

Movement Node

Audit Report

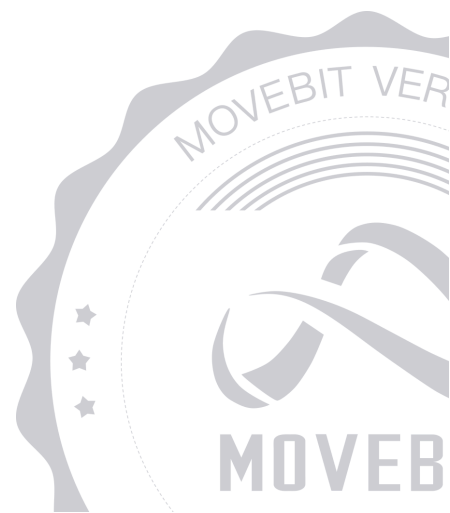


contact@bitslab.xyz



https://twitter.com/movebit_

Fri Nov 01 2024



Movement Node Audit Report

1 Executive Summary

1.1 Project Information

Description	The Movement Network is a Move-based L2 on Ethereum.
Type	L2
Auditors	MoveBit
Timeline	Fri Jul 26 2024 - Fri Nov 01 2024
Languages	Rust
Platform	Ethereum
Methods	Dependency Check, Fuzzing, Static Analysis, Manual Review
Source Code	https://github.com/movementlabsxyz/movement https://github.com/movementlabsxyz/aptos-core
Commits	https://github.com/movementlabsxyz/movement: 030fe011da8bd7eca75fa0a37197205767bef7ec 388ada0b2d10318348aac4d55bd50f3b02e397cd https://github.com/movementlabsxyz/aptos-core: 7d117d7eb8052d4bc580a23094bed4e1250e1113 7a0e210fe29e2c81c378568eba2b467a7ef6a56e

1.2 Files in Scope

The following are the directories of the original reviewed files.

Directory
https://github.com/movementlabsxyz/movement/types
https://github.com/movementlabsxyz/movement/util
https://github.com/movementlabsxyz/movement/protocol-units
https://github.com/movementlabsxyz/movement/networks
https://github.com/movementlabsxyz/aptos-core/protocol-units
https://github.com/movementlabsxyz/aptos-core/networks

1.3 Issue Statistic

Item	Count	Fixed	Partially Fixed	Acknowledged
Total	23	8	2	13
Informational	11	4	0	7
Minor	2	0	0	2
Medium	4	0	0	4
Major	3	3	0	0
Critical	3	1	2	0

1.4 MoveBit Audit Breakdown

MoveBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Integer overflow/underflow
- Infinite Loop
- Infinite Recursion
- Race Condition
- Traditional Web Vulnerabilities
- Memory Exhaustion Attack
- Disk Space Exhaustion Attack
- Side-channel Attack
- Denial of Service
- Replay Attacks
- Double-spending Attack
- Eclipse Attack
- Sybil Attack
- Eavesdropping Attack
- Business Logic Issues
- Contract Virtual Machine Vulnerabilities
- Coding Style Issues

1.5 Methodology

Our security team adopted "**Dependency Check**", "**Automated Static Code Analysis**", "**Fuzz Testing**", and "**Manual Review**" to conduct a comprehensive security test on the code in a manner closest to real attacks. The main entry points and scope of the security testing are specified in the "**Files in Scope**", which can be expanded beyond the scope according to actual testing needs. The main types of this security audit include:

(1) Dependency Check

A comprehensive check of the software's dependency libraries was conducted to ensure all external libraries and frameworks are up-to-date and free of known security vulnerabilities.

(2) Automated Static Code Analysis

Static code analysis tools were used to find common programming errors, potential security vulnerabilities, and code patterns that do not conform to best practices.

(3) Fuzz Testing

A large amount of randomly generated data was inputted into the software to try and trigger potential errors and exceptional paths.

(4) Manual Review

The scope of the code is explained in section 1.2.

(5) Audit Process

- Clarify the scope, objectives, and key requirements of the audit.
- Collect related materials such as software documentation, architecture diagrams, and lists of dependency libraries to provide background information for the audit.
- Use automated tools to generate a list of the software's dependency libraries and employ professional tools to scan these libraries for security vulnerabilities, identifying outdated or known vulnerable dependencies.
- Select and configure automated static analysis tools suitable for the project, perform automated scans to identify security vulnerabilities, non-standard coding, and potential risk points in the code. Evaluate the scanning results to determine which findings require further manual review.

- Design a series of fuzz testing cases aimed at testing the software's ability to handle exceptional data inputs. Analyze the issues found during the testing to determine the defects that need to be fixed.
- Based on the results of the preliminary automated analysis, develop a detailed code review plan, identifying the focus of the review. Experienced auditors perform line-by-line reviews of key components and sensitive functionalities in the code.
- If any issues arise during the audit process, communicate with the code owner in a timely manner. The code owners should actively cooperate (this may include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- Necessary information during the audit process will be well documented in a timely manner for both the audit team and the code owner.

2 Summary

This report has been commissioned by [Movement](#) with the objective of identifying any potential issues and vulnerabilities within the source code of the [Movement Node](#) repository, as well as in the repository dependencies that are not part of an officially recognized library. In this audit, we have employed the following techniques to identify potential vulnerabilities and security issues:

(1) Dependency Check

A comprehensive analysis of the software’s dependency libraries was conducted using the dependency analysis tool.

(2) Automated Static Code Analysis

The code quality was examined using a code scanner.

(3) Fuzz Testing

Some functions that was fuzz tested listed bellow:

- `JellyfishMerkleTree::batch_put_value_sets()` in MerkleTree module.
- `JellyfishMerkleTree::put_value_set_with_proof()` in MerkleTree module.
- `JellyfishMerkleTree::put_value_set()` in MerkleTree module.
- `SparseMerkleProof::verify()` in MerkleTree module.

(4) Manual Code Review

The primary focus of the manual code review was:

- [movementlabsxyz/movement](#)
- [movementlabsxyz/aptos-core](#)

During the audit, we identified 23 issues of varying severity, listed below.

ID	Title	Severity	Status

AVM-1	No Gas Gharges for Failed Transactions	Critical	Partially Fixed
LIB-1	The Transaction Sorting Algorithm In <code>RocksdbMempool</code> Can Lead To A Series Of Issues	Major	Fixed
LIB-2	Lack of Atomicity in Multiple Database Write Operations in mempool transaction handling methods	Informational	Fixed
MAI-1	Centralization Risk Due to High Trust Assumptions Required	Informational	Acknowledged
MOD-1	Lack of Garbage Collection Mechanism in Movement-Aptos-Core's Mempool	Major	Fixed
PAR-1	DoS Vulnerability Due to Sequencer Not Checking for Sequence Number Too New During Block Packaging	Critical	Partially Fixed
PAR-2	Use a Higher Security Level Hash Function	Informational	Fixed
ROC-1	High Time Complexity in <code>num_nodes</code> Function Leading to Potential DOS Vulnerability	Minor	Acknowledged
ROC-2	Incorrect Implementation of <code>get_rightmost_leaf</code> Leading to Inaccurate Node Retrieval	Minor	Acknowledged
ROC-3	Lack of Atomicity in Multiple Database Write Operations in RocksdbJmt handling methods	Informational	Acknowledged

ROC-4	Inefficient Implementation of <code>get_value_option</code> Leading to Potential DOS Vulnerability	Informational	Acknowledged
ROC-5	<code>db</code> Object Inside <code>RocksdbJmt</code> Should Have A read-write Lock Added	Informational	Acknowledged
SEQ-1	Lack of Garbage Collection Mechanism in Mempool	Major	Fixed
SER-1	Enforce HTTPS-Only Access Full Node API to Mitigate Security Risks	Informational	Fixed
TPI-1	Invalid <code>sequence_number</code> Check for Mempool in aptos-core	Informational	Fixed
TRE-1	Library Function <code>batch_put_value_sets</code> Throws Panic, but This Is Not Mentioned in the Documentation	Medium	Acknowledged
TRE-2	Library Function <code>batch_put_value_sets</code> Throws Panic, but This Is Not Mentioned in the Documentation	Medium	Acknowledged
TRE-3	Library Function <code>batch_put_value_sets</code> Throws Panic, but This Is Not Mentioned in the Documentation	Medium	Acknowledged
TRE-4	Common Prefix Nodes Increasing Tree Height, Leading to Higher Computational and Storage Resource Consumption	Informational	Acknowledged
TRE-5	Lack of Version Parameter Validation in <code>put_value_sets</code>	Informational	Acknowledged

	Function		
TRE-6	Use of SHA256 in JMT Implementation Not Suitable for ZK Proofs	Informational	Acknowledged
LIB1-1	Multithreading Issue Leads to Transaction Replay Vulnerability	Critical	Fixed
LIB1-2	Code Style Recommendations	Medium	Acknowledged

3 Participant Process

Here are the relevant actors with their respective abilities within the [Movement Node](#) repository :

- **User:** A person who conducts transactions using the Movement network
- **Suzuka Full Node:** Responsible for executing transactions
- **Suzuka DA Light Node:** Responsible for ordering transactions and packaging blocks
- **Attester:** Responsible for voting on BlockCommits
- **Ethereum:** Responsible for storing BlockCommits

4 Findings

AVM-1 No Gas Charges for Failed Transactions

Severity: Critical

Discovery Methods: Manual Review

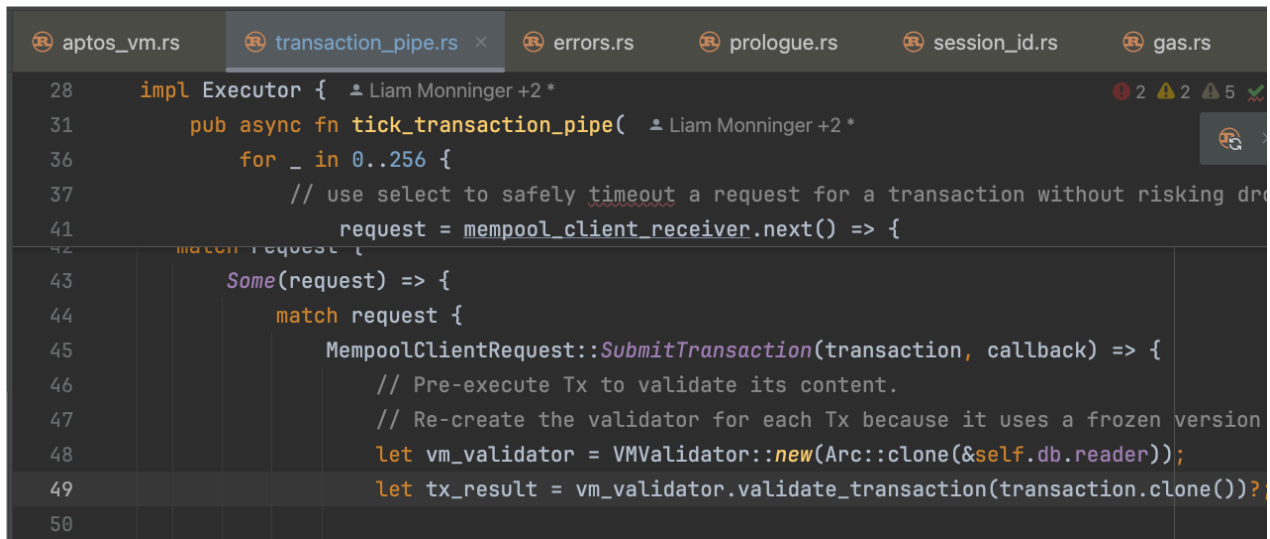
Status: Partially Fixed

Code Location:

aptos-move/aptos-vm/src/aptos_vm.rs#1654

Descriptions:

Pre-processing checks are performed on transactions as they enter the mempool

A screenshot of a code editor showing Rust code in the file aptos_vm.rs. The code is part of an impl block for the Executor trait. It defines a function tick_transaction_pipe which runs a loop from 0 to 256. Inside the loop, it selects a transaction request from a mempool client receiver. It then matches the request, and for a SubmitTransaction request, it pre-executes the transaction to validate its content. This involves creating a VMValidator with a frozen version of the database reader and then calling validate_transaction on the transaction. The code is as follows:

```
28 impl Executor {
31   pub async fn tick_transaction_pipe() {
36     for _ in 0..256 {
37       // use select to safely timeout a request for a transaction without risking dropping
41       request = mempool_client_receiver.next() => {
42         match request {
43           Some(request) => {
44             match request {
45               MempoolClientRequest::SubmitTransaction(transaction, callback) => {
46                 // Pre-execute Tx to validate its content.
47                 // Re-create the validator for each Tx because it uses a frozen version
48                 let vm_validator = VMValidator::new(Arc::clone(&self.db.reader));
49                 let tx_result = vm_validator.validate_transaction(transaction.clone());
50             }
42         }

```

But the SEQUENCE_NUMBER_TOO_NEW error for transactions with sequence_number greater than the current account.sequence_number is covered in the function. This is as expected (because future transactions are also going into the pool pending.) But the too_new error for transactions with sequence_number greater than the current account.sequence_number is covered in the function. This is as expected (since future transactions are also going into the transaction pool pending

```

2494 impl VMValidator for AptosVM {
2506   fn validate_transaction(
2554
2555       let storage : TraversalStorage = TraversalStorage::new();
2556
2557       // Increment the counter for transactions verified.
2558       let (counter_label : &str, result : VMValidatorResult) = match self.validate_signed_transaction(
2559           &mut session,
2560           &resolver,
2561           &txn,
2562           &txn_data,
2563           &log_context,
2564           is_approved_gov_script,
2565           &mut TraversalContext::new(&storage),
2566       ) {
2567           Err(err : VMStatus) if err.status_code() != StatusCode::SEQUENCE_NUMBER_TOO_NEW => (
2568               "failure",
2569               VMValidatorResult::new(Some(err.status_code()), score: 0),
2570           ),
2571           _ => (
2572               "success",
2573               VMValidatorResult::new(None, txn.gas_unit_price()),
2574           ),
2575       };

```

At this point the transaction goes to the mempool and is written to the DA. After that it will read the transaction from DA and execute it when the code is as follows:

```

aptos_vm.rs x transaction_pipe.rs vm_validator.rs errors.rs prologue.rs session_i
214 impl AptosVM {
1627   fn execute_user_transaction_impl(
1643       let exec_result : Result<&mut PrologueSession> = prologue_session.execute(|session : &mut SessionExt
1644           self.validate_signed_transaction(
1645               session,
1646               resolver,
1647               txn,
1648               &txn_data,
1649               log_context,
1650               is_approved_gov_script,
1651               &mut traversal_context,
1652           )
1653       });
1654       unwrap_or_discard!(exec_result);
1655       let storage_gas_params : &StorageGasParameters = unwrap_or_discard!(get_or_vm_startup_failure(
1656           &self.storage_gas_params,
1657           log_context
1658       ));

```

At this point, the transaction is still checked for the same things that are checked during preprocessing.

```

fun prologue_common(
    sender: signer,
    gas_payer: address,
    txn_sequence_number: u64,
    txn_authentication_key: vector<u8>,
    txn_gas_price: u64,
    txn_max_gas_units: u64,
    txn_expiration_time: u64,
    chain_id: u8,
) {
    assert!(
        timestamp::now_seconds() < txn_expiration_time,
        error::invalid_argument(PROLOGUE_ETRANSACTION_EXPIRED),
    );
    assert!(chain_id::get() == chain_id, error::invalid_argument(PROLOGUE_EBAD_CHAIN_ID));

    let transaction_sender = signer::address_of(&sender);

    if (

```

aptos_vm.rs × transaction_pipe.rs transaction_validation.move × vm_validator.rs errors

Plugins supporting *.move files found. [Install plugins](#) [log](#)

```

101         );
102
103         let account_sequence_number = account::get_sequence_number(transaction_sender);
104         assert!(
105             txn_sequence_number < (1u64 << 63),
106             error::out_of_range(PROLOGUE_ESEQUENCE_NUMBER_TOO_BIG)
107         );
108
109         assert!(
110             txn_sequence_number >= account_sequence_number,
111             error::invalid_argument(PROLOGUE_ESEQUENCE_NUMBER_TOO_OLD)
112         );
113
114         assert!(
115             txn_sequence_number == account_sequence_number,
116             error::invalid_argument(PROLOGUE_ESEQUENCE_NUMBER_TOO_NEW)
117         );
118     } else {
119         // In this case, the transaction is sponsored and the account does not exist
120         // the default values match

```

In this case, the sequence_number is higher than the current sequence_number, resulting in an error. However, the handling fee is deducted after this check is performed, so no handling fee is deducted for the transaction. This issue can lead to DA of transactions, mempool being dosed at 0 cost.

Suggestion:

It is suggested to refer to the design idea of aptos to do the pending processing for the transaction whose sequence_number is larger than the current sequence_number.

LIB-1 The Transaction Sorting Algorithm In RocksdbMempool Can Lead To A Series Of Issues

Severity: Major

Discovery Methods: Manual Review

Status: Fixed

Code Location:

protocol-units/mempool/move-rocks/src/lib.rs#37

Descriptions:

The sorting algorithm for transactions in RocksdbMempool is as follows:

1. First, transactions are sorted by timestamp (the timestamp is in 2-second increments).
2. Then, they are sorted by sequence_number .
3. Finally, they are sorted by transaction hash.

```
pub fn construct_mempool_transaction_key(transaction: &MempoolTransaction) ->
String {
    // pad to 32 characters for slot_seconds
    let slot_seconds_str = format!("{:032}", transaction.timestamp);

    // pad to 32 characters for sequence number
    let sequence_number_str = format!("{:032}",
transaction.transaction.sequence_number);

    // Assuming transaction.transaction.id() returns a hex string of length 32
    let transaction_id_hex = transaction.transaction.id(); // This should be a String of hex
characters

    // Concatenate the two parts to form a 80-character hex string key
    let key = format!("{:032}:{:032}", slot_seconds_str, sequence_number_str,
transaction_id_hex);

    key
}
```

The light_node transaction packing algorithm works as follows:

1. It packs 10 transactions into one block.
2. If there are fewer than 10 transactions, it packs a block every second.

This could result in the following situation:

1. Transactions A and B belong to the same account.
2. Transaction A has a timestamp of 12 and a `sequence_number` of 11.
3. Transaction B has a timestamp of 10 and a `sequence_number` of 15.
4. In this case, transaction B would be executed before transaction A. However, Aptos rules require that transaction A must be executed before transaction B.

This sorting algorithm could lead to a series of unknown issues, which would require a deep analysis of the Aptos-core module to understand fully.

Suggestion:

It is recommended to refer to the sorting method used by Aptos-core.

LIB-2 Lack of Atomicity in Multiple Database Write Operations in mempool transaction handling methods

Severity: Informational

Discovery Methods: Manual Review

Status: Fixed

Code Location:

protocol-units/mempool/move-rocks/src/lib.rs#92,93,107,111,152,158

Descriptions:

In the current implementation of the `RocksdbMempool`, the mempool transaction handling methods `add_mempool_transaction`, `remove_mempool_transaction`, and `pop_mempool_transaction` perform multiple write operations to the database. However, these operations are not atomic, meaning that if an error occurs during one of these write operations, it can leave the database in an inconsistent state. Specifically, the issue arises because each method makes multiple calls to `delete_cf` and `put_cf` without wrapping them in a batch operation to ensure atomicity.

Suggestion:

To ensure atomicity and maintain database consistency, it is recommended to use RocksDB's batch operations for all database writes within these methods. By grouping multiple write operations into a single batch, we can guarantee that either all operations succeed or none of them are applied, thus preserving the integrity of the database.

MAI-1 Centralization Risk Due to High Trust Assumptions Required

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

networks/suzuka/suzuka-full-node/src/main.rs#8

Descriptions:

The current architecture introduces a centralization risk as it requires higher trust assumptions. Light nodes and Celestia nodes are pivotal to the network's security, and any malicious behavior from these nodes could have severe consequences. This level of centralization necessitates more robust trust assumptions than is ideal, potentially impacting the overall security and reliability of the network.

Suggestion:

It might be helpful to consider a review of the current trust model, with a focus on reducing centralization and trust assumptions on light nodes and Celestia nodes. This could contribute to a more balanced distribution of trust and further enhance the security of the network.

MOD-1 Lack of Garbage Collection Mechanism in Movement-Aptos-Core's Mempool

Severity: Major

Discovery Methods: Manual Review

Status: Fixed

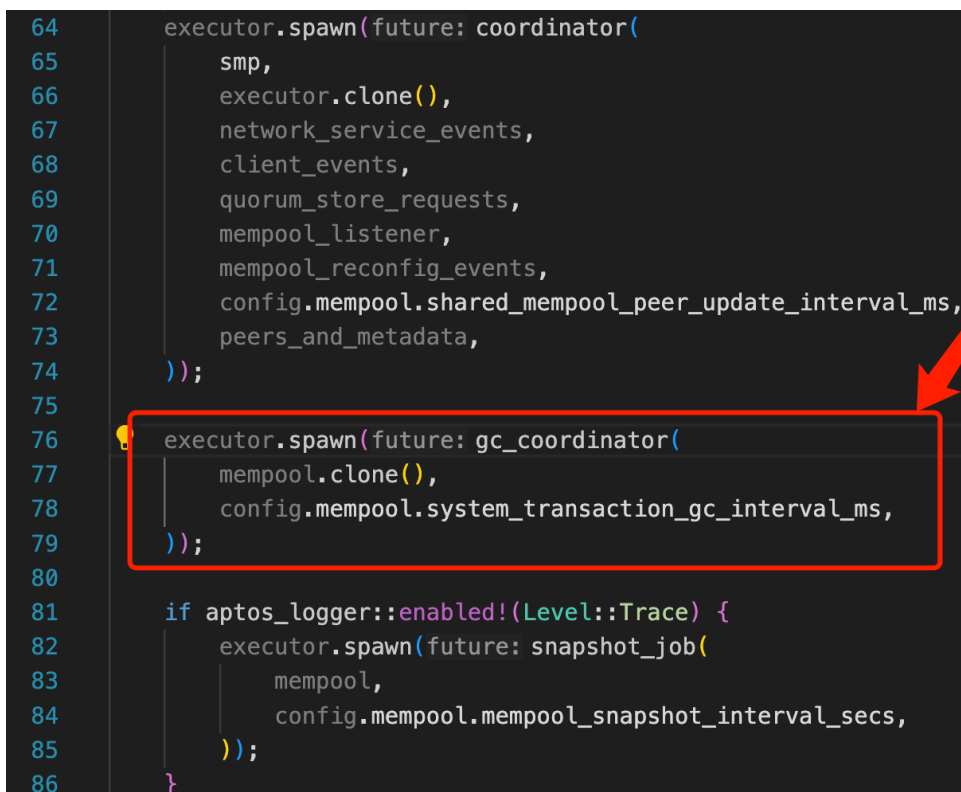
Code Location:

protocol-units/execution/opt-executor/src/executor/mod.rs#59

Descriptions:

A complete mempool should have a garbage collection mechanism, also known as a transaction aging mechanism, which periodically removes expired and unused transactions to prevent the mempool from growing indefinitely.

Aptos's garbage collection code is as follows:



```
64     executor.spawn(future: coordinator(  
65         smp,  
66         executor.clone(),  
67         network_service_events,  
68         client_events,  
69         quorum_store_requests,  
70         mempool_listener,  
71         mempool_reconfig_events,  
72         config.mempool.shared_mempool_peer_update_interval_ms,  
73         peers_and_metadata,  
74     ));  
75  
76     executor.spawn(future: gc_coordinator(  
77         mempool.clone(),  
78         config.mempool.system_transaction_gc_interval_ms,  
79     ));  
80  
81     if aptos_logger::enabled!(Level::Trace) {  
82         executor.spawn(future: snapshot_job(  
83             mempool,  
84             config.mempool.mempool_snapshot_interval_secs,  
85         ));  
86     }
```

Suggestion:

It is recommended to enable Aptos's mempool garbage collection feature.

PAR-1 DoS Vulnerability Due to Sequencer Not Checking for Sequence Number Too New During Block Packaging

Severity: Critical

Discovery Methods: Manual Review

Status: Partially Fixed

Code Location:

networks/suzuka/suzuka-full-node/src/partial.rs#126,142-227

Descriptions:

The reason for this DoS vulnerability is that the transaction of `sequence_number_too_new` will be packaged into the block and the block will be executed. Attackers can exploit this vulnerability by creating a transaction with a `sequence_number` greater than the current value (e.g., `sequence_number = 2` when the current is 0) and setting the `expiration_timestamp_secs` to a large value. After signing and sending such a transaction, it gets included in a block. Although the transaction itself will not succeed, repeatedly sending the transaction causes the block to be re-executed multiple times without incurring any cost to the attacker. Local testing has shown that concurrent sending of these attack transactions can significantly degrade the node's performance, preventing it from handling legitimate transactions.

```
// Blocks read from DA are executed without complete verification.  
let executable_block = ExecutableBlock::new(block_hash, block);  
let block_id = executable_block.block_id;  
let commitment = self.executor.execute_block_opt(executable_block).await?;
```

PoC

1. Transfer any amount of tokens to the attacker's address:

`0x60b5f67bb14334b371f207df24d6f717e50941a67f84db471d661de5140e23c9` .

2. Concurrently send the following POST request:

```
{  
  "sender":
```

```
"0x60b5f67bb14334b371f207df24d6f717e50941a67f84db471d661de5140e23c9",
  "sequence_number": "2",
  "max_gas_amount": "13",
  "gas_unit_price": "100",
  "expiration_timestamp_secs": "1755308158",
  "payload": {
    "function": "0x1::aptos_account::transfer",
    "type": "entry_function_payload",
    "type_arguments": [],
    "arguments": [
      "0x60b5f67bb14334b371f207df24d6f717e50941a67f84db471d661de5140e23c8",
      "100"
    ]
  },
  "signature": {
    "type": "ed25519_signature",
    "public_key":
"0x18367b2687ee313cc3baea86d31619e5bef1158df4bb0f6d2476a9601e978ae2",
    "signature":
"0x45e9e00617dc0af6160412bada9588853e4a9cb7aea893e202efcfc37c735d9233b108f61b8
  }
}
```

Another similar and noteworthy attack method involves an attacker constructing a large number of non-ready transactions with different nonce values to try to fill up the mempool.

Suggestion:

- Transaction Validation: Validate transactions before executing the block to identify and reject potentially malicious transactions that could lead to DoS attacks.
- To prevent the Aptos mempool from being filled, you can refer to the implementation of Aptos [transaction eviction mechanism](#).

PAR-2 Use a Higher Security Level Hash Function

Severity: Informational

Discovery Methods:

Status: Fixed

Code Location:

networks/suzuka/suzuka-full-node/src/partial.rs#199-202

Descriptions:

```
3 // hash the block bytes
9 let mut hasher : CoreWrapper<CtVariableCoreWrapper<...>> = sha2::Sha256::new();
0 hasher.update(&block_bytes);
1 let slice : Box<[u8]> = hasher.finalize();
2 let block_hash : HashValue = HashValue::from_slice(slice.as_slice())?;
```

It is recommended to use a higher security level hash function (eg.sha3)The Sha256 algorithm is susceptible to hash length expansion attacks and will require urgent upgrades in the future if security issues arise

Suggestion:

It is recommended to use a higher security level hash function (eg.sha3)

ROC-1 High Time Complexity in `num_nodes` Function Leading to Potential DOS Vulnerability

Severity: Minor

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/storage/jelly-move/src/rocksdb.rs#214-222

Descriptions:

The function `num_nodes` has a high time complexity due to its implementation, which requires iterating through all records in RocksDB.

```
pub fn num_nodes(&self) -> usize {
    let cf_handle = self.nodes_cf().unwrap();
    let mut iter = self.db.iterator_cf(&cf_handle, rocksdb::IteratorMode::Start);
    let mut count = 0;
    while let Some(_) = iter.next() {
        count += 1;
    }
    count
}
```

This function iterates over all records in the `nodes_cf` column family to count the number of nodes. The time complexity is proportional to the number of records, which can be extremely high for large datasets. In local testing, counting 20,000,000 nodes took approximately 30.5 seconds, whereas using `rocksdb.estimate-num-keys` took only 13.9 microseconds.

Suggestion:

To reduce the time complexity and mitigate the risk of DOS attacks, it is recommended to use `rocksdb.estimate-num-keys`, which provides an estimated count of the keys in the database much more efficiently.

Resolution:

Current implementation has been deprecated.

ROC-2 Incorrect Implementation of `get_rightmost_leaf` Leading to Inaccurate Node Retrieval

Severity: Minor

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

`protocol-units/storage/jelly-move/src/rocksdb.rs#156-164`

Descriptions:

The function `get_rightmost_leaf` might not return the actual rightmost leaf node.

The issue arises because using `rocksdb::IteratorMode::End` results in keys being sorted in descending order based on their serialized byte representation using `borsh::to_vec()`. This means higher versions of `NodeKeyNodeKey` will be sorted before lower versions, even if the latter is actually the rightmost leaf. For example:

```
NodeKey { version: 1, nibble_path: e } will be sorted before  
NodeKey { version: 0, nibble_path: f }
```

even though the latter is the rightmost leaf node.

Suggestion:

To ensure that the correct rightmost leaf node is retrieved, a possible solution is to adjust the order of the fields in `NodeKey` by swapping `nibble_path` and `version`. This adjustment will prioritize sorting by `nibble_path` first, followed by `version`. This approach will ensure the correct rightmost leaf node is found. This change would require further detailed testing and analysis to confirm its effectiveness.

Resolution:

Current implementation has been deprecated.

ROC-3 Lack of Atomicity in Multiple Database Write Operations in RocksdbJmt handling methods

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/storage/jelly-move/src/rocksdb.rs#96,108,188,204,206

Descriptions:

In the current implementation of the `RocksdbJmt`, the `RocksdbJmt` handling methods `write_node_batch`, `write_tree_update_batch`, and `purge_stale_nodes` perform multiple write operations to the database. However, these operations are not atomic, meaning that if an error occurs during one of these write operations, it can leave the database in an inconsistent state. Specifically, the issue arises because each method makes multiple calls to `delete_cf` and `put_cf` without wrapping them in a batch operation to ensure atomicity.

Suggestion:

To ensure atomicity and maintain database consistency, it is recommended to use RocksDB's batch operations for all database writes within these methods. By grouping multiple write operations into a single batch, we can guarantee that either all operations succeed or none of them are applied, Thus ensuring the atomicity of operations.

Resolution:

Current implementation has been deprecated.

ROC-4 Inefficient Implementation of `get_value_option` Leading to Potential DOS Vulnerability

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/storage/jelly-move/src/rocksdb.rs#143

Descriptions:

The function `get_value_option` has an inefficient implementation that can lead to potential Denial-of-Service (DOS) attacks. This function iterates backwards from `max_version` down to `0` until it finds a value. This can be inefficient and poses a risk for DOS attacks, especially if `max_version` is a large integer. The time complexity of the function is directly proportional to `max_version`, making it vulnerable to excessive delays when handling large version numbers.

In comparison, the implementation from the [reference project](#) does not iterate over `max_versions` down to `0` but rather accesses the version history and iterates only within the relevant versions.

Suggestion:

- Instead of iterating through all versions from `max_version` down to `0`, consider maintaining a version history that allows for efficient access and iteration.
- Add input validation to ensure `max_version` is within an acceptable range, preventing malicious actors from passing excessively large values.

Resolution:

Current implementation has been deprecated.

ROC-5 db Object Inside RocksdbJmt Should Have A read-write Lock Added

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/storage/jelly-move/src/rocksdb.rs#21

Descriptions:

Although `rocksdb::DB` is already interior mutably locked, `RocksdbJmt` uses `rocksdb::DB` multiple times within the same function, making this function not an atomic operation. This could lead to unpredictable consequences due to concurrency.

```
pub fn purge_stale_nodes(&self, last_readable_version: Version) -> Result<(),  
anyhow::Error> {  
    let cf_handle = self.stale_nodes_cf()?;  
    let mut iter = self.db.iterator_cf(&cf_handle, rocksdb::IteratorMode::Start);  
    while let Some(res) = iter.next() {  
        let (key, _) = res?;  
        let key : (Version, KeyHash) = BorshDeserialize::try_from_slice(&key)?;  
        if key.0 <= last_readable_version {  
            let cf_handle = self.nodes_cf()?;  
            self.db.delete_cf(&cf_handle, borsh::to_vec(&key.1)?);  
            //  
            //maybe there is another thread reading or writing between 2 `delete_cf`  
operations.  
            //  
            self.db.delete_cf(&cf_handle, borsh::to_vec(&key)?);  
        }  
    }  
  
    Ok(())  
}
```

Suggestion:

It is recommended to add a read-write lock to the `db` object.

Resolution:

Current implementation has been deprecated.

SEQ-1 Lack of Garbage Collection Mechanism in Mempool

Severity: Major

Discovery Methods: Manual Review

Status: Fixed

Code Location:

protocol-units/da/m1/light-node/src/v1/sequencer.rs#19

Descriptions:

A complete mempool should have a garbage collection mechanism, also known as a transaction aging mechanism, which periodically removes expired and unused transactions to prevent the mempool from growing indefinitely.

Since Movement's mempool is based on RocksDB and stored on disk, the issue becomes more severe. In this case, a node crash or server restart will not result in the deletion of the old, expired transactions.

Suggestion:

It is recommended to establish a mempool garbage collection mechanism by referencing Aptos.

SER-1 Enforce HTTPS-Only Access Full Node API to Mitigate Security Risks

Severity: Informational

Discovery Methods:

Status: Fixed

Code Location:

protocol-units/execution/opt-executor/src/executor/services.rs#14

Descriptions:

The current testnet api `http://aptos.testnet.suzuka.movementlabs.xyz/v1` , supports both HTTP and HTTPS protocols. While both protocols are functional, the use of HTTP increases the risk of man-in-the-middle (MitM) attacks.

Suggestion:

Enforce the use of HTTPS for the full node API to mitigate the risk of man-in-the-middle (MitM) attacks. Ensure that all interactions with the API, including RPC interfaces of light nodes, are conducted over a secure connection to enhance overall security. Additionally, make sure that only full nodes have access to these interfaces, preventing unauthorized calls from third parties. This will help avoid various Denial-of-Service (DoS) and security issues that could arise from exposing these interfaces.

TPI-1 Invalid sequence_number Check for Mempool in aptos-core

Severity: Informational

Discovery Methods: Manual Review

Status: Fixed

Code Location:

protocol-units/execution/opt-executor/src/executor/transaction_pipe.rs#61-67

Descriptions:

Transactions are added to core_mempool after they have been pre-executed:

```
28 impl Executor {
31     pub async fn tick_transaction_pipe(
36         for _ in 0..256 {
37             // use select to safely timeout a request for a transaction without risking dr
41             request = mempool_client_receiver.next() => {
55             } else {
56                 // add to the mempool
57                 let mut core_mempool : RwLockWriteGuard<Mempool> = self.core_mempool.w
58
59                 debug!("Adding transaction to mempool: {:?} {:?}", transaction, tra
60
61                 let status : MempoolStatus = core_mempool.add_txn(
62                     transaction.clone(),
63                     0,
64                     transaction.sequence_number(),
65                     TimelineState::NonQualified,
66                     true
67                 );
68
69                 match status.code {
70                     MempoolStatusCode::Accepted => {
```

The join is checked against the sequence_number of the joining transaction:

```

227     pub fn add_txn(
234     ) -> MempoolStatus {
235         trace!(
236             LogSchema::new(LogEntry::AddTxn)
237             .txns(TxnsLog::new_txn(txn.sender(), txn.sequence_number())),
238             committed_seq_number = db_sequence_number
239         );
240
241         // don't accept old transactions (e.g. seq is less than account's current seq_number)
242         if txn.sequence_number() < db_sequence_number {
243             return MempoolStatus::new(MempoolStatusCode::InvalidSeqNumber).with_message( message:
244                 "transaction sequence number is {}, current sequence number is {}",
245                 txn.sequence_number(),
246                 db_sequence_number,
247             );
248         }
249
250         let now = SystemTime::now();

```

Since the `db_sequence_number` passed in is `transaction.sequence_number`, this check does not work, but it is not harmful because of the pre-execution check that was performed earlier.

Suggestion:

Change the incoming parameter to the real `db_sequence_number`.

TRE-1 Library Function `batch_put_value_sets` Throws Panic, but This Is Not Mentioned in the Documentation

Severity: Medium

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

`protocol-units/cryptography/tentacles/src/tree.rs#74`

Descriptions:

In the function `jmt::tree::JellyfishMerkleTree::batch_put_value_sets`, if the developer inputs an empty `value_set`, it directly throws a panic.

```
assert!(  
    !value_set.is_empty(),  
    "Transactions that output empty write set should not be included.",  
);
```

Suggestion:

Return an error if the developer inputs an empty `value_set`

Resolution:

JMT is not in use.

TRE-2 Library Function `batch_put_value_sets` Throws Panic, but This Is Not Mentioned in the Documentation

Severity: Medium

Discovery Methods:

Status: Acknowledged

Code Location:

`protocol-units/cryptography/tentacles/src/tree.rs#87`

Descriptions:

In the function `jmt::tree::JellyfishMerkleTree::batch_put_value_sets`, if the developer inputs `value_sets` and `node_hashes` of different lengths, it directly throws a panic.

```
for (idx, (value_set, hash_set)) in
    itertools::zip_eq(value_sets.into_iter(), hash_sets.into_iter()).enumerate()
{
```

Suggestion:

Return an error if the developer inputs `value_sets` and `node_hashes` of different lengths

Resolution:

JMT is not in use.

TRE-3 Library Function `batch_put_value_sets` Throws Panic, but This Is Not Mentioned in the Documentation

Severity: Medium

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

`protocol-units/cryptography/tentacles/src/tree.rs#66`

Descriptions:

In the function `jmt::tree::JellyfishMerkleTree::batch_put_value_sets`, if the `node_hashes` provided by the developer are not in a tree structure, it directly throws a panic.

```
fn get_hash(
    node_key: &NodeKey,
    node: &Node,
    hash_cache: &Option<&HashMap<NibblePath, [u8; 32]>>,
) -> [u8; 32] {
    if let Some(cache) = hash_cache {
        match cache.get(node_key.nibble_path()) {
            Some(hash) => *hash,
            //it is reachable!!!!!!!
            None => unreachable!("{:?} can not be found in hash cache", node_key),
        }
    } else {
        node.hash::<H>()
    }
}
```

Suggestion:

Remove the `unreachable` function and return an error instead

Resolution:

JMT is not in use.

TRE-4 Common Prefix Nodes Increasing Tree Height, Leading to Higher Computational and Storage Resource Consumption

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/cryptography/tentacles/src/tree.rs#74

Descriptions:

In the current implementation of the Jelly Merkle Tree (JMT), inserting keys with multiple common prefixes results in the creation of multiple InternalNodes, which consequently increases the tree's height. According to the [white paper](#), for one billion randomly distributed leaf nodes, the average tree height should be around 8. However, deliberately constructed keys with 63 common prefixes can increase the tree height to 64. If an attacker can continuously construct leaf nodes with common prefixes at a low cost, it could exacerbate the storage and computational load on the full nodes.

Suggestion:

Restrict the ability of users to control key selection, increasing the difficulty and cost for attackers to construct keys with numerous common prefixes and reducing the potential for abuse.

Resolution:

Current implementation has been deprecated.

TRE-5 Lack of Version Parameter Validation in `put_value_sets` Function

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

`protocol-units/cryptography/tentacles/src/tree.rs#344`

Descriptions:

The function `put_value_sets` does not validate the `Version` parameter. This allows the insertion of higher version data, even if no prior versioned data exists in the tree, as in `tree.put_value_sets(batches.clone(), u64::MAX-1).unwrap();`. This scenario can result in the system reaching the maximum version value, preventing further data insertions.

Suggestion:

Ensure that the versioning system correctly tracks and enforces sequential version increments

Resolution:

Current implementation has been deprecated.

TRE-6 Use of SHA256 in JMT Implementation Not Suitable for ZK Proofs

Severity: Informational

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/cryptography/tentacles/src/tree.rs#29

Descriptions:

The current implementation of the Jelly Merkle Tree (JMT) uses SHA256 as the hash function. However, according to the development documentation, "Aptos has already implemented a Jellyfish Merkle Tree storage for Move accounts and modules. We will use their implementation as a reference. However, we do not recommend using their implementation directly as it is not suitable to the exact pre-flight requirements of the M1 ZKFP implementation." For the intended application in M1 ZKFP, generating zero-knowledge proofs (zk proofs) requires a hash function that is more efficient in zk contexts. SHA256 is not zk-friendly and results in higher computational resource consumption when generating proofs.

Suggestion:

Replace SHA256 with zk-friendly hash functions such as MiMC or Poseidon for constructing the JMT. These hash functions are designed to be more efficient for zero-knowledge proof generation, reducing computational overhead and improving performance in zk applications.

Resolution:

Current implementation has been deprecated.

LIB1-1 Multithreading Issue Leads to Transaction Replay Vulnerability

Severity: Critical

Discovery Methods: Manual Review

Status: Fixed

Code Location:

protocol-units/mempool/util/src/lib.rs#50;

protocol-units/da/m1/light-node/src/v1/sequencer.rs#76;

networks/suzuka/suzuka-full-node/src/partial.rs#134

Descriptions:

In the light-node, when adding a transaction to the mempool, the

`mempool_util::MempoolTransactionOperations::add_transaction` function is called, as shown below:

```
/// Adds a transaction to the mempool.
async fn add_transaction(&self, tx: Transaction) -> Result<(), anyhow::Error> {
    //
    // check if Tx exists to avoid a replay attack!!!
    //
    if self.has_transaction(tx.id()).await? {
        return Ok(());
    }

    let mempool_transaction = MempoolTransaction::slot_now(tx);
    self.add_mempool_transaction(mempool_transaction).await
}
```

This function checks whether the transaction already exists to prevent replay attacks.

However, this check can be bypassed. The `tick_write_transactions_to_da` thread of the full-node is responsible for writing transactions to the mempool, while the `run_block_proposer` thread is responsible for popping transactions out of the mempool and sending them to the DA.

If a transaction A is popped out, it no longer exists in the mempool, and it can be re-added to the mempool. The prerequisite for this attack is that transaction A has not yet been executed, but this condition is easily met because the speed of writing to the mempool is faster than the speed of executing blocks.

Impact: Under normal circumstances, repeated transactions that are packaged into a block will not execute successfully but will consume gas fees. A third-party attacker can exploit this by replaying a large number of user transactions, resulting in asset loss for the victim.

Suggestion:

The fix is quite complex, it is recommended to refer to Aptos's replay prevention mechanism. A simple approach could be to revalidate transactions when packaging a block after the execution of a block.

LIB1-2 Code Style Recommendations

Severity: Medium

Discovery Methods: Manual Review

Status: Acknowledged

Code Location:

protocol-units/cryptography/tentacles/src/lib.rs#1

Descriptions:

In the `tentacles` module source code, there are too many instances of `except`, `unwrap`, and `unreachable` code that may cause panic. This not only poses security risks but can also cause confusion for developers

Suggestion:

Modify places where panic is thrown to return an error instead

Resolution:

JMT is not in use.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information or assets at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information or assets at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.

