hashlock.

Security Audit

L1X Consensus (L1 Blockchain)

February 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

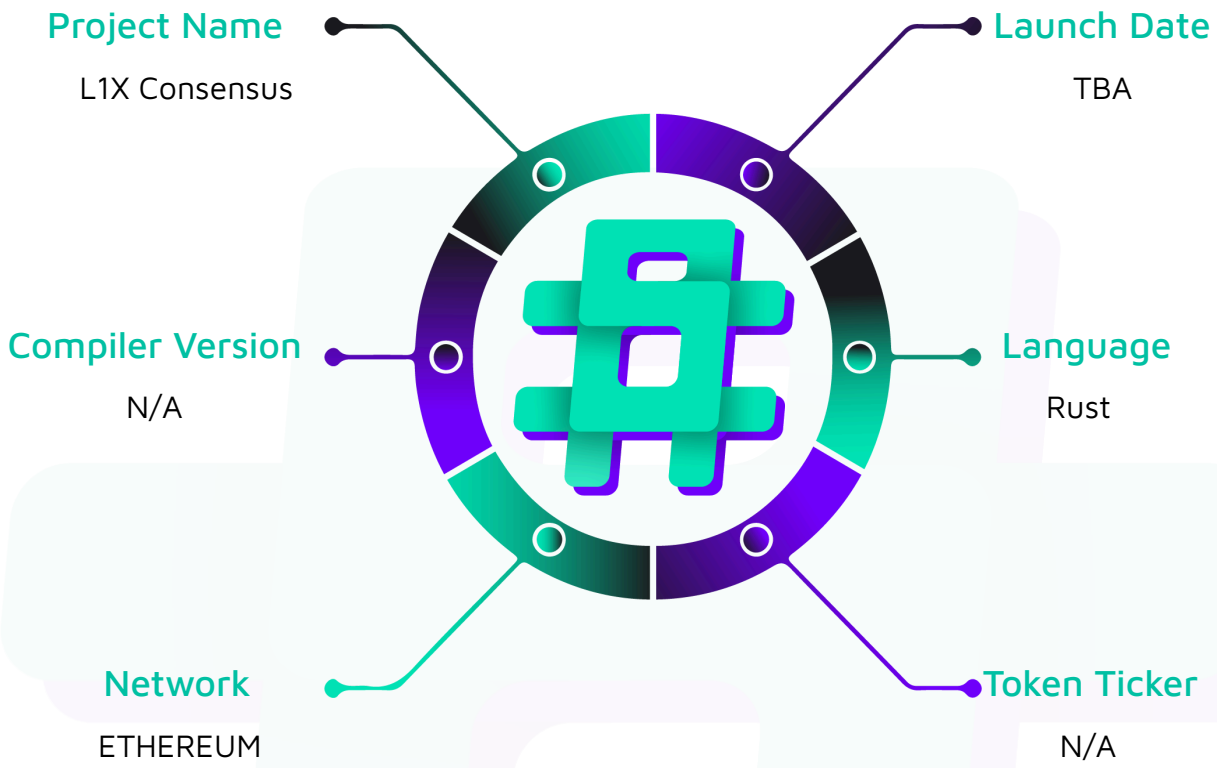
The L1X Consensus team partnered with Hashlock to conduct a security audit of their blockchain. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

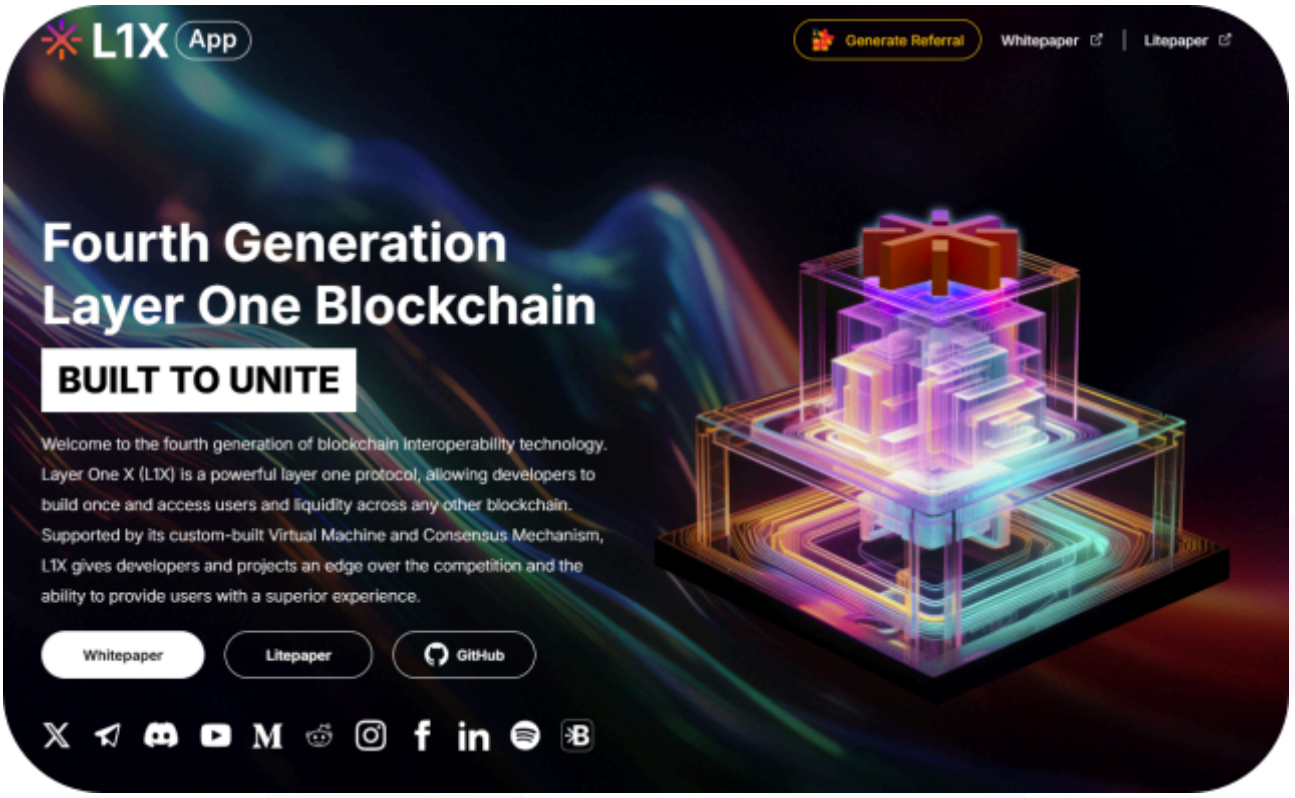
Layer One X (L1X) is a fourth-generation decentralized platform designed to make interoperability between blockchains seamless and efficient.
The L1X Foundation is dedicated to advancing this technology by fostering innovation and collaboration within the blockchain community. Through initiatives like the L1X NFT Launchpad and Marketplace, the foundation aims to provide developers and users with the tools and resources needed to build and engage with decentralized applications across multiple blockchain networks.

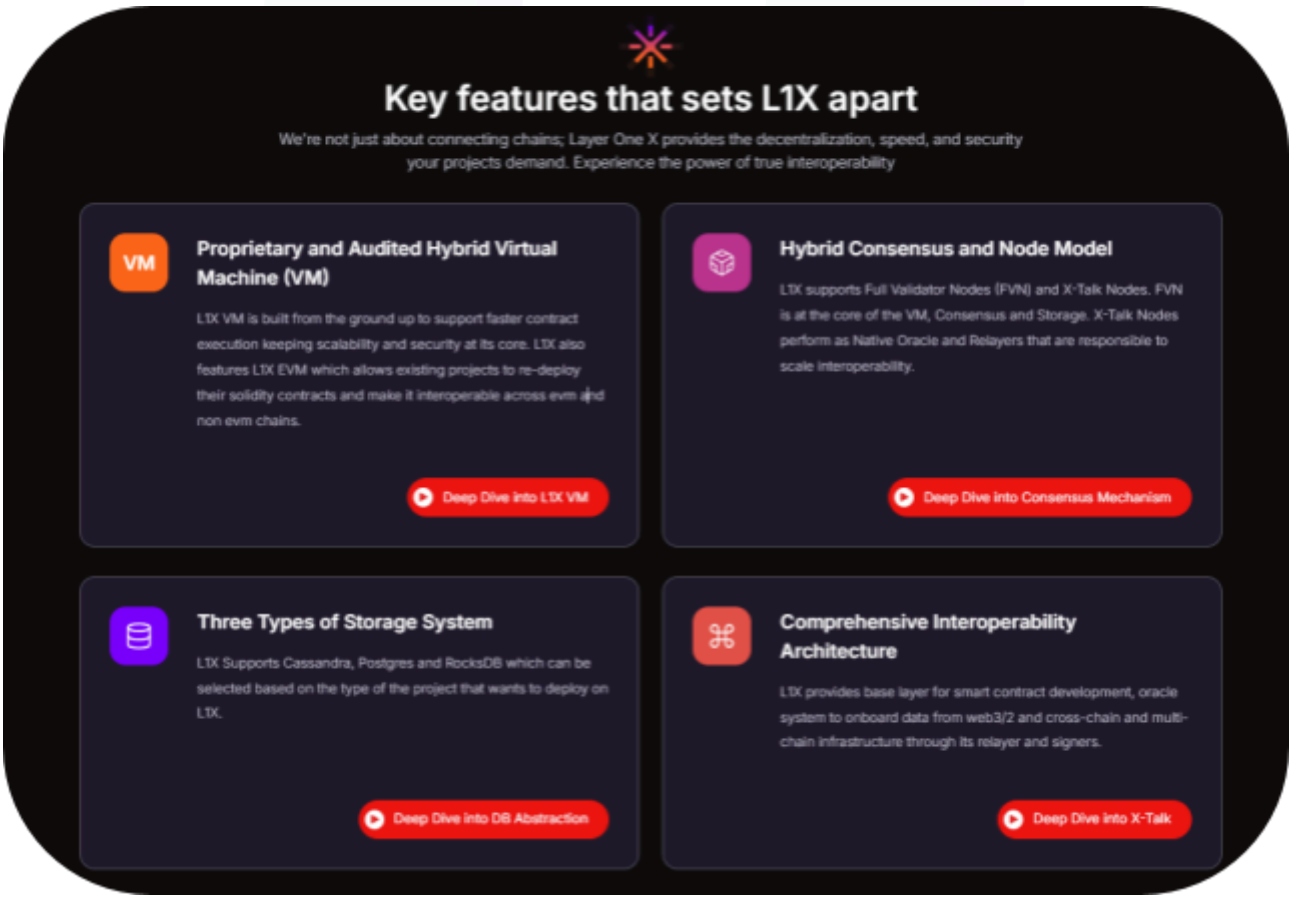
Project Name: L1X Consensus
Project Type: L1 Blockchain
Protocol Version: *2.4.4*
Website: https://l1xapp.com/l1x-foundation
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the blockchain code within the L1X Consensus project, the scope of work included a comprehensive review of the files listed in this report. We tested the codebase components to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**L1X Consensus Protocol Blockchain**
Platform	**Rust**
Audit Date	**February, 2025**
GitHub Repo	https://github.com/LayerOneX/l1x-core

Security Rating

After Hashlock's Audit, we found the Blockchain to be **"Secure"**. We initially identified some significant vulnerabilities that have since been addressed.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Blockchain Components Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 High severity vulnerabilities

8 Medium severity vulnerabilities

6 Low severity vulnerabilities

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Blockchain Components Functions

Claimed Behaviour	Actual Behaviour
Node - Allows users to: - Process and validate incoming transactions through an integrated mempool. - Propose, produce, and broadcast new blocks based on time or mempool conditions. - Handle diverse network events (e.g., votes, block proposals, node info) to maintain consensus. - Synchronize with bootnodes to update blockchain state, genesis data, and node health. - Monitor real-time events and adjust operational parameters dynamically.	**Protocol achieves this functionality.**
Genesis - Define and initialize foundational chain parameters (e.g., genesis time, initial amount, block version/number, block time, fee structure). - Generate and manage the initial set of genesis accounts and validators—including support for additional developer accounts in dev mode. - Establish the initial block header and block proposer state by storing them in the database. - Recover and verify account addresses using mnemonic phrases for secure key management. - Support multiple chain types (Mainnet, Testnet, Localnet) to tailor the genesis setup for diverse environments.	**Protocol achieves this functionality.**
Block_proposer - Allows nodes to:	**Protocol achieves this**

- Select the epoch's block proposer from eligible validators using a seed-based (modulo) pseudo-random algorithm. - Persist and update block proposer assignments in the database for cluster-wide consensus. - Verify whether a given node is the designated block proposer for the current epoch. - Retrieve block proposer details for block production and validator coordination.	**functionality.**
XScore - Enables nodes to: - Compute a composite "XScore" by integrating both StakeScore and KinScore metrics to assess node performance and participation eligibility. - Validate that the configured weightage for each scoring component is correctly balanced (i.e., sums to one) before executing calculations. - Dynamically combine stake-related data (e.g., balance, age, locking period) with performance indicators (e.g., uptime, response time) for a holistic evaluation. - Leverage integrated unit tests that simulate various scenarios—from edge cases to threshold validations—to ensure scoring accuracy and reliability.	**Protocol achieves this functionality.**
Consensus - Enables nodes to: - Process and Validate Network Data: Receive, verify, and aggregate diverse consensus messages—including node health reports, node info, block proposals, votes, and vote results—to	**Protocol achieves this functionality.**

maintain a consistent blockchain state. - Execute and Finalize Blocks: Validate incoming blocks, execute them, purge related transactions from the mempool, and broadcast finalized block data and consensus events. - Manage Pending Consensus Operations: Temporarily store pending blocks and votes, coordinate vote results, and finalize blocks once consensus conditions are met. - Interact with Validators and Proposers: Select, store, and verify validator sets and block proposers based on runtime configuration and real-time network conditions. - Respond to Peer Queries: Handle block query requests by returning signed block payloads along with finalization status and associated vote results.	
Validator - Enables nodes to: - Select and Rank Validators - Compute Dynamic XScore - Ensure Fair Validator Selection	**Protocol achieves this functionality.**
Mempool - Enables nodes to: - Efficiently handle incoming transactions by enforcing fee and rate limits, resolving conflicts, maintaining prioritized ordering, and purging expired transactions to ensure a consistent mempool state.	**Protocol achieves this functionality.**

- Aggregate validated transactions (including reward transactions), create new blocks via coordinated block proposals, execute blocks, and broadcast finalized block data alongside consensus events. - Calculate and generate reward transactions based on vote results and system configurations, ensuring that fee limits are respected and the rewards distributor maintains sufficient balance.	
Execute - Enables nodes to execute a broad spectrum of on-chain operations—including token transfers, transaction processing, smart contract execution, block creation, and fee enforcement—by integrating diverse execution components into a unified module that validates and updates the blockchain state securely while ensuring consistent consensus and economic sustainability.	**Protocol achieves this functionality.**
Fee - Enables nodes to: - Accurately track fee consumption and gas usage by maintaining a fee limit, updating consumed fees, and deducting burnt gas, ensuring that transaction execution stays within predefined cost constraints. - Dynamically determine available gas limits through a gas station mechanism that converts the remaining fee balance into usable gas units for executing transactions.	**Protocol achieves this functionality.**

- Provide clear, retrievable metrics on both fee and gas consumption, supporting consistent and secure management of execution costs during blockchain operations.	
L1x_node_health - Enables nodes to manage and monitor their health metrics by persistently storing, retrieving, and updating node health records, as well as computing a composite health score based on uptime, response time, transaction count, and block proposal count to assess overall performance and reliability.	**Protocol achieves this functionality.**
Vote - Enables nodes to persist, query, and manage vote data through a unified state interface that abstracts over multiple database backends (RocksDB, PostgreSQL, and Cassandra), ensuring consistent and reliable storage and retrieval of consensus votes across diverse environments.	**Protocol achieves this functionality.**
Vote_result - Enables nodes to aggregate validator votes, filter and verify their uniqueness, assess whether predefined vote thresholds are met, and generate cryptographically signed vote result payloads for secure consensus and broadcast across the network.	**Protocol achieves this functionality.**
P2p - Enables nodes to seamlessly participate in a decentralized peer-to-peer network by establishing and managing a comprehensive	**Protocol achieves this functionality.**

event loop that integrates peer discovery, gossip-based message broadcasting (for node info, transactions, block proposals, votes, and health data), subscription management, and request–response protocols, thereby supporting dynamic network configuration, robust consensus mechanisms, and real-time monitoring of network state.	
runtime_config - Enables nodes to dynamically refresh and retrieve a centralized, strongly typed runtime configuration—covering validator scoring parameters (xscore, stake score, and kin score), rewards, and node lists (org, whitelist, and blacklist)—from a TOML-based source stored in a thread-safe cache, ensuring consistent, validated system settings across the network.	**Protocol achieves this functionality.**
Server - Enables nodes to provide a comprehensive RPC interface for interacting with and querying the blockchain state. This module supports account management (creating, importing, and fetching account states), transaction submission and fee estimation (including handling various transaction types and smart contract operations), and retrieval of blockchain data such as block details (across multiple versions), transaction receipts, and events. Additionally, it facilitates access to higher-level state information—including chain state, node info, runtime configuration, staking data, and	**Protocol achieves this functionality.**

validator details—ensuring that nodes can both update and serve critical data for consensus, monitoring, and decentralized application interactions.	
Compile_time_config - Enables nodes to leverage a centralized configuration system that defines critical protocol parameters—such as block and protocol versions, scoring thresholds, validator limits, gas pricing, and designated system addresses—and to dynamically select environment-specific settings (mainnet, testnet, or devnet) while enforcing compile-time constraints to prevent incompatible feature combinations.	**Protocol achieves this functionality.**
Evm_interpreter - Enables nodes to execute EVM bytecode by leveraging an evaluation table to step through opcodes, manage control flow (including jumps, traps, and exits), and update the machine state securely, thereby ensuring precise and reliable smart contract execution.	**Protocol achieves this functionality.**
Evm_precompile - Enables nodes to execute native EVM precompile functions efficiently by mapping designated contract addresses to optimized implementations (such as ECRecover, SHA256, RIPEMD160, Identity, Modular Exponentiation, BN128 arithmetic, and Blake2F), thereby reducing gas costs and enhancing smart contract performance during execution.	**Protocol achieves this functionality.**

l1x_evm - Enables nodes to perform comprehensive EVM smart contract execution by abstracting the details of contract deployment and function calls (both state-changing and read-only). This trait provides a unified interface for deploying contracts with specified parameters (including nonce, gas limit, and deposit), executing contract functions while capturing exit reasons and gas consumption, and integrating state updates and event broadcasting seamlessly into the EVM runtime.	**Protocol achieves this functionality.**
l1x_htm - Allows nodes to perform real-time anomaly detection using Hierarchical Temporal Memory (HTM) by extracting and encoding relevant features from transactions and blocks, computing sparse distributed representations via spatial pooling, and leveraging temporal memory to capture dynamic patterns, ultimately generating an anomaly score that reflects deviations from expected behavior.	**Protocol achieves this functionality.**
State - Allows nodes to maintain and update an in-memory cache of blockchain state—including accounts, contracts, contract instances, contract states, events, staking pools, staking accounts, and transaction metadata—by providing efficient merge and commit operations that synchronize state changes with persistent storage, ensuring	**Protocol achieves this functionality.**

consistent and reliable state management across the network.	
system_contracts - Enables nodes to deploy and interact with essential system contracts by providing standardized initialization data, designated addresses, and call parameter structures for multisig, configuration, node registry, staking, and denylist contracts. This module centralizes contract bytecodes and environment-specific settings (for mainnet, testnet, or devnet) to ensure that protocol governance, configuration, and staking operations are executed consistently and securely across the network.	**Protocol achieves this functionality.**
Account - Enables nodes to manage user accounts by creating and initializing both regular and system accounts, securely transferring funds with robust nonce handling, and minting tokens to update balances, thereby ensuring consistent and accurate state updates across the blockchain ledger.	**Protocol achieves this functionality.**
Block - Allows nodes to create and validate blocks by assembling valid transactions, computing secure block hashes using SHA256, determining current epochs and slot timings, and generating new block headers and blocks based on the latest chain state, thereby ensuring synchronized and reliable block production throughout the network.	**Protocol achieves this functionality.**

Cluster - Enables nodes to manage network clusters by providing a unified interface for storing, querying, and updating cluster addresses across multiple database backends (RocksDB, PostgreSQL, and Cassandra), ensuring consistent and reliable configuration of cluster-wide settings for decentralized operations.	**Protocol achieves this functionality.**
Contract - Enables nodes to deploy and manage smart contracts by providing a unified interface for creating new contract instances, persisting them in a centralized contract state, and retrieving contract code, thereby ensuring consistent and secure lifecycle management of on-chain contracts.	**Protocol achieves this functionality.**
Contract_instance - Enables nodes to manage smart contract instances by abstracting database backend operations and providing a unified interface for storing, retrieving, updating, and deleting contract instance data and associated state key-value pairs, thereby ensuring consistent contract execution and state management across the network.	**Protocol achieves this functionality.**
Contract_instance_manager - Allows nodes to execute and manage smart contract operations seamlessly by unifying contract deployment, function calls (both state-changing and read-only), and native	**Protocol achieves this functionality.**

staking actions across multiple virtual machine environments (EVM, L1XVM, and L1XVM staking). This module abstracts the underlying execution details into dedicated sync APIs, integrates state updates, and broadcasts events, thereby ensuring that contract interactions and staking operations are performed reliably and consistently within the network.	
Db - Enables nodes to manage and abstract database connectivity and state persistence across multiple backends—such as PostgreSQL, Cassandra, and RocksDB—by initializing, re-initializing, and providing unified access to database connections and connection pools while caching system configuration, thereby ensuring robust, environment-specific data storage and retrieval for blockchain operations.	**Protocol achieves this functionality.**
Event - Enables nodes to manage and retrieve blockchain event data by abstracting over multiple database backends (RocksDB, PostgreSQL, Cassandra) into a unified event state interface, thereby supporting efficient event creation, filtering, and validity checks for reliable, event-driven operations.	**Protocol achieves this functionality.**
Mint - Nodes can identify minting transactions by verifying that both the sender and receiver match the designated mint master address, thereby distinguishing mint operations from	**Protocol achieves this functionality.**

regular token transfers.	
Node_info - Enables nodes to manage their identity and cluster membership by securely storing, verifying, updating, and removing node information, ensuring that each node's credentials and cluster affiliations are accurately maintained for robust network communication.	**Protocol achieves this functionality.**
Primitives - Allows nodes to operate on foundational blockchain primitives—such as addresses, balances, gas, nonces, timestamps, block numbers, cryptographic keys, and transaction identifiers—ensuring consistent, type-safe interactions across all protocol components.	**Protocol achieves this functionality.**
Staking - Enables nodes to manage staking operations by abstracting the persistence of staking pools and accounts across multiple database backends (RocksDB, PostgreSQL, and Cassandra). This module provides a unified interface for creating, updating, and querying staking pools; verifying the existence and retrieving details of individual staking accounts; inserting new staking accounts; and listing all stakers in a given pool, thereby ensuring consistent and reliable state management for staking functionalities within the blockchain protocol.	**Protocol achieves this functionality.**
Staking_manager	**Protocol achieves this functionality.**

- Allows nodes to manage staking operations by creating and maintaining staking pools, facilitating token transfers for staking and unstaking, updating staking account balances, and enforcing pool constraints (such as minimum and maximum balances) through coordinated interactions with account and staking state modules.	
Traits - Allows nodes to interact with and execute smart contracts by providing a unified interface for retrieving ownership details, managing contract storage, performing token transfers, invoking contract functions (including native operations like balance checks, transfers, staking, and unstaking), emitting events, and even executing remote contract calls, thereby ensuring seamless and consistent contract execution within the blockchain environment.	**Protocol achieves this functionality.**
Validate - Enables nodes to validate block integrity by verifying signatures, ensuring that block header fields (including block number, parent hash, cluster address, and timestamp) and nonce sequences match expected values, and by checking that reward transactions conform to configured staking parameters, thereby upholding the overall security and consistency of the blockchain.	**Protocol achieves this functionality.**
System	**Protocol achieves this**

- Allows nodes to interact with core system components—including access control (via types like AccessType), node health, block processing, node info management, chain state synchronization, transaction execution, VM result handling, and vote result verification—to enforce protocol rules and maintain overall network integrity.	**functionality.**

Code Quality

This audit scope involves the Blockchain of the L1X Consensus project, as outlined in the Audit Scope section. All the codebase mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the L1X Consensus project codebase

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this blockchain infrastructure are based on well-known industry standard open source projects.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.

Gas	Gas Optimisations, issues, and inefficiencies
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.

Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] add_and_broadcast_block - Logic function treating zero pending blocks as an error prevents adding the first block

Description

The `add_and_broadcast_block` function treats zero pending blocks as an error, preventing the addition of the first block.

Vulnerability Details

Take a look at the `add_and_broadcast_block` method's early "pending-block" check:

```
// check if we have any block present in the pending list
if self.pending_blocks.get_blocks().len() == 1 {
    ...
} else {
    let pending_blocks = self.pending_blocks.get_blocks().keys().collect::<Vec<_>>();
    // return error if a new block is proposed while a previous one is still pending
    return Err(anyhow!(
        "Last block is not executed yet, #{}, pending blocks: {:?}",
        block_number,
        pending_blocks
    ));
}
```

The code treats any `pending_blocks.len()` other than exactly 1 as an error—even if there are zero pending blocks.

If there are no pending blocks (`len() == 0`), we should normally accept the block as the first pending block—but this branch incorrectly returns an error.

If there are two or more pending blocks, it might also be legitimate logic to reject another new block, but the current code never distinguishes these cases properly.

Because of that `if == 1 { ... } else { return Err(...) }` structure, a node that tries to produce (or receive) a new block when `len() == 0` finds itself in the else branch, which raises the confusing "Last block is not executed yet" error.

Impact

Initial block addition fails outright when there are zero pending blocks. The code is always returning Err(...), so the node never correctly enqueues (and later finalizes) its own newly produced block

Recommendation

Allow a new block if there are no pending blocks, and only refuse it if there is already a pending block that is not yet finalized

Status

Resolved

Medium

[M-01] read_request - Unbounded read_to_end in request-response can exhaust memory with large inputs

Description

The `read_to_end` function does not have an upper bound on input size, leading to potential memory exhaustion if a large payload is received.

Vulnerability Details

The issue is the way the request-response protocol reads incoming data in:

```
// In the Codec implementation

async fn read_request<T>(… socket: &mut T, …) -> io::Result<Self::Request>

where

    T: AsyncRead + Unpin + Send,

{

    let mut response = Vec::new();

    // This reads the entire stream into an unbounded `Vec`.

    // An attacker can send arbitrarily large data, causing an OOM (DoS).

    socket.read_to_end(&mut response).await?;

    deserialize(&response)

}
```

Impact

Because the code calls `socket.read_to_end(&mut response)` without imposing any maximum limit, a peer could send arbitrarily large data, forcing the node to allocate unbounded memory (leading to potential memory exhaustion).

Recommendation

Impose an upper bound on the size of the request/response before reading it into memory. The simplest remedy is to replace read_to_end with a bounded read, for example: Use a library or utility function that limits how many bytes are read, e.g., `read_exact` into a fixed-size buffer, or `tokio_util::io::ReaderStream` plus manual checks. In short, we need to ensure we have an agreed-upon maximum payload size (and enforce it in the I/O code) to prevent memory-exhaustion attacks.

While the existing mitigations reduce the severity, implementing a size limit would be a very good practice because while the transport has a timeout, there's still no explicit size limit on the message, the unbounded read could still allow a peer to send an extremely large message.

Status

Resolved

[M-02] QueryNodeStatus - Trusts peer_id in payload instead of actual sender

Description

`QueryNodeStatus` trusts `peer_id` in the payload instead of the actual sender, enabling ID spoofing.

Vulnerability Details

The `QueryStatusRequest` flow shows that when we receive a node-status response we are trusting the `peer_id` string returned in the response payload rather than the actual remote peer that sent the response. This allows a node to spoof another peer's ID in the `QueryNodeStatus` response, effectively impersonating other peers.

In the `request_response::Event::Message` handler:

```
request_response::Event::Message { peer: _, message } => match message {
    request_response::Message::Response { response, .. } => {
        match response {
```

```
            L1xResponse::QueryNodeStatus(peer_id, request_time) => {

                                                    if   let   Ok(current_timestamp)   =
util::generic::current_timestamp_in_millis() {

                    let response_time = current_timestamp - request_time;

                    self.event_sender.send(Event::PingResult {

                         peer_id: peer_id,         // <--- BUG: from the payload, not the
real peer

                        is_success: true,

                        rtt: response_time as u64

                    }).await.unwrap_or_else(|_| {

                        error!("Failed to send PingResult event");

                    });

                } else {

                    error!("Failed to get current timestamp in L1xResponse")

                }

            }

            ...

        }

    }

    ...

}
```

Notice that in the pattern `Message::Response { response, .. } => ...` we do not use the actual peer that libp2p tells us about (the real remote who sent the message). Instead, we destructure the response—and in the case of `L1xResponse::QueryNodeStatus`, we trust the `peer_id` string inside that response.

But the remote node itself puts `local_peer_id.to_string()` into the response:

```
L1xRequest::QueryNodeStatus(request_time) => {
```

```
    let local_peer_id = self.swarm.local_peer_id().clone();

           let   response   =     L1xResponse::QueryNodeStatus(local_peer_id.to_string(),
request_time);

                       self.swarm.behaviour_mut().request_response.send_response(channel,
response).unwrap_or_else(|e| {

        error!("Failed to send query status response: {:?}", e);

    });

}
```

Impact

When we accept `peer_id` from the response at face value, any malicious responder could send back "fake" IDs. the code then emits an `Event::PingResult` that logs or records "successful" pings to the wrong peer. A node could spoof ping responses, manipulate node health metrics, and affect monitoring data.

Status

Resolved

[M-03] updated_state#un_stake - Incorrect checking when handling stake/unstake operations

Description

Currently, there are several checks implemented when performing stake and unstake operations. The problem is that if a staker has a stake left in the system, there are no checks to ensure that this amount is greater than the required min stake allowing a validator to still be able to participate in the consensus.

Vulnerability Details

Let's take a look at the current implementation of the `un_stake()` functionality in the `updated_state`:

```
      staking_account.balance = staking_account.balance.checked_sub(amount)
```

```
                    .ok_or(anyhow!("Unstake: no enough fund to unstake, account: {},
staked amount: {}, amount: {}",
                            hex::encode(account_address),          staking_account.balance,
amount))?;


            self.update_staking_account(&staking_account)?;
            self.update_staking_pool_block_number(pool_address, block_number)?;
```

As you can see here, the function just updates the balance of the staking account and does not have any checking on its min stake. So if there is a situation where a user can withdraw some of his stake and his stake at the same time falls below the minimum required stake, there is no penalization or the total withdrawal of the staker performed allowing him to continue participation in the protocol's consensus.

Impact

Validators can violate minimum stake requirements without being slashed.

Recommendation

Consider checking whether the left stake is lower than the minimum stake and remove the stake fully in this case. Another solution that can be used is the introduction of some sort of slashing functionality.

Status

Resolved

[M-04] staking_manager#stake - Some validators can face DoS when trying to unstake due to the min pool balance condition

Description

At the moment, validators can stake a certain amount of tokens that is greater than the minimum stake and have an opportunity to actively participate in the protocol's consensus voting for blocks and earning rewards in parallel. However, there is a certain condition that prevents them from being able to freely exit the system anytime.

Vulnerability Details

Consider the `un_stake()` functionality in the `staking_manager`:

```
if value_after_un_stake.le(&min_pool_balance) {

                    let msg =

                          format!("Pool balance cannot be less than min pool balance -
{}", min_pool_balance);

                    log::error!("{}", msg);

                    return Err(anyhow!(msg))

             }
```

First, it calculates the `value_after_stake` by subtracting the amount that is currently being unstaked. Then it compares this amount to the `min_pool_balance`. This way, some of the validators can be blocked from withdrawing their stake if it falls below the `min_pool_balance`. So in a scenario where we have 3 validators and 2 of them have already unstaked, the third unstake operation will face DoS as that's highly likely that the min pool balance will not be satisfied in such a case.

Impact

Validators' funds can be frozen as there is no free exit from the system in case of stake is lower than the `min_pool_balance`.

Recommendation

Consider removing such a check for the `min_pool_balance` when implementing the `un_stake` function and adding it only to the stake functionality.

Status

Resolved

[M-05] validator_manager#select_validators_for_epoch() - Inconsistency when calculating the current epoch parameter

Description

The current implementation of the `select_validators_for_epoch()` function uses the predetermined algorithm for validators selection for a certain epoch. This includes checking the staking balance, calculating the `xscore` parameter and many other checks finally resulting in a list of validators being selected. The issue lies in the fact that the current epoch is calculated using the previous `block_number` rather than the `block_number + 1` so that the usage of a block number is about to be created.

Vulnerability Details

The function that's used to calculate the current epoch is the following:

```
pub fn calculate_current_epoch(&self, block_number: BlockNumber) -> Result<Epoch, Error>
{
            if block_number == 0 {
                  return Ok(0);
            }
            let current_epoch = block_number / SLOTS_PER_EPOCH;
            Ok(current_epoch as Epoch)
      }
```

The problem is that the block number provided in the `select_validators_for_epoch()` for this function is a block number from the last header (from the last block):

```
let current_epoch = block_manager.calculate_current_epoch(last_block_header.block_number)?;
```

This is incorrect as the eligible validators are being selected for the new block number. In comparison, consider the `propose_block()` function in the mempool:

```
let current_epoch = block_manager.calculate_current_epoch(last_block_header.block_number)?;
```

```
let current_epoch = block_manager.calculate_current_epoch(block_number)?;
```

Impact

The block number is incorrect therefore the values for validators (such as node health) can be incorrect.

Recommendation

Consider implementing the same calculation of the current epoch in all of the functions to avoid inconsistency and unexpected errors.

Status

Resolved

[M-06] vote_result#vote_result() - Min votes number is not calculated consistently

Description

At the moment, the number of minimum votes required for the votes to pass differs from one function to another. This leads to a potential incorrect output when trying to use both `vote_result()` and `try_to_generate_vote_result()` in calculations.

Vulnerability Details

`vote_result()` has the following requirement regarding the `min_votes` parameter:

```
      let votes = all_votes_hashmap;

                  // Calculate the minimum number of votes required (70% of validators)

                  // let min_votes = (validators.len() as f64 * 0.7) as usize;

                  let min_votes = 1;


                  // Check if the number of votes is less than the required minimum

                  // We assume 30% of validators will be unresponsive or not available

                  if votes.len() < min_votes {
```

```
                        return Ok(None) //Err(anyhow!("Number of votes is less than
70% of validators"));

                    }
```

So the function assumes that 30% of validators may be unavailable and therefore accepts the result if there are 70% of the current validators who voted (currently the minimum number of votes is assigned to 1 but it'll supposedly be changed later).

In comparison, `try_to_generate_vote_result()` has a different implementation:

```
// If 50% of the vote is in favour of the block, the block is accepted

                    // let vote_passed = (total_favoured_stake / pool_balance as f64) >
0.5;

                    let min_votes = (validators.len() as f64 * 0.6).round() as usize;
```

It can be seen that the function requires not 70% but already 60% of the validators to vote (and the comment says even 50%).

Impact

`min_votes` requirement is crucial for the system's security as it ensures that the consensus mechanism works as expected and such inconsistencies result in a system's requirements being violated.

Recommendation

Consider unifying the way how `min_votes` are calculated.

Status

Resolved

[M-07] vote_result#vote_result() - The current voting mechanism is subjected to a centralization attack

Description

Currently, the mechanism to determine if the vote is passed and the block is accepted is flawed as it assumes that the `total_favoured_stake` parameter is supposed to reflect

whether exactly 50% of the votes are in favor of the block which is not correct as validators with huge stake may be able to impact the system even if the majority of the voters are against it.

Vulnerability Details

Let's take a look at how it's currently determined whether the vote has passed or not:

```
// If 50% of the vote is in favour of the block, the block is accepted

                    // let vote_passed = (total_favoured_stake / pool_balance as f64) >
0.5;

                    let vote_passed = total_favoured_stake > 0.0;
```

As you can see, the `total_favoured_stake` is the total stake of all the validators who voted for the block and the number is currently supposed to be greater than 50% of the total `pool_balance`. This gives serious leverage to some participants in the system as they can significantly impact the result of the vote just by staking more. So, in a scenario where there are 70% voted and only 1% is in favor of the block, the result can be to accept the block if the 1% holds > 50% of the pool balance.

Impact

The system can be compromised by the validators with the biggest stakes.

Recommendation

Consider changing the current mechanism of how the passed vote is determined.

Status

Resolved

[M-08] Cassandra vote table mismatch - Schema inconsistency

Definition

The code reads from the Cassandra vote table expecting 8 columns, but the schema defines only 7.

Vulnerability Details

In the `Cassandra` implementation (`state_cas.rs`), the code attempts to read eight columns—including an epoch—from the voting table despite the schema defining only seven columns. the code uses:

```
row.into_typed::<(i64, BlockHash, Address, Address, Vec<u8>, Vec<u8>, bool, i64)>()?;
```

The actual table has only:

```
block_number bigint,
block_hash   blob,
cluster_address blob,
validator_address blob,
signature   blob,
verifying_key blob,
vote        boolean
```

Table Definition (Cassandra):

```
CREATE TABLE IF NOT EXISTS vote (
    block_number Bigint,
    block_hash blob,
    cluster_address blob,
    validator_address blob,
    signature blob,
    verifying_key blob,
```

```
    vote boolean,
    PRIMARY KEY (block_hash, validator_address)
);
```

That is 7 columns in total.

Code Reading 8 Columns In `state_cas.rs`, the function `load_all_votes` does:

```
let (
    block_number,
    block_hash,
    cluster_address,
    validator_address,
    signature,
    verifying_key,
    vote,
    epoch,
) = row.into_typed::<(i64, BlockHash, Address, Address, Vec<u8>, Vec<u8>, bool, i64)>()?;
```

This tuple destructuring corresponds to 8 columns:

```
block_number
block_hash
cluster_address
validator_address
signature
verifying_key
vote
epoch (not in the schema)
```

Hence the extra epoch column doesn't exist in the vote table definition for Cassandra.

Recommendation

Consider either removing the epoch or implementing it if it's necessary

Status

Resolved

Low

[L-01] validator_manager- Validators are not discarded if they have less than the minimum balance deviating from the spec

Description

The specification of the protocol states that validators will be discarded during the selection process if their stake is lower than the minimum required. Currently, such a requirement is not enforced and they're only discarded if their stake is equal to 0:

```
if rt_config.org_nodes.contains(node_address) || info.staked_balance == 0  {
                        continue;
                    }
```

Recommendation

Consider changing the check above on the following one:

```
if rt_config.org_nodes.contains(node_address) || info.staked_balance < min_stake  {
                        continue;
                    }
```

Status

Resolved

[L-02] validator_manager - min_balance parameter is taken as the min_pool_balance but this may not work in certain cases

Description

When assigning the params for the `stake_score_calculator`, the function takes the `min_pool_balance` and `max_pool_balance` to use them for the calculation of the stake score:

```
stake_score_calculator.min_balance = node_pool_info.min_pool_balance;

stake_score_calculator.max_balance = node_pool_info.max_pool_balance;
```

However, this may not work in the case if the `min_pool_balance` is lower than the `min_stake` value for example. In this situation, it's correct to use the `min_stake` parameter rather than the `min_pool_balance`.

Recommendation

Use `min_stake` and `max_stake` values instead of the `min_pool_balance` and `max_pool_balance` correspondingly.

Status

Acknowledged

[L-03] state_hash - No verification of state_hash in block headers

Definition

The system does not verify `state_hash` values in block headers, potentially allowing invalid state transitions.

Vulnerability Details

Validating the `state_hash` (or "post-state root") in the `BlockHeader` against a computed/expected state root is never performed. Although the `BlockHeader` structure includes a `state_hash` field, nowhere do we actually compute and compare that hash to ensure the block's state root matches what the block header claims it should be.

In most blockchains, a "state root" or "post-state hash" is crucial because it is how other nodes verify that the state resulting from applying all transactions matches exactly the state the block producer claims. Without verifying `state_hash`

We check signatures, parent hash, number/ordering, timestamps, and so on. But we never validate that `block_header.state_hash` matches an actually computed new state root.

`validate_expected_block_fields`:

```
fn validate_expected_block_fields(
    last_block_header: &BlockHeader,
    new_block: &Block,
) -> Result<(), Error> {
    // ...
    // Validate block_number, parent_hash, cluster_address, timestamp, num_transactions
    // ...
    Ok(())
}
```

This function checks all the fields except for the `state_hash`.

`validate_expected_block_hash`:

```
fn validate_expected_block_hash(block_payload: &BlockPayload) -> Result<(), Error> {
    let expected_block_hash = compute_block_hash(block_payload)?;
    if expected_block_hash != block_payload.block.block_header.block_hash {
        return Err(anyhow!("Incorrect block hash: ..."));
    }
    Ok(())
}
```

Even though we validate the block hash, that hash is (by the project's design) only computed over certain fields, and does not confirm that state_hash is computed accurately from transaction execution.

`validate_block_header_and_transactions`:

```
async fn validate_block_header_and_transactions<'a>(
    block_payload: &BlockPayload,
    db_pool_conn: &'a DbTxConn<'a>,
    cluster_address: Address,
    ...
) -> Result<(), Error> {
    let last_block_header = block_state.block_head_header(cluster_address).await?;
    // ... calls validate_expected_block_fields & validate_expected_block_hash
    // Then checks each transaction ...
}
```

Notice we do not see any `check_state_hash`(...) logic that ties the declared header's `state_hash` to a computed root of applying all of the block's transactions to prior state.

Impact

Without `state_hash` validation, nodes cannot cryptographically verify state transitions

Note

L1X team informed Hashlock this will be added in the near future.

Status

Acknowledged

[L-04] node_private; node_keypair - Unreachable fallback and Dead code path

Definition

Certain code paths are unreachable, making parts of the system redundant and harder to maintain.

Vulnerability Details

If we looked at the two places where `node_priv_key` is handled:
First, the code force requires `node_priv_key` to be Some and panics otherwise:

```
let node_private_key = match node_priv_key.clone() {
    Some(node_private_key) => node_private_key,
    None => {
        println!("An error occurred, node_private_key not found");
        panic!()
    },
};
```

Later, the code tries to allow an optional fallback (`map_or_else`) for `node_priv_key`, where it would generate a new P2P key if `node_priv_key` were None:

```
let node_keypair = node_priv_key.as_ref().map_or_else(
    || Keypair::generate_secp256k1(),  // fallback if None
    |private_key| {
        // use provided node_priv_key
        let mut keypair_bytes =
            hex::decode(private_key).expect("Failed to hex decode privkey");
        let secret_key =
            libp2p::identity::secp256k1::SecretKey::try_from_bytes(&mut keypair_bytes)
                .expect("Failed to parse keypair");
```

```
        let secp_keypair = libp2p::identity::secp256k1::Keypair::from(secret_key);
        let keypair: libp2p::identity::Keypair =
            secp_keypair.try_into().expect("Failed to convert to Keypair");
        keypair
    },
);
```

Because the code *already* panics whenever `node_priv_key` is None, the "fallback" path of `map_or_else` (`Keypair::generate_secp256k1()`) is actually unreachable. In other words:

The first snippet forces `node_priv_key` to be Some, otherwise it just does `panic!()`.

The second snippet is written *as though* it can handle the None case by generating a P2P identity key on the fly—but that branch is never taken due to the prior panic.

The "fallback" is dead code

Impact

The fallback code to generate a new keypair in the `map_or_else()` call is unreachable due to the earlier panic

Recommendation

Decide if we truly need to force a single key or allow an optional fallback:

If the node absolutely must have a single "master" key for both networking and consensus, then remove the `map_or_else(|| Keypair::generate_secp256k1())` path altogether, as it is dead code. Otherwise, remove the initial panic and let map_or_else generate the key properly if `node_priv_key` is not provided.

Status

Resolved

[L-05] apply_db_dump - Restore code incorrectly treats archive_nodes[0] as an S3 URL

Definition

The restore function mistakenly assumes `archive_nodes[0]` is an S3 URL, preventing proper database dumps from being downloaded.

Vulnerability Details

Inside `apply_db_dump`, the function always calls `download_file(&config.archive_nodes[0], file).await?`; expecting an S3 endpoint (an HTTP base URL) in config.archive_nodes[0]. In reality, archive_nodes is used elsewhere in the code as a list of *node endpoints* (multiaddresses or gRPC addresses), not S3 HTTP URLs. This causes a clear mismatch/bug in the logic: the code incorrectly treats `archive_nodes[0]` as though it were an S3 URL, which will either panic or fail to download the dump from an actual S3 bucket.
the code expects:

```
// apply_db_dump calls:

for file in s3_files {

    // The function signature is: download_file(base_url: &str, file_name: &str)

    // "base_url" is effectively expected to be "http://some-s3-bucket/..."

    // or something similar.

    download_file(&config.archive_nodes[0], file).await?;

}
```

The helper `download_file` function then does:

```
let url = format!("{}{}", base_url, file_name);

let response = client.get(url).send().await?;
```

Impact

Because the code simply does `&config.archive_nodes[0]` and treats it as though it were an S3 bucket URL, there is a logic error.

Recommendation

Proper fix would be:

```
// Separate the concerns into different config fields
struct Config {
   // For node synchronization
   archive_nodes: Vec<String>,  // multiaddresses
   // For S3 database dumps
   archive_s3_endpoint: String,  // HTTP URL
}
```

This would prevent the misuse of `archive_nodes` for S3 operations and make the code's intent clearer.

Status

Resolved

[L-06] `selected_next` - selected_next flag never updates correctly

Definition

The `selected_next` flag is never set to true, causing `is_selected_next` to always return false.

Vulnerability Details

It's usual to store each new proposer with `selected_next = false`, but the code that flips it to true is never called. As a result, `is_selected_next` is always false because there is at least one row for that address with `selected_next = false`.

`The selected_next` is set to false by default, in the Cassandra (and similarly in RocksDB) backend, when you call `store_block_proposer`, the new row is stored with `selected_next = false` :

```
session.query(
 "INSERT INTO block_proposer (cluster_address, epoch, address, selected_next) VALUES (?, ?, ?, ?);",
   (&cluster_address, &epoch, &address, &false),
  )
  .await?;
```

To `selected_next` is supposed to become true, the only place `selected_next` gets changed to true is in the `update_selected_next` method:

```
   async fn update_selected_next(
        &self,
        selector_address: &Address,
        cluster_address: &Address,
    ) -> Result<(), Error> {
                                            if let Some(epoch_numbers) = self.load_selectors_block_epochs(selector_address).await? {
            for epoch in epoch_numbers {
                // ...
                session.query(
                       "UPDATE block_proposer SET selected_next = ? WHERE address = ? AND cluster_address = ? AND epoch = ?;",
                    (&true, selector_address, cluster_address, &epoch),
                )
                .await?;
            }
```

```
        }

        Ok(())

    }
```

Notice this only runs if you pass non-None values for both `selector_address` and `cluster_address` to `store_block_proposers`. But the manager always calls with None

In `BlockProposerManager`:

```
   block_proposer_state

        .store_block_proposers(

            &cluster_block_proposers,

            None,  // <--- always None

            None,  // <--- always None

        )

        .await?;
```

Because both parameters are None, the `update_selected_next` logic is never triggered.

Effect: `selected_next` remains false. Since every row is inserted with `selected_next = false` and never updated to true, any check that depends on `selected_next` (like `is_selected_next`) always finds at least one row having `selected_next = false` and returns false.

```
    async fn is_selected_next(&self, address: &Address) -> Result<bool, Error> {

        let (count,) = self

            .session

            .query(

                 "SELECT COUNT(*) FROM block_proposer WHERE address = ? AND selected_next
= false ALLOW FILTERING;",

                (address,),

            )

            .await?
```

```
            .single_row()?
            .into_typed::<(i64,)>()?;


         // If there is *any* row for this address with selected_next=false, this returns
false:
        Ok(!(count > 0))
    }
```

Impact

Since there is always at least one entry with selected_next = false, is_selected_next will forever be false. The logic is effectively inverted: the code as it stands never ends up marking the active proposer with selected_next = true.

Recommendation

Either pass the "selector address" (meaning the block proposer that we want to mark as the next proposer) and the `cluster_address` when calling `store_block_proposers`, so that `update_selected_next` is invoked:

```
   block_proposer_state.store_block_proposers(
        &cluster_block_proposers,
        Some(selected_proposer_address),
        Some(block_header.cluster_address),
    ).await?;
```

Status

Resolved

Centralisation

The L1X Consensus project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the L1X Consensus project seems to have a sound and well-tested codebase, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these Blockchain components in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the Blockchain components source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this blockchain.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au