AUDITONE

Audit Report

AURORA

Bitcoin Light Client

23.09.2024

Table of Contents

CONTENTS

INTRODUCTION

Smart Contract Security Analysis Report

Note: This report may contain sensitive information on potential vulnerabilities and exploitation methods. This must be referred internally and should be only made available to the public after issues are resolved (to be confirmed prior by the client and AuditOne).

INTRODUCTION

Ubermensch3dot0, Berndartmueller and MarioPoneder, who are auditors at AuditOne, successfully audited the smart contracts (as indicated below) of Aurora Btc. The audit has been performed using manual analysis. This report presents all the findings regarding the audit performed on the customer's smart contracts. The report outlines how potential security risks are evaluated. Recommendations on quality assurance and security standards are provided in the report.

01-PROJECT DESCRIPTION

The Aurora environment consists of the Aurora Engine, a high performance EVM—Ethereum Virtual Machine—and the Rainbow Bridge, facilitating trustless transfer of ETH and ERC-20 tokens between Ethereum and Aurora, within a great user experience.

Aurora exists and is operated as an independent, self-funded initiative, but will continue to leverage the shared team DNA and continually evolving technology of the NEAR Protocol.

The governance of Aurora will take a hybrid form of a Decentralized Autonomous Organization—the AuroraDAO—complemented by a traditional entity which will hold one of several seats in the AuroraDAO.

This audit focused on the Bitcoing light client including the merkle tools and the relayer for the ecosystem.

02-Project and Audit Information

Term	Description
Auditor	Ubermensch3dot0, Berndartmueller and MarioPoneder
Reviewed by	Luis Buendia and Gracious Igwe
Type	Near
Language	Rust
Ecosystem	Near
Methods	Manual Review
Repository	https://github.com/Near-One/btc-light-client-contract/
Commit hash (at audit start)	42924163b33c1d00d30e23a3fca73e18ea86260a
Commit hash (after resolution)	613639b979748cc1ef01e98de8b1715acbc8ec44
Documentation	N/A
Unit Testing	N/A
Website	https://aurora.dev/
Submission date	18/08/2024
Finishing date	06/09/2024

03-Contracts in Scope

Contract in scope for folder btc-light-client-contract:

- contract/src/lib.rs

Contract in scope for folder merkle-tools:

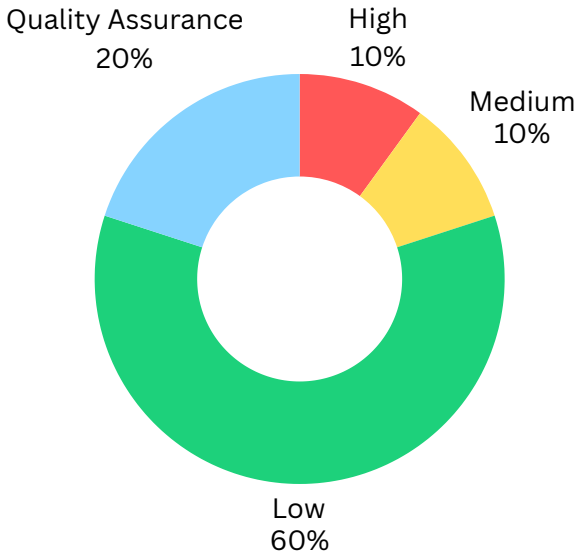
- src/lib.rs

Contract in scope for folder relayer:

- src/bitcoint_client.rs
- config.rs
- main.rs
- near_client.rs

04-Executive summary

Aurora BTC smart contracts were audited between 07-08-2024 and 23-08-2024 by Ubermensch3dot0, Berndartmueller and MarioPoneder. Manual analysis was carried out on the code base provided by the client. The following findings were reported to the client. For more details, refer to the findings section of the report.



Issue Category	Issues Found	Resolved	Acknowledged
High	1	1	0
Medium	1	0	1
Low	6	5	1
Quality Assurance	2	2	0

05-Severity Definitions

Risk factor matrix	Low	Medium	High
Occasional	L	M	H
Probable	L	M	H
Frequent	M	H	H

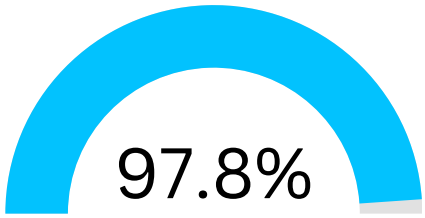
High: Funds or control of the contracts might be compromised directly. Data could be manipulated. We recommend fixing high issues with priority as they can lead to severe losses.

Medium: The impact of medium issues is less critical than high, but still probable with considerable damage. The protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions.

Low: Low issues impose a small risk on the project. Although the impact is not estimated to be significant, we recommend fixing them on a long-term horizon. Assets are not at risk: state handling, function incorrect as to spec, issues with comments.

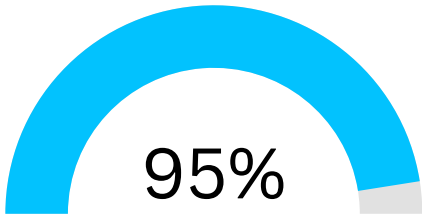
Quality Assurance: Informational and Optimization - Depending on the chain, performance issues can lead to slower execution or higher gas fees. For example, code style, clarity, syntax, versioning, off-chain monitoring (events etc.)

06-Audit Overview



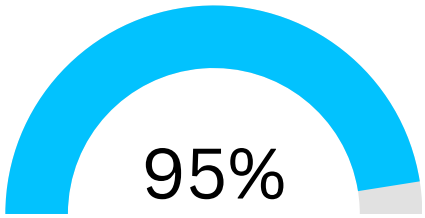
Security score

Security score is a numerical value generated based on the vulnerabilities in smart contracts. The score indicates the contract's security level and a higher score implies a lower risk of vulnerability.



Code quality

Code quality refers to adherence to standard practices, guidelines, and conventions when writing computer code. A high-quality codebase is easy to understand, maintain, and extend, while a low-quality codebase is hard to read and modify.



Documentation quality

Documentation quality refers to the accuracy, completeness, and clarity of the documentation accompanying the code. High-quality documentation helps auditors to understand business logic in code well, while low-quality documentation can lead to confusion and mistakes.

07-Findings

Finding: #1

Issue: Header difficulty target is not validated, allowing for very low PoW blocks to be submitted.

Severity: High

Where: contract/src/lib.rs#L295-L299

Impact: Blocks that have been mined with little work can be submitted, allowing an attacker to temporarily overturn the main chain with many malicious blocks. Consequently, blocks will have many confirmations in a short amount of time. Additionally, GC deletes valid main chain blocks, preventing new and valid blocks from being submitted, resulting in a bricked light client contract.

Description: Submitting Bitcoin block headers to the light client via submit_blocks verifies each header's proof of work to make sure a sufficiently large amount of work (hashing) has been done to mine the block. The PoW is verified by comparing the block header's hash with the target (difficulty) encoded in the header. If the hash is less than or equal to the target, the proof of work and, thus, the block header is considered valid.

However, the difficulty target, encoded in the header's bits field, can be set to a very high value, making it very easy to mine. This is caused by not checking if the target matches the expected difficulty at this block height.

As a consequence, an attacker can submit a large number of such malicious block headers, which will be temporarily added to the main chain (until the next legitimate block with a high PoW is mined and submitted -> reorg). This can be used to cause a block to have many (invalid) confirmations, thus messing with the transaction inclusion verification.

Moreover, by submitting many such malicious block to the main chain, triggering the GC mechanism would delete the legitimate block headers from the main chain. Worst case, all valid blocks are deleted, leaving only the malicious blocks in the main chain. Subsequent valid blocks cannot be submitted as there is no valid chain to build on and the previous blocks are deleted.

The following simple test case demonstrates how the difficulty can be set to a very large value, i.e. requiring very little work to mine a block:

```rust
#[test]
fn test_malicious_block_header_target() {
    let header = Header {
        version: 1,
        prev_block_hash: H256::from([0; 32]),
        merkle_root: H256::from([0; 32]),
        time: 0,
        bits: 0xff7fffff, // equals difficulty of 6646139186642954221875659365962219 52
        nonce: 0,
    };
    let mut block_hash = header.block_hash();

    // mine a "valid" block by adjusting the nonce to get a block hash that is less than the target
    let mut nonce = 0;
    let max_iter = 1_000;

    while U256::from_le_bytes(&block_hash.0) > header.target() && nonce < max_iter {
        nonce += 1;
        let mut header = header.clone();
        header.nonce = nonce;
        block_hash = header.block_hash();
    }

    print!("resulting block_hash: {}\n", &block_hash);
    print!("resulting nonce: {:?}\n", nonce);

    assert!(U256::from_le_bytes(&block_hash.0) <= header.target());
}
```

Recommendation: Consider validating the encoded difficulty target (bits) in the block header to ensure it matches the expected difficulty at this block height, based on the previous block's difficulty and timestamp (for example, see https://github.com/ethereum/btcrelay/blob/3cdf41a87750bfcc2ea62cb2c5d693fd71bdd47f/btcrelay.se#L113C13-L133).

Status: Resolved

Finding: #2

Issue: Intermediate Merkle tree nodes can be used as transaction hashes in the **verify_transaction_inclusion** function, resulting in a successful verification

Severity: Medium

Where: merkle-tools/src/lib.rs#L32-L51

Impact: Intermediate Merkle nodes can be used as transaction hashes in the **verify_transaction_inclusion** function, resulting in a successful verification and potentially causing severe issues in 3rd party contracts.

Description: Bitcoin transaction inclusion can be verified with the light client contract via the verify_transaction_inclusion function, by providing the following arguments:

```
pub struct ProofArgs {
    pub tx_id: H256,
    pub tx_block_blockhash: H256,
    pub tx_index: u64,
    pub merkle_proof: Vec<H256>,
    pub confirmations: u64,
}
```

Given the Merkle proof and the transaction hash (**tx_id**), the Merkle root is reconstructed and compared with the block header's Merkle root. If the Merkle root matches, the transaction is considered included in the block.

However, it is not guaranteed that **tx_id** is the hash of an actual transaction. Rather, it can be an intermediate node in the Merkle tree, i.e., the hash of the concatenation of two child nodes. The Merkle proof can then be adjusted to lead to a valid reconstructed Merkle root, even if this **tx_id** is not a leaf node (i.e., a transaction hash).

The following test demonstrates the issue:

```rust
#[test]
fn test_merkle_proof_verification_use_intermediate_node_as_tx_hash() {
    let tx_hashes = vec![
        decode_hex("18afbf37d136ff62644b231fcde72f1fb8edd04a798fb00cb06360da635da275"),
        decode_hex("30b19832a5f4b952e151de77d96139987492becc8b6e1e914c4103cfbb06c01e"),
        decode_hex("b94ed12902e35b29dd53cf25e665b4d0bc92f22adbc383ad90566584902b061d"),
        decode_hex("1920e5d8a10018dc65308bb4d1f11d30b5406c6499688443bfcd1ef364206b14"),
        decode_hex("048f3897c16bdc59ec1187aa080a4b4aa5ec1afcb4b776cf8b8a214b01990a7b"),
        decode_hex("266a660e2be5f2fdf41ae21d5a29c4db6270b2686dfe3902bd2dd3bca3626d7c"),
        decode_hex("17c3b888226ce70908303eaecb88ba02aa5ab858fade8576261b1203c6885528"),
        decode_hex("8a06d54b8b411e99b7e4d60c330b8cde4feb23d62edfc25047c4d837dfb5b253"),
    ];

    let calculated_merkle_root = merkle_root_calculator(&tx_hashes);
    let calculated_merkle_proof = merkle_proof_calculator(tx_hashes, 0);

    let computed_root_from_merkle_proof = compute_root_from_merkle_proof(
        decode_hex("dbf83868416633fe15eb98015a835f75574706e46bdf119f99e1d54dd252a800"), // intermediate node
        0,
        &vec![
            decode_hex("0b06c3a626979541511b397e6abdb503129dfcc4c3407ee994cedb2e5c498411") // merkle proof
        ]
    );
    assert_eq!(computed_root_from_merkle_proof, calculated_merkle_root);
}
```

There is no direct impact on the light client contract itself. However, this verification mechanism is supposed to be used by third-party actors (e.g., a NEAR contract). If they don't implement sufficient measures to prevent supplying an intermediate node, the verification can be bypassed, and depending on the third-party contract, this can lead to very severe consequences.

Thus, the light client's **verify_transaction_inclusion** function interface should already prevent this issue and not burden the caller.

Moreover, if **merkle_proof** is empty, **compute_root_from_merkle_proof** returns the transaction_hash itself, also bypassing the verification when the transaction hash is set to the Merkle root itself.

Recommendation: Instead of accepting the transaction hash **tx_id** as an argument, the function should accept the raw transaction data bytes and calculate the transaction hash in the contract. This way, it is very likely that the passed transaction data is indeed a transaction (a leaf node). An attacker would have to brute-force the pre-image of an intermediate node to bypass the verification, which is computationally very expensive and unlikely to succeed.

Status: Acknowledged.

Finding: #3

Issue: Off-By-One Error in Batch Size Limit for Block Submission

Severity: Low

Where: relayer/src/main.rs#L63-L73

Impact: The relayer can submit one more block than the intended limit (batch size), which could lead to exceeding the expected transaction size.

Description: The contract aims to limit the number of blocks a relayer can submit in a single batch to 15. However, the condition that breaks the loop (**i > batch_size**) allows one extra block to be submitted. The correct condition should be **i >= batch_size** to prevent the loop from processing more blocks than the allowed batch size.

Recommendations: Change the condition in the loop from if **i > batch_size** to if **i >= batch_size** to ensure the batch size limit is respected. This will correctly enforce the intended block submission cap.

Status: Resolved.

Finding: #4

Issue: The oldest main chain block **mainchain_initial_blockhash** is not updated in case of a sufficiently deep reorg.

Severity: Low

Where: contract/src/lib.rs#L420

Impact: mainchain_initial_blockhash will not be updated if a deep reorg occurs, leading to the contract pointing to an incorrect block hash as the oldest block. This will affect both **get_last_n_blocks_hashes** and **run_mainchain_gc**, resulting in incorrect behavior.

Description: mainchain_initial_blockhash stores the oldest available block hash of the main chain. Whenever the main chain is pruned (garbage collected), this block hash is updated to reflect the new oldest block hash.
However, if a new block is submitted that leads to a reorg, the **mainchain_initial_blockhash** is not updated in the reorg_chain function.

Under normal circumstances, this is not an issue, as the oldest main chain block is sufficiently "deep" into the chain, and such deep reorgs that would affect this block are very unlikely.
But if the light client contract has just been deployed and initiated with a very recent block as the "genesis" block, a subsequent reorg could affect **mainchain_initial_blockhash**. In such a scenario, the reorg would change the main chain's oldest block hash, while **mainchain_initial_blockhash** remains unchanged, pointing to a block that is no longer on the main chain.

Recommendation: Consider updating **mainchain_initial_blockhash** in the **reorg_chain** function if the reorg is deep enough.

Status: Acknowledged

Finding: #5

Issue: Unpausable **run_mainchain_gc** function

Severity: Low

Where: contract/src/lib.rs#L225

Impact: The permissionless **run_mainchain_gc** function cannot be paused.

Description: The main chain is periodically cleaned, i.e., old blocks get pruned, via the permissionless **run_mainchain_gc** function.

However, this function cannot be paused, which can lead to issues if the contract has been paused for emergency reasons to fix the main chain. Pruning should not take place during this time.

Recommendations: Consider also adding the pause mechanism to the **run_mainchain_gc** function.

Status: Resolved

Finding: #6

Issue: Relayer has hardcoded **confirmations** parameter to **0**, preventing sufficient verification of transactions

Severity: Low

Where: relayer/src/near_client.rs#L266

Impact: Transactions are not verified with a sufficient number of confirmations.

Description: The relayer can be run with **VERIFY_MODE=true** to verify whether the given transaction is included in the specified block by calling the verify_transaction_inclusion function in the light client contract.

To ensure that the block is sufficiently "deep", i.e., the black has enough confirmations by subsequent blocks, the contract accepts the confirmations parameter, which is used in the check in lib.rs#L201-L205.

However, the relayer has the confirmations parameter hardcoded to 0, which cannot be overridden by a CLI argument or environment variable.

Recommendation: Consider making the **confirmations** parameter configurable via CLI arguments or environment variables.

Status: Resolved

Finding: #7

Issue: Inconsistent transaction verification behavior if expected **confirmations** exceed **gc_threshold**.

Severity: Low

Where: contract/src/lib.rs#L190-L205

Impact: The **verify_transaction_inclusion** method for a valid transaction might fail or succeed depending on whether the target block was already removed by the garbage collection or not.

Description: In case the **args.confirmations** exceed the **self.gc_threshold** on transaction verification, it is not guaranteed that the target block is still available in storage. Therefore, the contract behavior is inconsistent and the garbage collection could be specifically invoked to grief users who rely on this edge case.

Recommendation: Add a check to the **verify_transaction_inclusion** method which consistently raises an error when the expected args.confirmations exceed the **self.gc_threshold**.

Status: Resolved

Finding: #8

Issue: Garbage collection can still be invoked while the contract is paused.

Severity: Low

Where: [contract/src/lib.rs#L225-L257]

Impact: The contract's state can be modified by anyone although it is paused by invoking the garbage collection.

Description: The permissionless **run_mainchain_gc** method is missing the **pause** attribute.

Recommendation:

```
+ #[pause]
  pub fn run_mainchain_gc(&mut self, batch_size: u64) {
```

Status: Resolved

Finding: #9

Issue: Invalid parameter description of **verify_transaction_inclusion** method.

Severity: Quality Assurance

Where: contract/src/lib.rs#L180

Impact: Misleading parameter description.

Description: The **verify_transaction_inclusion** method's **ProofArgs** parameter does not have a **tx_block_blockheight** field but a **tx_block_blockhash** field instead.

Recommendations:

```
- /// @param txBlockHeight block height at which transacton is supposedly included
+ /// @param txBlockHash block hash of the transaction which is supposedly included
```

Status: Resolved.

Finding: #10

Issue: The variable **amount_of_headers_we_store** is off by one.

Severity: Quality Assurance

Where: [contract/src/lib.rs#L236-L237]

Impact: The mainchain garbace collection removes one block header less than intended.

Description: The computation of the variable **amount_of_headers_we_store** is off by one.
Example: When **tip_blockheader == initial_blockheader**, then **amount_of_headers_we_store** would be **0** although actually **1** header is stored.

Recommendation:

```
    let amount_of_headers_we_store =
-       tip_blockheader.block_height - initial_blockheader.block_height;
+       tip_blockheader.block_height - initial_blockheader.block_height + 1;
```

Status: Resolved.

08 - Disclaimer

The smart contracts provided to AuditOne have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions). The ethical nature of the project is not guaranteed by a technical audit of the smart contract. Any owner-controlled functions should be carried out by the responsible owner. Before participating in the project, all investors/users are recommended to conduct due research.

The focus of our assessment was limited to the code parts associated with the items defined in the scope. We draw attention to the fact that due to inherent limitations in any software development process and product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which cannot be free from any errors or failures. These preconditions can impact the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure, which adds further inherent risks as we rely on correctly executing the included third-party technology stack itself. Report readers should also consider that over the life cycle of any software product, changes to the product itself or the environment in which it is operated can have an impact leading to operational behaviors other than initially determined in the business specification.

Contact

auditone.io

@auditone_team

hello@auditone.io

A trust layer of our
multi-stakeholder world.