11/9/2022 8:44:44 PM] : [StateTreiSyncService()] : Balance Off: RFmB1nNn426MmqvUviSnWt4pxfunt3SRcn | Reported: 1320.999988505859375 - Actual: 1288.999988505859375
[11/9/2022 8:44:44 PM] : [StateTreiSyncService()] : Balance Off: RK3UEV1ntpfh2r7vieWlqvXgS899VfKEeo | Reported: 1544.00 - Actual: 1512.00

```

This was a simulated example above, but this is an example of how it would and can presently run in mainnet.

4.2 – Potentially Large Database Size

While this is something that is an accepted burden in blockchain technology, the core wallet does have pruning on its list of features to be added currently. While the risk of the size growing too large is not of concern today, adding pruning will alleviate any potential constraints in the future.

The auditor also deduced that block downloads were happening synchronously amongst the nodes and created a parallel download tool that allowed clients to now download blocks from nodes in async and in parallel to increase block download performance for now and in the

future. There was also code added to seek out higher bandwidth peers to increase performance in that regard as well. Below is a code excerpt provided by auditor and implemented in the RBX Core client.

```
public static async Task DropLowBandwidthPeers()
{
    await DropDisconnectedPeers();

    var PeersWithSamples = Globals.Nodes.Where(x => x.Value.SendingBlockTime > 60000)
        .Select(x => new
        {
            IPAddress = x.Key,
            BandWidth = x.Value.TotalDataSent / ((double)x.Value.SendingBlockTime)
        })
        .OrderBy(x => x.BandWidth)
        .ToArray();

    var Length = PeersWithSamples.Length;
    if (Length < 3)
        return;

    var MedianBandWidth = Length % 2 == 0 ? .5 * (PeersWithSamples[Length / 2 - 1].BandWidth + PeersWithSamples[Length / 2].BandWidth) :
        PeersWithSamples[Length / 2 - 1].BandWidth;

    foreach (var peer in PeersWithSamples.Where(x => x.BandWidth < .5 * MedianBandWidth))
    {
        if(Globals.Nodes.TryRemove(peer.IPAddress, out var node))
            await node.Connection.DisposeAsync();
    }
}
```

In the future it would be prudent to implement a block clustering system that allows for more than 1 block at a time and can be accepted at the masternode level and implemented by devs, however, at this time the chain is syncing in a very quick manner, and it is not currently needed.

4.3 - Thread Safety

The auditor went through the entire code base and found non-thread safe items and improved using locks, semaphores, and concurrent data structures like ConcurrentDictionary and ConcurrentBag. Moving forward these models will continue to be adopted throughout the code to ensure thread safety remains at a high priority.

Some examples of this in mainnet are the following variables:

```
public static ConcurrentQueue<Block> MemBlocks = new ConcurrentQueue<Block>();
public static ConcurrentDictionary<string, NodeInfo> Nodes = new ConcurrentDictionary<string, NodeInfo>(); // IP Address
public static ConcurrentDictionary<string, Validators> InactiveValidators = new ConcurrentDictionary<string, Validators>(); // RBX address
public static ConcurrentDictionary<string, string> Locators = new ConcurrentDictionary<string, string>(); // BeaconUID
public static ConcurrentBag<string> RejectAssetExtensionTypes = new ConcurrentBag<string>();
```

The ConcurrentMultiDictionary is a prime example of using these in a thread safe manner. You can see this code in the ConcurrentMultiDictionary.cs file. Below is a code example:

```
private ConcurrentDictionary<K1, (K2, V)> Dict1 = new ConcurrentDictionary<K1, (K2, V)>();
private ConcurrentDictionary<K2, (K1, V)> Dict2 = new ConcurrentDictionary<K2, (K1, V)>();
private object WriteLock = new object();
private bool UseDict2 = false;

3 references
public V this[(K1, K2) key]
{
    set {
        lock (WriteLock)
        {
            if (Dict1.TryGetValue(key.Item1, out var Out1))
            {
                var Comparer = EqualityComparer<K2>.Default;
                if (!Comparer.Equals(Out1.Item1, key.Item2))
                    Dict2.TryRemove(Out1.Item1, out _);
            }

            if (Dict2.TryGetValue(key.Item2, out var Out2))
            {
                var Comparer = EqualityComparer<K1>.Default;
                if (!Comparer.Equals(Out2.Item1, key.Item1))
                {
                    UseDict2 = true;
                    Dict1.TryRemove(Out2.Item1, out _);
                }
            }

            Dict1[key.Item1] = (key.Item2, value);
            Dict2[key.Item2] = (key.Item1, value);
            UseDict2 = false;
        }
    }
}
```

5. Databases

5.1 – Denormalization

The auditor was requested to help find the conflict in a header record that was causing a denormalization of one of the treis. This denormalization was corrected in mainnet with the state sync, but this was not a desired permanent solution. The auditor provided an elegant solution that did not require a refactor or use of another library or tools. “The validation code was modified to properly use transactions to make sure balance updates occur in lockstep with the addition of new blocks.” from the audit. This can be reviewed in the SafeDbExtensions.cs file in the source code.

SemaphoreSlim was leverage to ensure DBs are performing in a manner to prevent any kind of denormalization. Two commands for Func<S> and Action were created to allow for this in the code.

```
23 references
public static S Command<S, T>(this ILiteCollection<T> col, Func<S> cmd)
{
    var DbSemaphore = col.GetSlim();
    DbSemaphore.Wait();
    try
    {
        return cmd();
    }
    finally
    {
        if (DbSemaphore.CurrentCount == 0)
            DbSemaphore.Release();
    }
}

1 reference
public static void Command<T>(this ILiteCollection<T> col, Action cmd)
{
    var DbSemaphore = col.GetSlim();
    DbSemaphore.Wait();
    try
    {
        cmd();
    }
    finally
    {
        if (DbSemaphore.CurrentCount == 0)
            DbSemaphore.Release();
    }
}
```

From here updates and inserts will now use the following “safe” methods.

```
public static int UpdateSafe<T>(this ILiteCollection<T> col, IEnumerable<T> entities)
{
    return Command(col, () => col.Update(entities));
}
```

```
public static int InsertSafe<T>(this ILiteCollection<T> col, IEnumerable<T> entities)
{
    return Command(col, () => col.Insert(entities));
}
```


5.2 - Unfiltered Queries

The auditor was tasked with finding inefficient or improper queries that could cause performance issues down the life of the project due to large amounts of blocks. The auditor went through and improved all queries, if needed, in all files of the project to ensure only the needed amounts of records are needed.

Also the lead developer added an in-memory set of blocks that function to perform many task both speedily and accurately.

The blocks are now Dequeued from the variable and the latest is adding keeping a record of the most recent 300 blocks in memory for faster querying and block processing rather than reading from the hard drive every time.

```
internal static void StartupMemBlocks()
{
    var blockChain = BlockchainData.GetBlocks();
    Globals.MemBlocks = new ConcurrentQueue<Block>(blockChain.Find(LiteDB.Query.All(LiteDB.Query.Descending), 0, 300));
}
```

```
2 references
public static void UpdateMemBlocks(Block block)
{
    Globals.MemBlocks.TryDequeue(out Block test);
    Globals.MemBlocks.Enqueue(block);
}
```

5.3 - Indices on Natural Keys

The largest issue here was to ensure that the protocol performs an EnsureIndex on the proper tables. This is now down at startup in the DbContext.cs. The code below shows required indexes have now been ensured and in a thread safe manner.

```
var blocks = DB.GetCollection<Block>(RSRV_BLOCKS);
blocks.EnsureIndexSafe(x => x.Height);

var transactionPool = DbContext.DB.GetCollection<Transaction>(DbContext.RSRV_TRANSACTION_POOL);
transactionPool.EnsureIndexSafe(x => x.Hash, false);
transactionPool.EnsureIndexSafe(x => x.FromAddress, false);
transactionPool.EnsureIndexSafe(x => x.ToAddress, false);

var transactions = DbContext.DB_Wallet.GetCollection<Transaction>(DbContext.RSRV_TRANSACTIONS);
transactions.EnsureIndexSafe(x => x.Hash, false);
transactions.EnsureIndexSafe(x => x.FromAddress, false);
transactions.EnsureIndexSafe(x => x.ToAddress, false);

var aTrei = DbContext.DB_AccountStateTrei.GetCollection<AccountStateTrei>(DbContext.RSRV_ATYPE_TREI);
aTrei.EnsureIndexSafe(x => x.Key, false);
```

5.4 - Updating via Natural Keys

This is a design decision and while the auditor is not wrong, this is also not a problem within the code and no change is needed, however, in areas of simple table records with natural or singular keys a pattern could be implemented as the auditor described, however, no queries are done in these areas of large intent thus not causing any performance issues.

5.5 - The $n + 1$ Problem

While this is an interesting observation by the auditor an improvement is not needed at this time. As once synced new blocks come in only one at a time, and so each block needs to be processed immediately as block times are relatively fast (25 seconds), so they are processed immediately to create space and free up resources for next incoming block.

There were some areas where the network does batch processes to ensure no bottlenecks are reached. One example would be in how the network processes the mempool queues to ensure the entire pool is processed before ever crafting the transactions into a block and that they are removed to ensure no duplication of these items are kept. This is also joined with the very strict consensus rules.

5.6 - Block Corruption

The auditor provided “code that proved it is feasible to craft blocks with any given previous hash that would be accepted as valid even if

it was not considered a winning block (assuming the winning block has not already been added). These crafted blocks could be given to peers that request them for downloading”.

Another system of peer banning logic was added to also not only reject bad/corrupt blocks, but to also ban the peer attempting to spread them. The following code shows if a block does not pass validation then it can result in a peer being banned.

```
foreach (var height in heights)
{
    if (!BlockDownloadService.BlockDict.TryRemove(height, out var blockInfo))
        continue;
    var (block, ipAddress) = blockInfo;
    var result = await ValidateBlock(block, true);
    if (!result)
    {
        Peers.BanPeer(ipAddress, ipAddress + " at height " + height, "ValidateBlocks");
        ErrorLogUtility.LogError("Banned IP address: " + ipAddress + " at height " + height, "ValidateBlocks");
        if (Globals.Nodes.TryRemove(ipAddress, out var node))
            await node.Connection.DisposeAsync();
        Console.WriteLine("Block was rejected from: " + block.Validator);
    }
    else
    {
        if (Globals.IsChainSynced)
            ConsoleWriterService.OutputSameLineMarked($"Time: [yellow]{DateTime.Now}[/] | Block [green]({block.Height})[/]");
        else
            Console.Write($"\\rBlocks Syncing... Current Block: {block.Height} ");
    }
}
```

There is currently a command to revert back to a specific block height and there is a service that will create checkpoints for a client to create snapshots of the current state of all database files allowing for an easy recovery or transfer of a node. This can be found in the BlockRollbackUtility.cs

```
public static async Task<bool> RollbackBlocks(int numBlocksRollback)
{
    Globals.IsResyncing = true;
    Globals.StopAllTimers = true;
    var height = Globals.LastBlock.Height;
    var newHeight = height - (long)numBlocksRollback;

    var blocks = Block.GetBlocks();
    blocks.DeleteManySafe(x => x.Height > newHeight);
    DbContext.DB.Checkpoint();

    var result = await ResetTreis();

    Globals.IsResyncing = false;
    Globals.StopAllTimers = false;

    return result;
}
```

6. Cryptography

6.1 – Storage of Private Keys

The auditor pointed out rather quickly that there is storing of private keys in plain text. While this is not an issue if good OPSEC is followed, there still needed to be a way to allow clients to protect keys should a wallet database file be stolen, or leaked.

The solution was to allow the encryption of private keys.

Wallet encryption works with keystore DB and AES encryption as well as a user provided password. The user inputs a password and that password is then used to encrypt the encryption keys that encrypt private keys in the keystore. At the time of encryption 1000 addresses are generated so the user does not have to type in their password to create new addresses. Should they reach this threshold a password is requested and a new set of addresses is generated.

If desired the WalletEncryptionService.cs file outlines the algorithms used and the Keystore.GenerateKeystoreAddresses() method can be reviewed to see the generation of keys.

6.2 - Post-Quantum Attacks

Something all projects need to be mindful of is the use of quantum machines to attack the keys created. It was determined that while the threat itself is real, the timeline of this threat is deemed to be far enough out that a scheduled refactor of the keys can be performed in the future.

The project has been following the Brown mathematicians' four algorithms for a post-quantum era and can integrate one of the solutions before the issue potentially can become a risk.

7. Client and Server

7.1 - Empty Block Exploit

This was resolved by requiring an adjudicator to sign a block and will no longer allow an empty block to be submitted to exploit a guaranteed win. The following code shows where a signature is now required in the adjudicator processes.

```
//process winners block
//1.
var signature = await AdjudicatorSignBlock(winnersBlock.WinningBlock.Hash);
winnersBlock.WinningBlock.AdjudicatorSignature = signature;
var result = await BlockValidatorService.ValidateBlock(winnersBlock.WinningBlock);
```

7.2 - Denial of Service (DoS)

The auditor was tasked with reviewing areas where DoS events could occur and then provide code to help resolve these types of issues. Below is an excerpt of the code to help prevent a node attempting to spam from a singular IP address.

```
public static async Task<T> SignalRQueue<T>(HubCallerContext context, int sizeCost, Func<Task<T>> func)
{
    if (Globals.LastBlock.Height <= Globals.BlockLock)
        return await func();

    var now = TimeUtil.GetMillisecondTime();
    var ipAddress = GetIP(context);
    if (Globals.MessageLocks.TryGetValue(ipAddress, out var Lock))
    {
        var prev = Interlocked.Exchange(ref Lock.LastRequestTime, now);
        if (Lock.ConnectionCount > 20)
            Peers.BanPeer(ipAddress, ipAddress + ": Connection count exceeded limit. Peer failed to wait for responses before sending new");

        if (Lock.BufferCost + sizeCost > 5000000)
        {
            throw new HubException("Too much buffer usage. Message was dropped.");
        }
        if (now - prev < 1000)
            Interlocked.Increment(ref Lock.DelayLevel);
        else
        {
            Interlocked.CompareExchange(ref Lock.DelayLevel, 1, 0);
            Interlocked.Decrement(ref Lock.DelayLevel);
        }
    }

    return await SignalRQueue(Lock, sizeCost, func);
}
```

The client benefits from this as it will help protect the non-advanced users who perhaps does not know how to configure a firewall properly, or for those who have improperly configured.

7.3 – Persistent Mempool Saturation

This was an area the client needed review and it was determined the mempool could be saturated with endless amounts of transactions. This is mainly due to the “near-zero” fee structure. A gasless environment does require code in order to prevent or defend this from being exploited.

Two solutions were integrated by the lead developer after audit and are implemented in mainnet now. The first was to create a transaction rating system that would intelligently rate transactions and give them a grade of A-F. This rating also provides as a system of importance. If a client attempts to spam a large amount of micro-transactions they could be given a grade of C or D. This would tell all validators to process them last, if there is any space left in block. Should a user spam many transactions beyond normal thresholds any period, the system will automatically determine that spamming is occurring, and it will begin giving all new transactions for the next block a rating of F. Any transaction with a rating of F will automatically fail.

The second solution implemented, was to allow each address 10 transactions per block. This may appear as an inherent bottleneck, however it really is not as spreading legitimate transactions amongst a few addresses is rather simple and can scale rather easily due to the low fee structure.

This system was tested in testnet and is now in mainnet and the data has proven to resolve spamming of transactions and forces users to act responsibly and limit abuse of the networks mempool.

The above examples can be seen in the code below:

This gives transactions a rating.

```
public static async Task<TransactionRating> GetTransactionRating(Transaction tx)
{
    TransactionRating rating = TransactionRating.F; //start at F in the event something is received we don't expected
    try
    {
        if (tx.TransactionType == TransactionType.TX)
        {
            rating = await TXRating(tx);
        }
        if (tx.TransactionType == TransactionType.ADNRR)
        {
            rating = await ADNRRating(tx);
        }
        if (tx.TransactionType == TransactionType.NFT_MINT ||
            tx.TransactionType == TransactionType.NFT_SALE ||
            tx.TransactionType == TransactionType.NFT_BURN ||
            tx.TransactionType == TransactionType.NFT_TX)
        {
            rating = await NFTRating(tx);
        }
    }
    return rating;
}
catch(Exception ex)
{
    //something failed. should not happen unless TX is malformed or malicious, in which case it gets an F and won't be processed.
    ErrorLogUtility.LogError($"Error rating transaction. TXId = {tx.Hash}. Error Message: {ex.ToString()}", "TransactionRatingService.");
    return rating;
}
```

This will give an NFT transaction a rating.

```
private static async Task<TransactionRating> NFTRating(Transaction tx)
{
    TransactionRating rating = TransactionRating.A;
    var mempool = TransactionData.GetMempool();
    var pool = TransactionData.GetPool();

    if (mempool != null)
    {
        if (mempool.Count() > 10)
        {
            var txs = mempool.FindAll(x => x.FromAddress == tx.FromAddress &&
                (x.TransactionType == TransactionType.NFT_MINT ||
                 x.TransactionType == TransactionType.NFT_SALE ||
                 x.TransactionType == TransactionType.NFT_BURN ||
                 x.TransactionType == TransactionType.NFT_TX));
            if (txs.Count() > 10)
            {
                rating = TransactionRating.F; // Fail. Too many tx's being broadcasted from that address.
                txs.ForEach(x =>
                {
                    x.TransactionRating = TransactionRating.E; // current TXs in mempool have been lowered to protect against spammers
                });
                pool.UpdateSafe(txs);
            }
            else
            {
                rating = TransactionRating.A;
            }
        }
        else
        {
            rating = TransactionRating.A;
        }
    }

    return rating;
}
```

7.4 - Centralized Network Topology

While the auditor was correct in his findings this was purely a beta testing option where the foundation has the network connecting to a few nodes to gather information on P2P protocols. As of today, a client can connect to any node acting as a full node and the network can suffer the loss of multiples of nodes and continue to operate as normal.

The client can now add peers at will and remove or even ban them should they so choose.

This can be seen in the BaseCommandServices.cs file. There are the following methods that will allow you to control the network topology for the client.

- `public static async void AddPeer()`
- `public static async void BanPeer()`
- `public static async void UnbanPeer()`
- `public static async void ReconnectPeers()`

7.5 - Instant Send Tradeoff

The auditor was asked to explore, but not implement anything on the idea of instant sends. This is something on the suggested roadmap as with a masternode like system the network can create trustless plenums to perform specific and targeted tasks. Instant sends would work very simply in that it would allow a client to connect to a network that would allow the client to take an address and instantly send a transaction to another client and it would settle in a later block. This would allow for much faster settlement of NFTs and allow for instant transfers and can be adopted by the network at any time.

8. Metrics

8.1 - Metrics in the Limit

The auditor reported that eventually validators will have mined most of the coin through the life of the block rewards and halving schedule, which is by design and used as an anti-inflationary measure, very similar to that of the Bitcoin Network.

```
public static decimal GetBlockReward()
{
    decimal blockReward = 32.00M;
    int currentBlockHeight = Globals.LastBlock.Height != -1 ? (int)Globals.LastBlock.Height : 1;
    int blockHalvingInterval = 4730400; // Roughly every 3 year halving

    //An int will always result in a rounded down whole number. 4730399 / 4730400 = 0 | 9,460,799 / 4730400 = 1
    int halving = currentBlockHeight / blockHalvingInterval;

    var n = 1;
    while (n <= halving)
    {
        blockReward /= 2;
        n++;
    }

    return blockReward;
}
```

Another anti-inflationary measure is all TX fees are burned and all ADNR creation cost (Currently 1 RBX) is burned as well.

The original code for fees was left for transparency but is commented out thus causing the TX fees to be burned and dropped from the circulating supply.

```
//var coinbase_tx = new Transaction
//{
//    Amount = 0,
//    ToAddress = validator,
//    Fee = 0.00M,
//    Timestamp = timestamp,
//    FromAddress = "Coinbase_TrxFees",
//    TransactionType = TransactionType.TX
//};

var coinbase_tx2 = new Transaction
{
    Amount = GetBlockReward(),
    ToAddress = validator,
    Fee = 0.00M,
    Timestamp = timestamp,
    FromAddress = "Coinbase_BlkJwd",
    TransactionType = TransactionType.TX
};
```

This can be seen in the ADNR transaction creation method where the TX is set to 1.

9. Masternodes

9.1 - Validation Complexity

The auditor also reported that the validation process is complex and while this is not incorrect, this is also an area that must be absolute to guarantee consensus amongst the nodes.

The idea of simplifying the code is not incorrect and has already received simplifications. This can be seen in the form of block version rules, double spend checks, and the replay protection that exist. That referenced code has been stripped out and put into Boolean methods. Continue refactoring of that code can happen over time and in a verifiable way to ensure the risk of losing consensus never occurs.

Version rules have been added based on the height of the chain is when they will activate now.

```
if (block.Version > 1)
{
    //Run block version 2 rules
    var version2Result = await BlockVersionUtility.Version2Rules(block);
    if (!version2Result)
        return result;
}
```

More block validation metrics can be seen in the BlockValidatorService.cs and the method: `public static async Task<bool> ValidateBlock(Block block, bool blockDownloads = false)`

9.2 – Masternode Censorship

This issue was resolved in the latest wallet version of 2.1 mainnet beta. Larger scaling solutions have been implemented to address the concerns in the form of pooled consensus. At present there are no issues presented in the audit left in mainnet. Transactions can be sent so long as a client has 1/8 peers connected and that peer is valid. If the client is a validator, then transactions are sent to all other validators every 3 minutes, however most transactions are completed within 1 block cycle. This does not mean transactions take 3 minutes, but rather validators are tasked with multiple operations and may sometimes take longer to do other functions if they are on weaker machines.

9.3 – Submit Time Moral Hazard

The auditor made a valid point and a group of 30 validators are now selected and out of that should the winner not arise, a random winner is chosen rather than basing anything off of time. These 30 are

selected based on the winning number and are all given an equally random chance should the winner not respond in a timely manner. The winner has 3 seconds to reply plus the time from the 30 others.

10. Adjudicators

10.1 – FortisPool Exploit

The code that allowed this was removed and this exploit has been effectively removed.

10.2 – Trust is Required

Trust in this process has been removed within the adjudicator pool. The auditor as of today has implemented a system that removed trusting an adjudicator and makes the number selection process both encrypted and random for all adjudicators thus making it a trustless plenum now that performs a task within the network void of any trust needed.

This process to reach consensus has a lot of steps involved and can be seen in the `ClientCallService.cs`, `P2PClient.cs`, `P2PAdjServer.cs`, `P2PServer.cs`, and last the `ConsensusServer.cs`.

10.3 – Skin in the game

While the auditor's point is not incorrect this is an opinion-based remark and the network has opted to take a more Proof of Authority model that instead of requiring an adjudicator to put skin in the game they are rather voted into the network and verified in order to remain altruistic. The biggest notable difference is the network separates tasks to allow validators to be a large network currently over 3,500 instead of traditional models that are much smaller doing all tasks.

For the user to be eligible for voting in, they must have a proven history of reliance and uptime as a validator or in similar experience. There is also a minimum spec requirement to ensure the machine is capable of being an adjudicator. The current minimum spec would be 8gb of ram and at least a 4 core CPU.

Should an adjudicator go offline, voting allows the present adjudicator pool to remove and add them as needed, or as validator voting may dictate.

There is no longer a bias amongst adjudicators because the randomized answers are now encrypted and shielded from the adjudicators themselves thus allowing even a compromised adjudicator to not affect the network.

The adjudicators exist in a trustless pool that follows a group consensus on deciding a winner. This allows for both random and stabilized validating on the RBX network. It also provides an inherent layer of redundancy should an adjudicator(s) go offline for any reason or become compromised.

Each adjudicator, can at will, also implement their own vetting processes allowing for a healthy wide range of ideas and concepts that will strengthen the pool and who enter it.

This process to reach consensus has a lot of steps involved and can be seen in the `ClientCallService.cs`, `P2PClient.cs`, `P2PAdjServer.cs`, `P2PServer.cs`, and last the `ConsensusServer.cs`.

10.4 – Network Partition

This was resolved by creating a consensus model within the adjudicator pool. While the auditor says improvement none he is as of today adding the model for this to be in Mainnet before beta exit.

This process to reach consensus has a lot of steps involved and can be seen in the `ClientCallService.cs`, `P2PClient.cs`, `P2PAdjServer.cs`, `P2PServer.cs`, and last the `ConsensusServer.cs`.

11. Beacon / NFT Relays

11.1 Storage Concerns

The auditor pointed out that beacons can become overburdened with data both due to malicious and non-malicious events. This prompted an immediate refactor of the system and now beacons can be owned by

anyone and beacons only act as a p2p relay system and no longer act as a data storage service. Once files are relayed properly they are cleared from memory, thus removing the risk of filling up beacons HDD.

There is also an intelligent queue system that allows a beacon to not waste resources when another node/client is not online to receive the assets of an NFT.

The beacon processes can be viewed in the P2PBeaconServer.cs file.

```
2 | references
public class P2PBeaconServer : Hub
{
    Connect/Disconnect methods
    Send Beacon Locator Info
    Beacon Receive Download Request - The receiver of the NFT Asset
    Beacon Receive Upload Request - The sender of the NFT Asset
    Beacon Data IsReady Flag Set - Sets IsReady to true if file is present
    Beacon Is File Ready check for receiver
    Beacon File Is Downloaded set
    SignalR DOS Protection
    Get IP
}
```

12. Smart Contracts

12.1 - Turing Completeness

The auditor pointed out that Trillium can suffer from both recursive functions as well as infinite loops. This is an inherent factor of Turing complete languages and something that was known to the foundation.

Since the audit Trillium has now removed the ability for code breaking recursion and optional parameters and time locks to prevent both malicious and poorly coded smart contracts from looping forever. This was done mainly due to the fact that the environment is a no gas

system so rather than relying on economics to solve this problem, Trillium itself now can solve the problems when a Self-Executing NFT is ran, thus still keeping the RBX network from needing a gas-like system.

```
public void ReportWhileLoop(TextLocation location)
{
    var message = $"While loop was detected.";
    Report(location, message);
}

public void ReportForLoop(TextLocation location)
{
    var message = $"For loop was detected.";
    Report(location, message);
}

public void ReportRecursion(TextLocation location, string signature)
{
    var message = $"Potential recursion was detected involving:";
    Report(location, message);
}
```

```
var Root = new CompilationUnitSyntax(_syntaxTree, members, endOfFileToken);
if (_syntaxTree.PreventLoopsAndRecursion)
    RecursionCheck(Root);
return Root;
```

```
if (_syntaxTree.PreventLoopsAndRecursion)
    _diagnostics.ReportWhileLoop(Current.Location);
var keyword = MatchToken(SyntaxKind.WhileKeyword);
```

To finalize the process in the Parser.cs file you can review the recursion check under the `private void RecursionCheck(SyntaxNode root)`.

This will show the checks to identify improper method calls that can lead to an infinite recursion loop.

13. Disclaimer

It was requested that the auditor provide this section to remain transparent and to release all information between the auditor and lead developer.

Conclusion

The auditor and the report were both invaluable and essential to ensuring that all steps are being taken to provide a safe, stable and scalable Blockchain product. All items from the audit have been addresses and plans for the future based on this audit have been added to roadmap for the project outside of items addressed immediately.