SHERLOCK

Security Review For Interchain Labs 2

Collaborative Audit Prepared For: Interchain Labs 2

Lead Security Expert(s): Bauchibred
hack3r-0m
iammxuse
kuprum

Date Audited: February 12 - March 6, 2025

Introduction

IBC Eureka is a new, simplified protocol for inter-blockchain communication. The audit focuses on making sure it is ready for a safe launch on the Cosmos Hub.

Scope

Repository: cosmos/ibc-go

Audited Commit: 79218a531e769bb5c29022d50ef017bd81e4bd9b

Final Commit: d316a8b6c3716bad1bc2b1e8a3f573a7144dfc2d

Files:

- modules/apps/callbacks/go.mod
- modules/apps/callbacks/go.sum
- modules/apps/callbacks/internal/process.go
- modules/apps/callbacks/types/callbacks.go
- modules/apps/callbacks/types/expected_keepers.go
- modules/apps/callbacks/types/keys.go
- modules/apps/callbacks/v2/ibc_middleware.go
- modules/apps/transfer/keeper/genesis.go
- modules/apps/transfer/keeper/grpc_query.go
- modules/apps/transfer/keeper/keeper.go
- modules/apps/transfer/keeper/migrations.go
- modules/apps/transfer/keeper/msg_server.go
- modules/apps/transfer/keeper/relay.go
- modules/apps/transfer/types/denom.go
- modules/apps/transfer/types/encoding.go
- modules/apps/transfer/types/genesis.go
- modules/apps/transfer/types/hop.go
- modules/apps/transfer/types/keys.go
- modules/apps/transfer/types/msgs.go
- modules/apps/transfer/types/packet.go
- modules/apps/transfer/types/solidity_abi.go
- modules/apps/transfer/types/token.go

- modules/apps/transfer/v2/ibc_module.go
- modules/core/02-client/genesis.go
- modules/core/02-client/v2/genesis.go
- modules/core/02-client/v2/keeper/grpc_query.go
- modules/core/02-client/v2/keeper/keeper.go
- modules/core/02-client/v2/module.go
- modules/core/02-client/v2/types/codec.go
- modules/core/02-client/v2/types/counterparty.go
- modules/core/02-client/v2/types/genesis.go
- modules/core/02-client/v2/types/keys.go
- modules/core/02-client/v2/types/msgs.go
- modules/core/04-channel/genesis.go
- modules/core/04-channel/module.go
- modules/core/04-channel/v2/genesis.go
- modules/core/04-channel/v2/keeper/grpc_query.go
- modules/core/04-channel/v2/keeper/keeper.go
- modules/core/04-channel/v2/keeper/msg_server.go
- modules/core/04-channel/v2/keeper/packet.go
- modules/core/04-channel/v2/module.go
- modules/core/04-channel/v2/types/acknowledgement.go
- modules/core/04-channel/v2/types/commitment.go
- modules/core/04-channel/v2/types/genesis.go
- modules/core/04-channel/v2/types/keys.go
- modules/core/04-channel/v2/types/merkle.go
- modules/core/04-channel/v2/types/msgs.go
- modules/core/04-channel/v2/types/packet.go
- modules/core/24-host/v2/packet_keys.go
- modules/core/api/module.go
- modules/core/api/router.go
- modules/light-clients/08-wasm/blsverifier/crypto.go
- modules/light-clients/08-wasm/blsverifier/handler.go

Repository: cosmos/solidity-ibc-eureka

Audited Commit: 4edeb49d9d439e4d882ffa9ed37519cbc1123e25

Final Commit: 67eeb109f6d3349653b3b07e7986395f3eca2402

Files:

- contracts/ICS20Transfer.sol
- contracts/ICS26Router.sol
- contracts/interfaces/IEscrow.sol
- contracts/interfaces/IIBCApp.sol
- contracts/interfaces/IIBCERC20.sol
- contracts/interfaces/IIBCPausable.sol
- contracts/interfaces/IIBCStore.sol
- contracts/interfaces/IIBCUUPSUpgradeable.sol
- contracts/interfaces/IICS02Client.sol
- contracts/interfaces/IICS20Transfer.sol
- contracts/interfaces/IICS26Router.sol
- contracts/interfaces/ILightClient.sol
- contracts/interfaces/IRateLimit.sol
- contracts/light-clients/ISP1ICS07Tendermint.sol
- contracts/light-clients/SP1ICS07Tendermint.sol
- contracts/light-clients/msgs/IICS07TendermintMsgs.sol
- contracts/light-clients/msgs/IMembershipMsgs.sol
- contracts/light-clients/msgs/IMisbehaviourMsgs.sol
- contracts/light-clients/msgs/ISP1Msgs.sol
- contracts/light-clients/msgs/IUcAndMembershipMsgs.sol
- contracts/light-clients/msgs/IUpdateClientMsgs.sol
- contracts/light-clients/utils/Paths.sol
- contracts/msgs/IIBCAppCallbacks.sol
- contracts/msgs/IICS02ClientMsgs.sol
- contracts/msgs/IICS20TransferMsgs.sol
- contracts/msgs/IICS26RouterMsgs.sol

- contracts/msgs/ILightClientMsgs.sol
- contracts/utils/Escrow.sol
- contracts/utils/IBCERC20.sol
- contracts/utils/IBCIdentifiers.sol
- contracts/utils/IBCPausableUpgradeable.sol
- contracts/utils/IBCStoreUpgradeable.sol
- contracts/utils/IBCUUPSUpgradeable.sol
- contracts/utils/ICS02ClientUpgradeable.sol
- contracts/utils/ICS20Lib.sol
- contracts/utils/ICS24Host.sol
- contracts/utils/RateLimitUpgradeable.sol
- packages/ethereum/ethereum-light-client/src/client_state.rs
- packages/ethereum/ethereum-light-client/src/consensus_state.rs
- packages/ethereum/ethereum-light-client/src/header.rs
- packages/ethereum/ethereum-light-client/src/lib.rs
- packages/ethereum/ethereum-light-client/src/membership.rs
- packages/ethereum/ethereum-light-client/src/misbehaviour.rs
- packages/ethereum/ethereum-light-client/src/trie.rs
- packages/ethereum/ethereum-light-client/src/update.rs
- packages/ethereum/ethereum-light-client/src/verify.rs
- packages/ethereum/ethereum-trie-db/src/error.rs
- packages/ethereum/ethereum-trie-db/src/lib.rs
- packages/ethereum/ethereum-trie-db/src/trie_db.rs
- packages/ethereum/ethereum-trie-db/src/types.rs
- packages/ethereum/ethereum-types/src/consensus/bls.rs
- packages/ethereum/ethereum-types/src/consensus/domain.rs
- packages/ethereum/ethereum-types/src/consensus/fork.rs
- packages/ethereum/ethereum-types/src/consensus/light_client_header.rs
- packages/ethereum/ethereum-types/src/consensus/merkle.rs
- packages/ethereum/ethereum-types/src/consensus/mod.rs
- packages/ethereum/ethereum-types/src/consensus/signing_data.rs

- packages/ethereum/ethereum-types/src/consensus/slot.rs
- packages/ethereum/ethereum-types/src/consensus/spec.rs
- packages/ethereum/ethereum-types/src/consensus/sync_committee.rs
- packages/ethereum/ethereum-types/src/execution/account_proof.rs
- packages/ethereum/ethereum-types/src/execution/mod.rs
- packages/ethereum/ethereum-types/src/execution/storage_proof.rs
- packages/ethereum/ethereum-types/src/lib.rs
- packages/ethereum/tree_hash/src/impls.rs
- packages/ethereum/tree_hash/src/lib.rs
- packages/ethereum/tree_hash/src/merkle_hasher.rs
- packages/ethereum/tree_hash/src/merkleize_padded.rs
- packages/ethereum/tree_hash/src/merkleize_standard.rs
- packages/ethereum/tree_hash/src/sha256.rs
- packages/sp1-ics07-tendermint-prover/build.rs
- packages/sp1-ics07-tendermint-prover/src/lib.rs
- packages/sp1-ics07-tendermint-prover/src/programs.rs
- packages/sp1-ics07-tendermint-prover/src/prover.rs
- packages/sp1-ics07-tendermint-utils/src/eth.rs
- packages/sp1-ics07-tendermint-utils/src/lib.rs
- packages/sp1-ics07-tendermint-utils/src/light_block.rs
- packages/sp1-ics07-tendermint-utils/src/merkle.rs
- packages/sp1-ics07-tendermint-utils/src/rpc.rs
- programs/cw-ics08-wasm-eth/src/contract.rs
- programs/cw-ics08-wasm-eth/src/custom_query.rs
- programs/cw-ics08-wasm-eth/src/lib.rs
- programs/cw-ics08-wasm-eth/src/msg.rs
- programs/cw-ics08-wasm-eth/src/query.rs
- programs/cw-ics08-wasm-eth/src/state.rs
- programs/cw-ics08-wasm-eth/src/sudo.rs
- programs/sp1-programs/membership/src/lib.rs
- programs/sp1-programs/membership/src/main.rs

- programs/spl-programs/misbehaviour/src/lib.rs
- programs/spl-programs/misbehaviour/src/main.rs
- programs/spl-programs/misbehaviour/src/types/mod.rs
- programs/spl-programs/misbehaviour/src/types/validation.rs
- programs/spl-programs/uc-and-membership/src/lib.rs
- programs/spl-programs/uc-and-membership/src/main.rs
- programs/spl-programs/update-client/src/lib.rs
- programs/spl-programs/update-client/src/main.rs
- programs/spl-programs/update-client/src/types/mod.rs
- programs/spl-programs/update-client/src/types/validation.rs

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	7	12

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Mismatched slots cause Ethereum light client state to be stored under incorrect keys

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/172

Summary

The `update_consensus_state` function returns an `updated_slot` value that can be inconsistent with the slot in the returned `ConsensusState`. When this value is used as a storage key in the CosmWasm contract, it results in consensus states being stored under incorrect keys, leading to data corruption and breaking state consistency.

Vulnerability Detail

First see update_consensus_state()

```
pub fn update_consensus_state(
    current_consensus_state: ConsensusState,
    current_client_state: ClientState,
    header: Header,
) -> Result<(u64, ConsensusState, Option<ClientState>), EthereumIBCError> {
    let trusted_sync_committee = header.trusted_sync_committee;
    let trusted_slot = trusted_sync_committee.trusted_slot;

    let consensus_update = header.consensus_update;

    let store_period = compute_sync_committee_period_at_slot(
        current_client_state.slots_per_epoch,
        current_client_state.epochs_per_sync_committee_period,
        current_consensus_state.slot,
    );

    let update_finalized_period = compute_sync_committee_period_at_slot(
        current_client_state.slots_per_epoch,
        current_client_state.epochs_per_sync_committee_period,
        consensus_update.attested_header.beacon.slot,
    );

    let mut new_consensus_state = current_consensus_state.clone();
    let mut new_client_state: Option<ClientState> = None;

    if let Some(next_sync_committee) = current_consensus_state.next_sync_committee {
        // sync committee only changes when the period change
        if update_finalized_period == store_period + 1 {
            new_consensus_state.current_sync_committee = next_sync_committee;
            new_consensus_state.next_sync_committee = consensus_update
```

```
                .next_sync_committee
                .map(|c| c.aggregate_pubkey);
        }
    } else {
        // if the finalized period is greater, we have to have a next sync committee
        ensure!(
            update_finalized_period == store_period,
            EthereumIBCError::StorePeriodMustBeEqualToFinalizedPeriod
        );
        new_consensus_state.next_sync_committee = consensus_update
            .next_sync_committee
            .map(|c| c.aggregate_pubkey);
    }

    // Some updates can be only for updating the sync committee, therefore the slot
    ↪  number can be
    // smaller. We don't want to save a new state if this is the case.
    // TODO: we might to remove this functionality if we don't use it as it
    ↪  complicates the light client
    let updated_slot = core::cmp::max(trusted_slot,
    ↪  consensus_update.attested_header.beacon.slot);

    if consensus_update.attested_header.beacon.slot > current_consensus_state.slot {
        new_consensus_state.slot = consensus_update.attested_header.beacon.slot;

        new_consensus_state.state_root =
        ↪  consensus_update.attested_header.execution.state_root;
        new_consensus_state.storage_root =
        ↪  header.account_update.account_proof.storage_root;

        new_consensus_state.timestamp = compute_timestamp_at_slot(
            current_client_state.seconds_per_slot,
            current_client_state.genesis_time,
            consensus_update.attested_header.beacon.slot,
        );

        if current_client_state.latest_slot <
        ↪  consensus_update.attested_header.beacon.slot {
            new_client_state = Some(ClientState {
                latest_slot: consensus_update.attested_header.beacon.slot,
                ..current_client_state
            });
        }
    }

    Ok((updated_slot, new_consensus_state, new_client_state))
}
```

To break this down, when updating only the sync committee (not the full state), the function returns a slot value as its first return parameter that differs from the slot in the consensus state.

The function returns three values:

```
Ok((updated_slot, new_consensus_state, new_client_state))
```

Here, `updated_slot` is calculated as:

```
let updated_slot = core::cmp::max(trusted_slot,
 ↪  consensus_update.attested_header.beacon.slot);
```

However, the consensus state's slot is only updated under certain conditions:

```
if consensus_update.attested_header.beacon.slot > current_consensus_state.slot {
    new_consensus_state.slot = consensus_update.attested_header.beacon.slot;
    // ..snip
}
```

This creates a scenario where the function correctly handles sync committee updates without changing the consensus state's slot (as explained in the comments):

```
// Some updates can be only for updating the sync committee, therefore the slot
 ↪  number can be
// smaller. We don't want to save a new state if this is the case.
```

But crucially, the first return value (`updated_slot`) doesn't reflect this behavior. In cases where we're only updating the sync committee with a smaller slot, `updated_slot` will be smaller than `new_consensus_state.slot`, creating a discrepancy between what the function returns and what the returned state actually contains.

For example, hypothetically, if:

- `current_consensus_state.slot` = 100
- `trusted_slot` = 90
- `consensus_update.attested_header.beacon.slot` = 90

The function would:

1. Update only the sync committee
2. Keep `new_consensus_state.slot` = 100 (unchanged)
3. But return `updated_slot` = 90

Now would be key to note that the verification process in the light client explicitly allows for this scenario. In validate_light_client_update, there's a specific check that permits updating only the sync committee without changing the slot:

```
    ensure!(
        update_attested_slot > trusted_consensus_state.finalized_slot()
            || (update_attested_period == stored_period
                && update.next_sync_committee.is_some()
                && trusted_consensus_state.next_sync_committee().is_none()),
        EthereumIBCError::IrrelevantUpdate { ... }
);
```

Impact

This causes a data inconsistency in the `cw-ics08-wasm-eth` implementation. Cause in `sudo.rs`, the `update_state` function uses the returned `updated_slot` directly as the key to store the consensus state:

```
let (updated_slot, updated_consensus_state, updated_client_state) =
    update_consensus_state(eth_consensus_state, eth_client_state, header)
        .map_err(ContractError::UpdateClientStateFailed)?;

// The updated_slot is used here to store the consensus state
store_consensus_state(deps.storage, &wasm_consensus_state, updated_slot)?;
```

Using our hypothetical slot number, this means that we overwrite what was previously stored under key 90 with a consensus state that has an internal slot value of 100.

This behavior creates a situation where the consensus state's storage location doesn't match its actual slot value. So one can say the system essentially ”loses track” of where updated consensus states are actually stored.

This creates several serious issues:

Cause we now have two different states both claiming to represent slot 100:

- The original state at key=100 (which would not have the latest sync committee updates)
- The updated state at key=90 (which contains updated consensus information)

So any code that tries to retrieve the latest state for `slot 100` will look at `key 100`, which in this case would contain outdated information.

This also means double spend attacks could be executed, since we have double instances of the consensus state at slot `100` stored at two different `key` locations.

Coded POC

Apply these changes to cw-ics08-wasm-eth/src/test/mod.rs

```
use std::marker::PhantomData;
use alloy_primitives::B256;
use cosmwasm_std::{
    testing::{
        mock_dependencies, MockApi, MockQuerier, MockQuerierCustomHandlerResult,
         ↪  MockStorage,
    },
    Binary, OwnedDeps, SystemResult,
};
use ethereum_light_client::test_utils::bls_verifier::{aggreagate,
 ↪  fast_aggregate_verify};
use ethereum_types::consensus::bls::{BlsPublicKey, BlsSignature};

+   // Include our slot mismatch test module
+   mod slot_mismatch_test;
```

Create a new `slot_mismatch_test.rs` file under the cw-ics08-wasm-eth/src/test directory and run the `test_slot_mismatch()` test:

```
use alloy_primitives::{Bytes, B256};
use cosmwasm_std::{
    coins,
    testing::{message_info, mock_env},
    Binary,
};
use ethereum_light_client::{
    client_state::ClientState as EthClientState,
    consensus_state::ConsensusState as EthConsensusState,
    header::{ActiveSyncCommittee, Header, TrustedSyncCommittee},
    update::update_consensus_state,
};
use ethereum_types::consensus::{
    fork::ForkParameters,
    light_client_header::{LightClientHeader, LightClientUpdate},
    sync_committee::{SyncAggregate, SyncCommittee},
};

use crate::{
    contract::instantiate,
    msg::{InstantiateMsg, UpdateStateMsg},
    state::{get_eth_client_state, get_eth_consensus_state},
    sudo::update_state,
    test::mk_deps,
};

// Helper function to create a mock consensus state with a specific slot
fn create_mock_consensus_state(slot: u64) -> EthConsensusState {
    EthConsensusState {
        slot,
        state_root: [0; 32].into(),
        storage_root: [0; 32].into(),
        timestamp: 1000 + slot,
        current_sync_committee: [1; 48].into(),
        next_sync_committee: None,
    }
}

// Helper function to create a mock client state
fn create_mock_client_state(latest_slot: u64) -> EthClientState {
    EthClientState {
        latest_slot,
        chain_id: 1,
        slots_per_epoch: 32,
        epochs_per_sync_committee_period: 256,
        seconds_per_slot: 12,
        genesis_time: 1606824023,
        genesis_validators_root: [0; 32].into(),
        fork_parameters: ForkParameters::default(),
        ibc_contract_address: [0; 20].into(),
        min_sync_committee_participants: 10,
        ibc_commitment_slot: B256::default().into(),
        is_frozen: false,
    }
}

// Helper function to create a header with sync committee update but smaller slot
fn create_sync_committee_update_header(trusted_slot: u64, header_slot: u64) ->
↪  Header {
    // Create basic sync committee data
    let sync_committee = SyncCommittee {
        aggregate_pubkey: [2; 48].into(),
        pubkeys: vec![[3; 48].into(); 512],
    };

    // Create new sync committee for the update
    let next_sync_committee = SyncCommittee {
        aggregate_pubkey: [4; 48].into(),
        pubkeys: vec![[5; 48].into(); 512],
    };

    // Create execution and beacon headers
    let beacon = LightClientHeader {
        beacon: ethereum_types::consensus::light_client_header::BeaconBlockHeader {
            slot: header_slot,
            proposer_index: 0,
            parent_root: [0; 32].into(),
            state_root: [0; 32].into(),
            body_root: [0; 32].into(),
        },
        execution:
         ↪  ethereum_types::consensus::light_client_header::ExecutionPayloadHeader {
            parent_hash: [0; 32].into(),
            fee_recipient: [0; 20].into(),
            state_root: [0; 32].into(),
            receipts_root: [0; 32].into(),
            logs_bloom: [0; 256].into(),
            prev_randao: [0; 32].into(),
            block_number: 0,
            gas_limit: 0,
            gas_used: 0,
            timestamp: 1000 + header_slot,
            extra_data: vec![].into(),
            base_fee_per_gas: B256::default().into(),
            block_hash: [0; 32].into(),
            transactions_root: [0; 32].into(),
            withdrawals_root: [0; 32].into(),
            blob_gas_used: 0,
            excess_blob_gas: 0,
        },
        execution_branch: [[0; 32].into(); 4],
    };

    // Finalized header
    let finalized = LightClientHeader {
        beacon: ethereum_types::consensus::light_client_header::BeaconBlockHeader {
            slot: header_slot - 1,
            proposer_index: 0,
            parent_root: [0; 32].into(),
            state_root: [0; 32].into(),
            body_root: [0; 32].into(),
        },
        execution:
         ↪  ethereum_types::consensus::light_client_header::ExecutionPayloadHeader {
            parent_hash: [0; 32].into(),
            fee_recipient: [0; 20].into(),
            state_root: [0; 32].into(),
            receipts_root: [0; 32].into(),
            logs_bloom: [0; 256].into(),
            prev_randao: [0; 32].into(),
            block_number: 0,
            gas_limit: 0,
            gas_used: 0,
            timestamp: 1000 + header_slot - 1,
            extra_data: vec![].into(),
            base_fee_per_gas: B256::default().into(),
            block_hash: [0; 32].into(),
            transactions_root: [0; 32].into(),
            withdrawals_root: [0; 32].into(),
            blob_gas_used: 0,
            excess_blob_gas: 0,
```

```rust
        },
        execution_branch: [[0; 32].into(); 4],
    };

    // Create aggregate with sufficient participants
    let sync_aggregate = SyncAggregate {
        sync_committee_bits: Bytes::from_iter(vec![0xFF; 64]),
        sync_committee_signature: [0; 96].into(),
    };

    Header {
        trusted_sync_committee: TrustedSyncCommittee {
            trusted_slot,
            sync_committee: ActiveSyncCommittee::Current(sync_committee),
        },
        consensus_update: LightClientUpdate {
            attested_header: beacon,
            next_sync_committee: Some(next_sync_committee),
            next_sync_committee_branch: Some([[0; 32].into(); 5]),
            finalized_header: finalized,
            finality_branch: [[0; 32].into(); 6],
            sync_aggregate,
            signature_slot: header_slot + 1,
        },
        account_update: Default::default(),
    }
}

#[test]
fn test_slot_mismatch() {
    // Setup: Create test environment
    let mut deps = mk_deps();
    let creator = deps.api.addr_make("creator");
    let info = message_info(&creator, &coins(1, "uatom"));

    // 1. Define our test slots
    let current_consensus_slot = 100;
    let trusted_slot = 90;
    let header_slot = 90;

    // 2. Create initial client state and consensus state
    let client_state = create_mock_client_state(current_consensus_slot);
    let consensus_state = create_mock_consensus_state(current_consensus_slot);

    // 3. Initialize the contract with these states
    let client_state_bz = serde_json::to_vec(&client_state).unwrap();
    let consensus_state_bz = serde_json::to_vec(&consensus_state).unwrap();

    let msg = InstantiateMsg {
        client_state: Binary::from(client_state_bz),
        consensus_state: Binary::from(consensus_state_bz),
        checksum: b"checksum".into(),
    };

    instantiate(deps.as_mut(), mock_env(), info, msg).unwrap();

    // 4. Create a header with sync committee update but smaller slot
    let header = create_sync_committee_update_header(trusted_slot, header_slot);

    // 5. Call update_consensus_state directly to examine return values
    let result = update_consensus_state(
        consensus_state.clone(),
        client_state.clone(),
        header.clone(),
    )
    .unwrap();

    let (updated_slot, new_consensus_state, _updated_client_state) = result;

    // 6. Verify the mismatch - updated_slot is smaller than the consensus state's
     ↪  internal slot
    println!("Updated slot returned: {}", updated_slot);
    println!(
        "Consensus state internal slot: {}",
        new_consensus_state.slot
    );
    assert_eq!(updated_slot, header_slot);
    assert_eq!(new_consensus_state.slot, current_consensus_slot); // Slot should
     ↪  remain unchanged
    assert!(new_consensus_state.next_sync_committee.is_some()); // But sync
     ↪  committee should be updated

    // 7. Now use the contract's update_state function to see storage impact
    let header_bz = serde_json::to_vec(&header).unwrap();
    let update_msg = UpdateStateMsg {
        client_message: Binary::from(header_bz),
    };
    update_state(deps.as_mut(), update_msg).unwrap();

    // 8. Check what's stored under each key by retrieving from the store directly
    let _eth_client_state = get_eth_client_state(&deps.storage).unwrap();
    let state_at_key_90 = get_eth_consensus_state(&deps.storage, 90).unwrap();
    let _state_at_key_100 = get_eth_consensus_state(&deps.storage, 100).unwrap();

    // 9. Verify the vulnerability: consensus state with slot=100 stored at key=90
    assert_eq!(state_at_key_90.slot, current_consensus_slot);
    assert!(state_at_key_90.next_sync_committee.is_some());
    assert_eq!(state_at_key_90.next_sync_committee.is_some(), true);

    // 10. Verify the original state at key=100 still has no sync committee update
        ↪  since this is a real issue if state_at_key_100 exists and doesn't have the
        ↪  committee update
```

```
    if let Ok(original_state) = get_eth_consensus_state(&deps.storage, 100) {
        assert!(original_state.next_sync_committee.is_none());
        println!("BUG PROVEN: Consensus state with slot 100 is stored under key
         ↪  90");
        println!("Original state at key 100 was not updated with new sync
         ↪  committee");
    }
}
```

Output:

```
running 1 test
test test::slot_mismatch_test::test_slot_mismatch ... ok

successes:

---- test::slot_mismatch_test::test_slot_mismatch stdout ----
Updated slot returned: 90
Consensus state internal slot: 100
BUG PROVEN: Consensus state with slot 100 is stored under key 90
Original state at key 100 was not updated with new sync committee
```

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/packages/ethereum/ethereum-light-client/src/verify.rs#L219-L234

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/packages/ethereum/ethereum-light-client/src/update.rs#L12-L88

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/programs/cw-ics08-wasm-eth/src/sudo.rs#L105-L107

Tool Used

Manual Review

Recommendation

Fix how updates work in respect to slots, so we correctly store the consensus state under the correct key while taking the consensus state's internal slot into account.

Issue M-1: `ICS26Router::recvPacket` can be DoSed via gas griefing attack, resulting in IBC unreliability

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/153

Summary

Function ICS26Router.sol::recvPacket is susceptible to gas griefing attack: under certain conditions, an attacker may call the function with little gas such that the inner call to application's `onRecvPacket` fails with `OutOfGas` exception, but `recvPacket` as a whole succeeds, writing packet acknowledgement. As a result, the IBC application is not able to process the received packet, and the funds will be returned to the user on the source chain, not reaching the destination. Cross-chain packets can be DoSed selectively: e.g. the high-volume packets, or the trading (arbitrage) packets. This makes IBC cross-chain delivery unreliable, breaking the core contract invariant of timely (or even eventual) message delivery, and may lead to secondary financial losses for users e.g. due to arbitrage packets missing the arbitrage time window.

Vulnerability Detail

In `ICS26Router.sol::recvPacket` we observe the following fragment:

```
try getIBCApp(payload.destPort).onRecvPacket(
    IIBCAppCallbacks.OnRecvPacketCallback({
        sourceClient: msg_.packet.sourceClient,
        destinationClient: msg_.packet.destClient,
        sequence: msg_.packet.sequence,
        payload: payload,
        relayer: _msgSender()
    })
) returns (bytes memory ack) {
    require(ack.length != 0, IBCAsyncAcknowledgementNotSupported());
    require(keccak256(ack) != ICS24Host.KECCAK256_UNIVERSAL_ERROR_ACK,
    ↪   IBCErrorUniversalAcknowledgement());
    acks[0] = ack;
} catch (bytes memory reason) {
    emit IBCAppRecvPacketCallbackError(reason);
    acks[0] = ICS24Host.UNIVERSAL_ERROR_ACK;
}


writeAcknowledgement(msg_.packet, acks);
emit RecvPacket(msg_.packet);
```

In the above, we see that any exception occurring in IBC App's `onRecvPacket` is caught and effectively ignored, with acknowledgement being written despite it. The problem with the above is that the error may be forced by the attacker via supplying too little gas, such that:

1. The call to IBC App's `onRecvPacket` fails with `OutOfGas` error
2. The remaining portion of `recvPacket` succeeds, in particular writing the package acknowledgement.

For external calls, due to EIP-150, 63/64 of the total gas remaining is supplied to the called function. For IBC applications with relatively heavy computations, it may happen that the attacker is able to supply the amount of gas when calling `recvPacket` such that above scenario happens, i.e. application's `onRecvPacket` fails with 63/64 of gas, but the remaining 1/64 of gas is sufficient for the acknowledgement to be written.

Impact

The main impact of the vulnerabilty described here is **selective DoS of cross-chain IBC packets**. The only precondition for the execution of the attack is that the `onRecvPacket` contains heavy enough computations, such that its computation complexity is 63x of the computational complexity of writing an IBC acknowledgement, which is relatively low. IBC Eureka is designed to be a general-purpose cross-chain messaging solution: arbitrary IBC apps can be permissionlessly added via the call to ICS26Router::addIBCApp; thus IBC apps satisfying the precondition are bound to exist.

As outlined in the IBC vs CCIP technical comparison, the main competitive advantage of IBC is that it's 3-15x faster than CCIP. Combined with the IBC generality described above, this opens for IBC Eureka users a wide range of potential applications, including complex financial trading schemes, such as fast cross-chain arbitrage. The vulnerability described here has the potential to degrade main IBC competitive advantages: speed & reliability. Attackers gain the ability to selectively DoS cross-chain transactions to either gain financial advantage, or harm competitors, or both.

Citing from Sherlock's standards:

> **V. How to identify a medium issue:**
> - Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss must be relevant to the affected party.
> - Breaks core contract functionality, rendering the contract useless or leading to loss of funds that's relevant to the affected party.

Execution of the attack has the following effects:

- it breaks core contract functionality: an attacker can selectively (and repeatedly) DoS specific IBC apps / specific users, breaking the invariant of timely (and even eventual) message delivery.
- it leads to loss of funds that's relevant to the affected party: each attack execution delays eventual message delivery by several minutes. Considering that resubmitted transactions may be DoSed again and again, the delay becomes

unbounded, and untimely delivery may in many cases lead to secondary financial losses, such as missing an arbitrage opportunity.

Thus, the combined impact of the finding is rated as **Medium**.

Even though likelihood doesn't play a role on Sherlock in general, it's still worthwhile to consider it. The only possibility for the attack to fail is if some honest relayer submits the `RecvPacket` transaction faster than the attacker. Relayers are in general not incentivized to financially compete for the speed of execution: they optimize their hardware, but there is no incentive to e.g. overpay for gas. The attacker, on the other hand, may overpay for gas, which guarantees the success of the attack transaction. Moreover, an attacker can select transactions (a small percentage), while relayers have to relay all transactions, which in general prevents relayers from effectively counteracting such an attack. Thus, the likelihood of the attack is rated as **High**.

Proof of Concept

In the PoC we slightly change the existing test setup, such that:

- Contract `TestERC20` performs more computation on transfers
- Auxiliary function `_receiveICS20Transfer` calls `recvPacket` with limited gas

The PoC demonstrates that during the execution of the originally present test function `test_success_receiveICS20PacketWithSourceDenom` the following happens:

- Function `recvPacket` as a whole succeds, in particular writing the package acknowledgement;
- The inner call to application's `onRecvPacket`fails with `OutOfGas`;
- The supposed changes to the user balance doesn't happen, i.e. the funds are no transferred as expected.

1. Apply the below diff

```
diff --git a/solidity-ibc-eureka/test/solidity-ibc/IntegrationTest.t.sol
 ↪  b/solidity-ibc-eureka/test/solidity-ibc/IntegrationTest.t.sol
index 9ed3220..d8d2f52 100644
--- a/solidity-ibc-eureka/test/solidity-ibc/IntegrationTest.t.sol
+++ b/solidity-ibc-eureka/test/solidity-ibc/IntegrationTest.t.sol
@@ -430,11 +430,7 @@ contract IntegrationTest is Test, DeployPermit2,
 ↪  PermitSignature {
          (,, IICS26RouterMsgs.Packet memory recvPacket) =
              _receiveICS20Transfer(defaultReceiverStr, defaultSenderStr,
               ↪  receivedDenom);

-         // acknowledgement should be written
-         bytes32 storedAck = ics26Router.getCommitment(
-
 ↪  ICS24Host.packetAcknowledgementCommitmentKeyCalldata(recvPacket.destClient,
 ↪  recvPacket.sequence)
```

```
-        );
-        assertEq(storedAck,
 ↪  ICS24Host.packetAcknowledgementCommitmentBytes32(singleSuccessAck));
+        // Disable checking the acknowledgement to illustrate balance mismatch

         // check balances after receiving back
         uint256 senderBalanceAfterReceive = erc20.balanceOf(defaultSender);
@@ -1141,7 +1137,8 @@ contract IntegrationTest is Test, DeployPermit2,
 ↪  PermitSignature {
         vm.expectEmit();
         emit IICS26Router.RecvPacket(receivePacket);

-        ics26Router.recvPacket(
+        // Exploit: execute `recvPacket` with limited gas
+        ics26Router.recvPacket{gas: 900000}(
             IICS26RouterMsgs.MsgRecvPacket({
                 packet: receivePacket,
                 proofCommitment: bytes("doesntmatter"), // dummy client will accept
diff --git a/solidity-ibc-eureka/test/solidity-ibc/mocks/TestERC20.sol
 ↪  b/solidity-ibc-eureka/test/solidity-ibc/mocks/TestERC20.sol
index afb2bf5..6d5b893 100644
--- a/solidity-ibc-eureka/test/solidity-ibc/mocks/TestERC20.sol
+++ b/solidity-ibc-eureka/test/solidity-ibc/mocks/TestERC20.sol
@@ -11,6 +11,16 @@ contract TestERC20 is ERC20 {
     function mint(address _to, uint256 _amount) external {
         _mint(_to, _amount);
     }
+
+    function _update(address from, address to, uint256 value) internal virtual
 ↪  override {
+        // Simulating some computation in the IBC App
+        for(uint i=0; i<20000; i++) {
+            uint x;
+            x = x*i;
+        }
+        super._update(from, to, value);
+    }
+
 }

 contract MalfunctioningERC20 is TestERC20 {
```

2. Execute the test

From `solidity-ibc-eureka`, execute the test via `forge test -vvvv --match-test test_success_receiveICS20PacketWithSourceDenom`. Observe the below output (only the last portion is shown):

```
             [774723] ICS20Transfer::onRecvPacket(OnRecvPacketCallback({
↪   sourceClient: "42-dummy-01", destinationClient: "client-0", sequence: 0,
↪   payload: Payload({ sourcePort: "transfer", destPort: "transfer", version:
↪   "ics20-1", encoding: "application/x-solidity-abi", value:
↪   0x0000000000000000000000000000000000000000000000000000000000000020000000000⌟
↪   00000000000000000000000000000000000000000000000000000a000000000000000000000⌟
↪   000000000000000000000000000000000000000001000000000000000000000000000000000⌟
↪   000000000000000000000000000000160000000000000000000000000000000000000000000⌟
↪   0000000de0b6b3a76400000000000000000000000000000000000000000000000000000000⌟
↪   000000001c000000000000000000000000000000000000000000000000000000000000003f⌟
↪   7472616e736665722f34322d64756d6d792d30312f307861346164346636386430623931636⌟
↪   664313936383763383831653530663361303032343238323863000000000000000000000000⌟
↪   00000000000000000000000000000000000000002a307862366434383035626636393433633⌟
↪   538373563306337623637656461323462326264616362663665000000000000000000000000⌟
↪   000000000000000000000000000000000000000000000000000000000000000000000000000⌟
↪   00000002a307863643137323266333934376465663463663134343637396461333963346333⌟
↪   326264633335363831000000000000000000000000000000000000000000000000000000000⌟
↪   00000000000000000000000000000000000000000000000046d656d6f0000000000000000⌟
↪   0000000000000000000000000000000000000000 }), relayer:
↪   0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496 })) [delegatecall]
                [739657] BeaconProxy::fallback(TestERC20:
↪   [0xA4AD4f68d0b91CFD19687c881e50f3A00242828c], sender:
↪   [0xCD1722F3947DEf4Cf144679Da39c4c32BDC35681], 1000000000000000000 [1e18])
                    [273] UpgradeableBeacon::implementation() [staticcall]
                       ← [Return] Escrow:
↪   [0x2e234DAe75C793f67A35089C9d99245E1C58470b]
                    [738815] Escrow::send(TestERC20:
↪   [0xA4AD4f68d0b91CFD19687c881e50f3A00242828c], sender:
↪   [0xCD1722F3947DEf4Cf144679Da39c4c32BDC35681], 1000000000000000000 [1e18])
↪   [delegatecall]
                       [735826] TestERC20::transfer(sender:
↪   [0xCD1722F3947DEf4Cf144679Da39c4c32BDC35681], 1000000000000000000 [1e18])
                           ← [OutOfGas] EvmError: OutOfGas
                       ← [Revert] EvmError: Revert
                    ← [Revert] EvmError: Revert
                ← [Revert] EvmError: Revert
             ← [Revert] EvmError: Revert
         emit IBCAppRecvPacketCallbackError(reason: 0x)
         [72] PRECOMPILES::sha256(0x4774d4a575993f963b1c06573736617a457abef8589⌟
↪   178db8d10c94b4ab511ab) [staticcall]
             ← [Return]
↪   0x809a755691ae452c1cc3132604327e29ae131940e72094b4f4922bf2add35e1d
         [84] PRECOMPILES::sha256(0x02809a755691ae452c1cc3132604327e29ae131940e⌟
↪   72094b4f4922bf2add35e1d) [staticcall]
             ← [Return]
↪   0xe2fb30dfbf7abdeaca82d426534d2b3a9d5444dd2a87fa16d38b77ba1a13ced7
```

```
        emit WriteAcknowledgement(packet: Packet({ sequence: 0, sourceClient:
↪   "42-dummy-01", destClient: "client-0", timeoutTimestamp: 1680221800
↪   [1.68e9], payloads: [Payload({ sourcePort: "transfer", destPort:
↪   "transfer", version: "ics20-1", encoding: "application/x-solidity-abi",
↪   value: 0x000000000000000000000000000000000000000000000000000000000000002000┘
↪   000000000000000000000000000000000000000000000000000000000000a00000000000000┘
↪   000000000000000000000000000000000000000000000000100000000000000000000000000┘
↪   000000000000000000000000000000000000016000000000000000000000000000000000000┘
↪   00000000000000de0b6b3a76400000000000000000000000000000000000000000000000000┘
↪   0000000000000001c0000000000000000000000000000000000000000000000000000000000┘
↪   000003f7472616e736665722f34322d64756d6d792d30312f30786134616434663638643062┘
↪   393163666431393638376338383165353066336130303234323832386300000000000000000┘
↪   000000000000000000000000000000000000000000000002a30786236643438303562663639┘
↪   343363353837356330633762363765646132346232626461636266366500000000000000000┘
↪   000000000000000000000000000000000000000000000000000000000000000000000000000┘
↪   000000000000002a30786364313732326633393437646566346366313434363739646133396┘
↪   334633332626463333536383100000000000000000000000000000000000000000000000000┘
↪   00000000000000000000000000000000000000000000000000000000046d656d6f000000000┘
↪   00000000000000000000000000000000000000000000000 })] }), acknowledgements:
↪   [0x4774d4a575993f963b1c06573736617a457abef8589178db8d10c94b4ab511ab])
        emit RecvPacket(packet: Packet({ sequence: 0, sourceClient:
↪   "42-dummy-01", destClient: "client-0", timeoutTimestamp: 1680221800
↪   [1.68e9], payloads: [Payload({ sourcePort: "transfer", destPort:
↪   "transfer", version: "ics20-1", encoding: "application/x-solidity-abi",
↪   value: 0x000000000000000000000000000000000000000000000000000000000000002000┘
↪   000000000000000000000000000000000000000000000000000000000000a00000000000000┘
↪   000000000000000000000000000000000000000000000000100000000000000000000000000┘
↪   000000000000000000000000000000000000016000000000000000000000000000000000000┘
↪   00000000000000de0b6b3a76400000000000000000000000000000000000000000000000000┘
↪   0000000000000001c0000000000000000000000000000000000000000000000000000000000┘
↪   000003f7472616e736665722f34322d64756d6d792d30312f30786134616434663638643062┘
↪   393163666431393638376338383165353066336130303234323832386300000000000000000┘
↪   000000000000000000000000000000000000000000000002a30786236643438303562663639┘
↪   343363353837356330633762363765646132346232626461636266366500000000000000000┘
↪   000000000000000000000000000000000000000000000000000000000000000000000000000┘
↪   000000000000002a30786364313732326633393437646566346366313434363739646133396┘
↪   334633332626463333536383100000000000000000000000000000000000000000000000000┘
↪   00000000000000000000000000000000000000000000000000000000046d656d6f000000000┘
↪   00000000000000000000000000000000000000000000000 })] }))
        ← [Return]
     ← [Return]
  [3578] ERC1967Proxy::fallback("0xa4ad4f68d0b91cfd19687c881e50f3a00242828c")
↪   [staticcall]
     [3258] ICS20Transfer::ibcERC20Contract("0xa4ad4f68d0b91cfd19687c881e50f3a0┘
↪   0242828c") [delegatecall]
        ← [Revert]
↪   ICS20DenomNotFound("0xa4ad4f68d0b91cfd19687c881e50f3a00242828c")
     ← [Revert] ICS20DenomNotFound("0xa4ad4f68d0b91cfd19687c881e50f3a00242828c")
  [513] TestERC20::balanceOf(sender:
↪   [0xCD1722F3947DEf4Cf144679Da39c4c32BDC35681]) [staticcall]
```

```
        ← [Return] 0
      [0] VM::assertEq(0, 1000000000000000000 [1e18]) [staticcall]
        ← [Revert] assertion failed: 0 != 1000000000000000000
      ← [Revert] assertion failed: 0 != 1000000000000000000

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 10.55ms (8.44ms
↪  CPU time)

Ran 1 test suite in 1.86s (10.55ms CPU time): 0 tests passed, 1 failed, 0 skipped
↪  (1 total tests)

Failing tests:
Encountered 1 failing test in
↪  test/solidity-ibc/IntegrationTest.t.sol:IntegrationTest
[FAIL: assertion failed: 0 != 1000000000000000000]
↪  test_success_receiveICS20PacketWithSourceDenom() (gas: 4322267)
```

As can be seen, `recvPacket` as a whole succeeds, in particular writing the packet acknowledgement, but the inner call to `onRecvPacket` fails; as a result the expected amount is not transferred to the user.

Tool Used

Manual Review; Foundry

Recommendation

Thanks to the obligatory registration of IBC apps on the destination chain, it's easy to counteract the attack via a small refactoring. We recommend to do the following:

- Extend the call to ICS26Router::addIBCApp with the additional parameter, `minGas`, specifying the minimum amount of gas with which `onRecvPacket` should be called. Store this information, and make it publicly available for relayers.
- During the call to `recvPacket`, control the minimum amount of gas supplied to `onRecvPacket`; see e.g. how it's done in the OptimismPortal2 contract. The SafeCall library won't work though, as data has to be received from `onRecvPacket`, so we recommend to employ the ExcessivelySafeCall library instead.

Issue M-2: Rate Limit for transfer channels can be exhausted, leading to DoS

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/155

Summary

Anyone can DOS the transfer channel, causing a failure of new cross chain transfers and griefing other users from receiving their refunds, if their transfer attempt failed/timed out.

Vulnerability Detail

Anyone can DOS new cross chain transfers when the ratelimit is set and grief other users, by sending out the set limit at the start of every `RATE_LIMIT_PERIOD`, this is achievable since we don't expect the admin with the `RATE_LIMITER_ROLE` to increase the limit after crossing as this is intended to protect the channel incase of a black swan event on the sending end and if the limit is increased, nothing stops the malicious user from back running this increment of the limit with another transfer.

To go into more details, when sending tokens from the escrow we assert and update the ratelimit if set:

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/utils/Escrow.sol#L48-L52

```
    function _assertAndUpdateRateLimit(address token, uint256 amount) internal {
        RateLimitStorage storage $ = _getRateLimitStorage();

        uint256 rateLimit = $._rateLimits[token];
        if (rateLimit == 0) {
            return;
        }

        bytes32 dailyTokenKey = _getDailyTokenKey(token);
        uint256 usage = $._dailyUsage[dailyTokenKey] + amount;
|>        require(usage <= rateLimit, RateLimitExceeded(rateLimit, usage));

        $._dailyUsage[dailyTokenKey] = usage;
    }
```

This is meant to block more than a set "limit" per `RATE_LIMIT_PERIOD`, however when sending we never charge any fees, asides the relaying cost which then means that any user can just at the beginning of each day send the limit while ensuring the packet timesout, which is easily achievable as timeouts are accepted to be just 1 seconds in the future, post timing out the packet, the refunds would be processed for this user which would then have limit hit for that period.

Impact

Transfer channel can easily be blocked when ratelimit is set.

This is even more tricky cause an attacker doing this for a `RATE_LIMIT_PERIOD` causes two DOSs:

- Channel users cannot receive fresh transfers using the channel, since we attempt to send tokens from the escrow in OnRecvPacket():

```
    function onRecvPacket(OnRecvPacketCallback calldata msg_) external onlyRouter
    ↪  nonReentrant  whenNotPaused returns (bytes memory) {
    {
        // ..snip

        // transfer the tokens to the receiver
        escrow.send(IERC20(erc20Address), receiver, packetData.amount);
}
```

- Also users who had previously attempted a transfer but this failed/timed out would have their funds stuck all through the remainder of the `RATE_LIMIT_PERIOD`, cause we would always revert in `onAcknowledgementPacket` and `onTimeoutPacket` when attempting to refund the assets since we pass the set ratelimit:

```
function onTimeoutPacket(OnTimeoutPacketCallback calldata msg_) external onlyRouter
↪  nonReentrant whenNotPaused {
    IICS20TransferMsgs.FungibleTokenPacketData memory packetData =
        abi.decode(msg_.payload.value,
         ↪  (IICS20TransferMsgs.FungibleTokenPacketData));
    _refundTokens(msg_.payload.sourcePort, msg_.sourceClient, packetData);
}
```

```


    function _refundTokens(
        string calldata sourcePort,
        string calldata sourceClient,
        IICS20TransferMsgs.FungibleTokenPacketData memory packetData
    )
        private
    {

        // ..snip
        ICS20TransferStorage storage $ = _getICS20TransferStorage();
        IEscrow escrow = $.escrows[sourceClient];
        require(address(escrow) != address(0),
         ↪  IICS20Errors.ICS20EscrowNotFound(sourceClient));

|>        escrow.send(IERC20(erc20Address), refundee, packetData.amount);
    }
```

Code Snippet

```
function _assertAndUpdateRateLimit(address token, uint256 amount) internal {
    RateLimitStorage storage $ = _getRateLimitStorage();

    uint256 rateLimit = $._rateLimits[token];
    if (rateLimit == 0) {
        return;
    }

    bytes32 dailyTokenKey = _getDailyTokenKey(token);
    uint256 usage = $._dailyUsage[dailyTokenKey] + amount;
    require(usage <= rateLimit, RateLimitExceeded(rateLimit, usage));// <|

    $._dailyUsage[dailyTokenKey] = usage;
}
```

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/ICS20Transfer.sol#L318-L358

Tool Used

Manual Review

Recommendation

Integrate a net in/outflow to really take into account how much was received/sent via the channel, this is similar to what's on Osmosis, where they have a specific `flow` struct and track the in/out flow of assets:

https://github.com/osmosis-labs/osmosis/blob/7b1a78d397b632247fe83f51867f319adf3a858c/x/ibc-rate-limit/contracts/rate-limiter/src/state/flow.rs#L29-L34

```
pub struct Flow {
    pub inflow: Uint256,
    pub outflow: Uint256,
    pub period_end: Timestamp,
}
```

And these get checked in `balance()` via the `exceeds` function during transfers:

```
    pub fn balance(&self) -> (Uint256, Uint256) {
        (
            self.inflow.saturating_sub(self.outflow),
            self.outflow.saturating_sub(self.inflow),
        )
    }

    /// checks if the flow, in the current state, has exceeded a max allowance
    pub fn exceeds(&self, direction: &FlowType, max_inflow: Uint256, max_outflow:
    ↪  Uint256) -> bool {
        let (balance_in, balance_out) = self.balance();
        match direction {
            FlowType::In => balance_in > max_inflow,
            FlowType::Out => balance_out > max_outflow,
        }
    }


    /// Checks if a transfer is allowed and updates the data structures
    /// accordingly.
    ///
    /// If the transfer is not allowed, it will return a RateLimitExceeded error.
    ///
    /// Otherwise it will return a RateLimitResponse with the updated data
    ↪  structures
    pub fn allow_transfer(
        &mut self,
        path: &Path,
        direction: &FlowType,
        funds: Uint256,
        channel_value: Uint256,
        now: Timestamp,
    ) -> Result<Self, ContractError> {
        // ..snip
        let (max_in, max_out) = self.quota.capacity();
        // Return the effects of applying the transfer or an error.
        match self.flow.exceeds(direction, max_in, max_out) {
|>            true => Err(ContractError::RateLimitExceded {
                channel: path.channel.to_string(),
                denom: path.denom.to_string(),
                amount: funds,
                quota_name: self.quota.name.to_string(),
                used: initial_flow,
                max: self.quota.capacity_on(direction),
                reset: self.flow.period_end,
            }),
            false => Ok(RateLimit {
                quota: self.quota.clone(), // Cloning here because self.quota.name
                ↪  (String) does not allow us to implement Copy
                flow: self.flow, // We can Copy flow, so this is slightly more
                ↪  efficient than cloning the whole RateLimit
            }),
        }
```

So in our case we could then track the flow, i.e in the case a refund was made, it would be tracked negatively against the flow.

Discussion

srdtrk

I think there is enough disincentives not to perform this attack since

1. The rate limits are going to be high
2. Users are charged off protocol for relaying
3. Ethereum gas fees are relatively high
4. We don't increase the rate limit on send packet.

However, it is true that it might be relatively easy to DOS the channel for a different reason since you can keep transferring and timing out the same packet, although it would be costly. I think we should count the net outflows rather than absolutes to address this.

srdtrk

addressed here https://github.com/cosmos/solidity-ibc-eureka/pull/342

Issue M-3: Missing check to ensure client state is active in Solidity implementation

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/158

Description

In IBC-go's `sendPacket` flow, the following check is performed:

```
if status := k.ClientKeeper.GetClientStatus(ctx, sourceClient); status !=
↪  exported.Active {
    return 0, "", errorsmod.Wrapf(clienttypes.ErrClientNotActive, "client (%s)
    ↪  status is %s", sourceClient, status)
}
```

This validation ensures that the client status is `Active` before processing packets.

However, there is no equivalent validation on the Solidity side within the `sendPacket` flow.

This breaks a core invariant as only `Active` clients are allowed to process packets as mentioned in the documentation:

- An `Active` status indicates that clients are allowed to process packets.
- A `Frozen` status indicates that misbehaviour was detected in the counterparty chain and the client is not allowed to be used.
- An `Expired` status indicates that a client is not allowed to be used because it was not updated for longer than the trusting period.
- An `Unknown` status indicates that there was an error in determining the status of a client.

But as of now, `Frozen` `Expired` or `Unknown` clients will be able send packets which should not be allowed.

Impact

Since `sendPacket` is called inside the `sendTransfer/PermitsendTransfer` flow this can even cause more damage as funds will be involved.

Furthermore, funds could be lost due in the following scenario:

- `sendTransfer` is performed with a `Frozen` client status
- `sendPacket` passes due to the missing status check
- the packet fails for some reason and a refund attempt is made

- This will however fail due to the call made to `membership` which performs the `notFrozen` check

```
function membership(ILightClientMsgs.MsgMembership calldata msgMembership)
        public
        notFrozen
        returns (uint256 timestamp)
    {
```

- Because of this the funds are now essentially stuck with no way to refund them

In the other case where the packet goes through succesfully, this essentially breaks a core invariant allowing invalid clients to pass packets.

Regardless, the impact is quite severe.

We would classify this as a **Critical** issue as per the following reason following the severity matrix

- Likelihood: Likely. This can happen anytime a client is anything other than `Active`
- Impact: Catastrophic: Loss of funds is highly likely in this scenario, as the user has no means of recovering them, as indicated in the finding. The only case where funds wouldn't be lost is if the packet is successfully transmitted, though I am uncertain whether that is even possible.

Recommendation

Confirm that the client status is verified within the `sendPacket` process just like the IBC-Go implementation.

Discussion

srdtrk

I think we have decided to not address this in the contracts and mitigate it in the frontends. https://github.com/cosmos/solidity-ibc-eureka/pull/344

Issue M-4: Malicious ERC-20 contracts may fail acks/timeouts, forcing relayers to lose funds

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/164

Summary

We describe a particular ICS20 token transfer scenario, when an attacker, employing a malicious ERC20 token contract, can make the submitted by relayers `ackPacket` transactions fail deterministically and repeatedly, causing unbounded financial damage for relayers. The attack is made possible due to allowing to execute ICS20 transfers of arbitrary ERC20 token contracts on Ethereum; malicious contracts may perform arbitrary balance accounting (in particular allowing to mint more than `type(uint256).max` tokens), as well as exhibit arbitrary behavior (e.g. consume a lot of gas and revert) when called from the `ICS20Transfer` or `Escrow` contracts. Combined with the fact that `ackPacket` / `timeoutPacket` methods of `ICS26Router` contract bubble up all errors from application's `onAckPacket` / `onTimeoutPacket` callbacks, this allows attackers to completely exhaust relayer funds.

Vulnerability Detail

`ICS20Transfer.sol` contract on Ethereum allows anyone to send arbitrary native ERC20 tokens: a token denomination is just an ERC20 contract address. This makes possible the attack described below:

1. An attacker creates a malicious ERC20 contract
2. The attacker sends ICS20 transfer `type(uint256).max` from this contract (which they control) to some Cosmos chain. This succeeds, as demonstrated by the implemented
   e2e test that transfers a uint256 amount from Ethereum to Cosmos and back
3. The attacker sends another ICS20 transfer for any amount from that contract. Sending of the transaction on the Ethereum side succeeds, because the malicious contract does its own (wrong) token accounting
4. The `OnRecvPacket` callback on Cosmos side fails, when the ICS20 transfer module tries to mint additional tokens above `type(uint256).max` (the `mint` operation of the `Bank` module fails). Error acknowledgement is written on the Cosmos chain
5. Relayers pick up the error acknowledgement, and submit it on Ethereum to ICS26Router::ackPacket, which calls ICS20Transfer::onAcknowledgementPacket
6. `ICS20Transfer` tries to refund the tokens, and calls
   `escrow.send(IERC20(erc20Address), refundee, packetData.amount)`

7. The `Escrow` contract calls `token.safeTransfer(to, amount)` on the malicious ERC20 contract
8. The malicious ERC20 contract can now do several things, in particular consume all available transaction gas, and revert the call
9. The revert bubbles up, and `ICS26Router::ackPacket` reverts.

The net effect is that each relayer tries to call `ICS26Router::ackPacket`, the calls revert, and packet acknowledgement is left in the dangling (uncleared) state, forcing relayers to resubmit it, thus losing funds for gas payments.

A Note On Simulation

As this is going to be inevitably brought up, let us point out that simulation of transactions by relayers is not a valid defense against such an attack. EVM/Solidity contains enough Block and Transaction Properties that allow to fingerprint the execution environment, and make the contract behave differently when simulated and when executed on the mainnet. For a good overview of how simulations can be tricked see e.g. the post Lies, damned lies, and simulations.

Impact

There are several important considerations to mention in order to assess the impact:

1. The affected IBC App is ICS20 Transfer: the main application, which packets are going to be relayed by all relayers.
2. Malicious packets can't be easily filtered out. Relayers allow to configure filtering based on channels, and the channel will be `transfer`.
3. Attackers may also incentivize malicious transfers, if this is required by relayers. The incentive provided is calculated based on the normal course of events, when packets are delivered successfully, and don't account for the repetitive cost of submitting failing packet acknowledgements. At the same time, relayers can't afford not to submit acknowledgements, as this would break the IBC flow.
4. Relayers have hard-coded retry attempts for failed transactions: e.g. Hermes relayer has a hard-coded retry amount of 5. When a transaction fails with the configured gas, relayers try to resubmit it with the maximum amount of gas.
5. Relayers attempt to repetitively clear uncleared packets, and acknowledgement packets from the attack will remain uncleared indefinitely.
6. Relayers collect statistics & telemetry collectively per channel, so depending on the frequency of the attack execution, it may remain unnoticed for long periods of time.

For the overview of the relevant relayer configuration parameters we refer the reader to Hermes' description of the parameters.

Taken together, we can do the following calculation. Assuming 5 competing relayers, 10 packet clearing attempts, and 5 retry attempts for failing transactions, we arrive at `5 x`

`10 x 5 = 250` resubmission attempts with max gas. As transactions happen on mainnet Ethereum, and assuming a max gas configuration to equal `10`, wearriveat * *2500 combined relayer losses per attack execution**.

Thus the impact is a guaranteed loss of funds for relayers, which, depending on the frequency of execution, will either eat on their profits, or even lead to their insolvency; the impact is rated as **High**.

It's also worth noting that the impact described here materializes the following threat as described by the protocol team:

> Packet flow can be incomplete (ie packet can never be successfully acknowledged or timed out)

Tool Used

Manual Review

Recommendation

Below are a few possible strategies of mitigation that we can propose.

Make ERC20 transfers permissioned

As a first line of defense, we recommend to allow ICS20 transfers only from a set of allowed ERC20 contracts (e.g. WETH, USDC, stETH, etc.), which is configurable by the admin. This is a relatively straightforward to implement feature, which should suffice for the first release without impacting much the system usability.

Perform internal accounting of Escrow balances

`ICS20Transfer` completely trusts ERC20 contracts to perform accounting of ERC20 balances of the Escrow contracts. As demonstrated in this finding, ERC20 accounting can be arbitrary skewed for attacker's benefits. We recommend to perform internal accounting of `Escrow` contracts balances in the `ICS20Transfer` contract, and act appropriately on any deviation between internal accounting and the values reported by ERC20 contracts.

Harden the calls to ERC20 contracts from `Escrow`

The `Escrow` contract employs the method safeTransfer to interact with (untrusted) ERC20 contracts; this method bubbles up all reverts. One possible protection would be to employ the trySafeTransfer method instead, which returns `false` instead of reverting. In case of an error on the side of the ERC20 contract, this error could be recorded for later investigation, but the `ackPacket` / `timeoutPacket` flow still successfully finalized.

Make calls to `onAckPacket` / `onTimeoutPacket` safe

Currently, the calls to `onAckPacket` / `onTimeoutPacket` callbacks are unsafe, in that they completely give away the control to the called apps; in the case of this attack, it's the trusted `ICS20Transfer` app, but which then transitively gives control to the ERC20 contract. The called app is free to consume a lot of gas, or revert, or both.

We recommend to execute `onAckPacket` / `onTimeoutPacket` callbacks with the help ExcessivelySafeCall library, which allows to both control the amount of gas used, and to catch reverts.

If an application callback, when called with enough gas, still reverts, this means an error on the side of the application. In that case the error should be written for later investigation, but the packet flow should continue, in the same way it's done for the `onRecvPacket` callback.

Discussion

kuprumxyz

As per Discord discussion, and considering that the protocol team doesn't perceive relayers as protocol users, but rather as external integrators, changing severity to Medium.

Issue M-5: Trustful execution of `onAckPacket`/ `onTimeoutPacket` callbacks may interrupt IBC packet flows

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/169

Summary

During this audit we have identified two findings that represent examples of a broader problem with IBC, both at the level of the specification, and at the implementation level. The problem is the asymmetry of treating `recvPacket` transactions, and `ackPacket` / `timeoutPacket` transactions: while the former continue the packet flow despite possible application errors in `onRecvPacket` callback, the latter ones terminate the packet flow on any error in `onAckPacket` / `onTimeoutPacket` callbacks, leaving the IBC packet dangling, which interrupts IBC packet flow, and forces relayers to resubmit the packets, or to give up on delivery, leading to users not obtaining refunds. While the other two findings focused on Ethereum smart contracts, in this finding we would like to explain this broader problem which is general both for the Cosmos and the Ethereum sides of IBC.

Vulnerability Detail

When operating in the Cosmos ecosystem, IBC was functioning in the much more restricted setting than when expanding to Ethereum. In particular, on Cosmos most applications are permissioned, and thus "trusted". In the finding Malicious ERC-20 contracts may fail acks/timeouts, forcing relayers to lose funds we have described a setup when a malicious ERC20 contract may perform wrong balance accounting in order to force the IBC packet to fail on Cosmos, and then revert when the `ackPacket` transaction tries to refund tokens. This situation was not possible on Cosmos, because there balance accounting and token transfers are performed by the trusted `Bank` module, with benign behavior. In the finding ICS26Router contract doesn't protect relayers from malicious IBC apps we have described another problem which also arises from `ICS26Router` trusting the `onAckPacket` / `onTimeoutPacket` app callbacks.

The problem we describe is that IBC is designed under the assumption of benign applications, which doesn't hold when extending from Cosmos to Ethereum, and may stop to hold on Cosmos chains as well (e.g. when integrating permissionless CosmWasm contracts). Due to that assumption, `ackPacket` **and** `timeoutPacket` **transactions fail whenever the** `onAckPacket` **/** `onTimeoutPacket` **application callbacks fail, thus interrupting the normal IBC packet flow.**

The asymmetry is best illustrated on the example of Solidity IBC contracts:

- ICS26Router::recvPacket: if `onRecvPacket` fails, an error acknowledgement is written, and the transaction continues successfully

```
try getIBCApp(payload.destPort).onRecvPacket(
    IIBCAppCallbacks.OnRecvPacketCallback({
        sourceClient: msg_.packet.sourceClient,
        destinationClient: msg_.packet.destClient,
        sequence: msg_.packet.sequence,
        payload: payload,
        relayer: _msgSender()
    })
) returns (bytes memory ack) {
    require(ack.length != 0, IBCAsyncAcknowledgementNotSupported());
    require(keccak256(ack) != ICS24Host.KECCAK256_UNIVERSAL_ERROR_ACK,
    ↪  IBCErrorUniversalAcknowledgement());
    acks[0] = ack;
} catch (bytes memory reason) {
    emit IBCAppRecvPacketCallbackError(reason);
    acks[0] = ICS24Host.UNIVERSAL_ERROR_ACK;
}

writeAcknowledgement(msg_.packet, acks);
emit RecvPacket(msg_.packet);
```

- ICS26Router::ackPacket: if `onAcknowledgementPacket` fails, the transaction is reverted

```
getIBCApp(payload.sourcePort).onAcknowledgementPacket(
    IIBCAppCallbacks.OnAcknowledgementPacketCallback({
        sourceClient: msg_.packet.sourceClient,
        destinationClient: msg_.packet.destClient,
        sequence: msg_.packet.sequence,
        payload: payload,
        acknowledgement: msg_.acknowledgement,
        relayer: _msgSender()
    })
```

The same asymmetry holds in IBC-Go:

- 04-channel/v2/keeper/msg_server.go::RecvPacket: if `OnRecvPacket` fails, an error acknowledgement is written, and the transaction continues successfully

```
    res := cb.OnRecvPacket(cacheCtx, msg.Packet.SourceClient,
    ↪  msg.Packet.DestinationClient, msg.Packet.Sequence, pd, signer)


    if res.Status != types.PacketStatus_Failure {
        // successful app acknowledgement cannot equal sentinel error
        ↪  acknowledgement
        if bytes.Equal(res.GetAcknowledgement(), types.ErrorAcknowledgement[:]) {
            return nil, errorsmod.Wrapf(types.ErrInvalidAcknowledgement,
             ↪  "application acknowledgement cannot be sentinel error
             ↪  acknowledgement")
        }
        // write application state changes for asynchronous and successful
        ↪  acknowledgements
        writeFn()
        // append app acknowledgement to the overall acknowledgement
        ack.AppAcknowledgements = append(ack.AppAcknowledgements,
        ↪  res.Acknowledgement)
    } else {
        isSuccess = false
        // construct acknowledgement with single app acknowledgement that is the
        ↪  sentinel error acknowledgement
        ack = types.Acknowledgement{
            AppAcknowledgements: [][]byte{types.ErrorAcknowledgement[:]},
        }
        // Modify events in cached context to reflect unsuccessful acknowledgement
        sdkCtx.EventManager().EmitEvents(internalerrors.ConvertToErrorEvents(cacheC
        ↪  tx.EventManager().Events()))
        break
    }

...

if !isAsync {
   ....
    // Set packet acknowledgement only if the acknowledgement is not async.
    // NOTE: IBC applications modules may call the WriteAcknowledgement
    ↪  asynchronously if the
    // acknowledgement is async.
    if err := k.writeAcknowledgement(ctx, msg.Packet, ack); err != nil {
        return nil, err
    }
```

- 04-channel/v2/keeper/msg_server.go::Acknowledgement: if `OnAcknowledgementPacket` fails, the transaction is reverted

```
err := cbs.OnAcknowledgementPacket(ctx, msg.Packet.SourceClient,
↪  msg.Packet.DestinationClient, msg.Packet.Sequence, ack, pd, relayer)
if err != nil {
    return nil, errorsmod.Wrapf(err, "failed OnAcknowledgementPacket for source port
    ↪  %s, source client %s, destination client %s", pd.SourcePort,
    ↪  msg.Packet.SourceClient, msg.Packet.DestinationClient)
}
```

Surprisingly enough, the (updated) IBC specification actually treats all cases symmetrically, and requires that `recvPacket` also reverts on the callback failure:

- Processing acknowledgements states the following error condition:

    - "3. Unsuccessful payload execution: `onAcknowledgePacket(packet.sourceChannel,payload, acknowledgement) == False`"

- Timeouts has a similar error condition:
    - "3. Unsuccessful payload execution: `onTimeoutPacket(packet.sourceChannel,payload) == False`"
- Receiving packets has the same error condition:
    - "4. Unsuccessful payload execution: `onReceivePacket(..)==False`"

Impact

Trustful execution of `onAckPacket` / `onTimeoutPacket` callbacks violates the key system invariant, materializing the threat described by the protocol team in IBC Eureka Actors/Use Cases/Threats:

> Packet flow can be incomplete (ie packet can never be successfully acknowledged or timed out)

Let us consider in more details under which conditions the packet flow can be incomplete. For that it's worth noting that IBC doesn't dictate any gas limits for applications it calls. Thus, applications may consume arbitrary amounts of gas, and still be legitimate IBC applications. Now the case-by-case analysis:

- A malicious IBC app may deliberately consume any amount of gas supplied, thus reverting the transaction as `Out-Of-Gas`, or with any other reason.
- A legitimate IBC app may consume the maximum amount of gas supplied by the relayer, while trying to fulfill its task. Notice that generally there is no reliable way to distinguish between this case and the case above (for arbitrary apps).
- An IBC app encounters a permanent internal error. This is the easy case, because the app will always revert, and should be updated.
- An IBC app encounters a transient internal error. E.g. its execution may depend on the amount of transferred funds relative to the current amount of funds in the pool, or on whatever other internal dynamic conditions there are in the contract.
- An IBC app encounters a transient external error, i.e. it depends on some other contract with dynamic conditions. For example this may be a trade executed via Uniswap pool, or submission of a CCIP message.

All of the transient errors may appear and disappear from one call to another. Without explicit assumptions from IBC towards IBC apps, such behavior is legitimate.

Thus, an IBC app may fail `onAckPacket` / `onTimeoutPacket`, reverting the corresponding transaction, and leaving the packet in dangling (uncleared) state. Relayers have to provide certain delivery guarantees, without such guarantees the IBC protocol won't function. Thus, when `ackPacket` / `timeoutPacket` transactions fail, we have a choice:

- Relayers try to fulfill their duty, and resubmit the failing transactions. This leads to relayers losing funds for retry attempts, as identified in other findings submitted in this audit;
- Relayers give up on these transactions, and don't resubmit them. This leads to the IBC protocol not functioning properly, and the users not receiving refunds due to the undelivered acks/timeouts.

Both alternatives represent loss of funds scenarios, either for relayers, or for users, or both.

Tool Used

Manual Review

Recommendation

- **Create IBC Application Guidelines**, in which make sure to reflect explicit assumptions from the IBC protocol towards IBC apps:
  - IBC apps should not consume excessive amounts of gas.
  - IBC apps should not revert under almost any circumstances. The only legitimate reason to revert should be wrong data passed in the callback. For all other kinds of transient errors an app should record the receipt in its state, and acknowledge it.
    * E.g., if an IBC app was designed to call external contract (which may fail), i.e. employing a Push model; then the app should be redesigned to try that call, but if it fails: record the obligation towards the user, and allow the user to manually pull the funds; i.e. switch to the Pull model.
- **Configure per-application gas limits** upon app registration. Such limits represent a maximum amount of gas that needs to be submitted for the callback to succeed. These limits will also provide an indicator for relayers of how much a user should be charged for relaying. Without explicit gas limits relayers don't have a reliable information for the fee estimation.
- **Treat calls to `onAckPacket` / `onTimeoutPacket` symmetrically to `onRecvPacket`**, while enforcing the application gas limits:
  - If a relayer submits a transaction with too little gas: revert such transaction as illegitimate.
  - If a relayer submits a transaction with enough gas, but the application callback fails: log delivery failure, but record the packet as delivered (clear the packet), such that relayers don't resubmit it. Also possibly suspend the IBC application, because by behaving that way it violated the application guidelines.

Discussion

kuprumxyz

As per Discord discussion, and considering that the protocol team doesn't perceive relayers as protocol users, but rather as external integrators, changing severity to Medium.

Issue M-6: Frozen clients are constantly relayed to be active

Source: https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/170

Summary

`query.rs`'s `status()` always returns "Active" regardless of the client's actual state. So post misbehaviour, while the client could be frozen due to misbehaviour detection, `status()` does not check the `is_frozen` flag from the client state.

This was initially hinted to be sorted out post #164, however freezing/misbehaviour detection was added in PR #270 before this audit, but wasn't correctly reflected in `query.rs`'s `status()`.

Vulnerability Detail

`query.rs` unconditionally returns "Active" status without checking the actual state of the light client:

```
/// Gets the status of the light client
/// # Returns
/// Active status, because no other state is currently implemented
/// # Errors
/// It won't error at this point
// TODO: Implement a proper status once freezing/misbehaviour is implemented #164
pub fn status() -> Result<Binary, ContractError> {
    Ok(to_json_binary(&StatusResult {
        status: "Active".to_string(),
    })?)
}
```

The code also has a stale TODO comment indicating this needed to be fixed:

```
// TODO: Implement a proper status once freezing/misbehaviour is implemented #164
```

Meanwhile, the misbehaviour detection functionality was already implemented in-scope via PR #270, where the `misbehaviour()` function in `sudo.rs` would update the client's `is_frozen` flag to `true` if indeed misbehaving:

```
pub fn misbehaviour(
    deps: DepsMut<EthereumCustomQuery>,
    _msg: UpdateStateOnMisbehaviourMsg,
) -> Result<Binary, ContractError> {
    let mut eth_client_state = get_eth_client_state(deps.storage)?;
    eth_client_state.is_frozen = true;
    // ...
}
```

Impact

This inconsistency between the client's actual state and its reported status could lead to several security issues:

If a client is frozen due to detected misbehaviour but still reports as "Active", other components relying on the status endpoint would continue to trust it for cross-chain operations, this also completely goes against IBC's documentation on the client state:

> An `Active` status indicates that clients are allowed to process packets. A `Frozen` status indicates that misbehaviour was detected in the counterparty chain and the client is not allowed to be used.

And this is also checked when attempting to send packets from cosmos chains to ensures that the client is `Active` before allowing the send:

```
func (k *Keeper) sendPacket(..snip) {

    // ..snip
    // check that the client of counterparty chain is still active
    if status := k.ClientKeeper.GetClientStatus(ctx, sourceClient); status !=
    ↪  exported.Active {
        return 0, "", errorsmod.Wrapf(clienttypes.ErrClientNotActive, "client (%s)
        ↪  status is %s", sourceClient, status)
    }

    // ..snip

}
```

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/programs/cw-ics08-wasm-eth/src/query.rs#L136-L147

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/programs/cw-ics08-wasm-eth/src/sudo.rs#L140-L157 https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/ibc-go/modules/core/04-channel/v2/keeper/packet.go#L48-L52

Tool Used

Manual Review

Recommendation

The status function should check the client state's `is_frozen` flag and return the appropriate status, something like:

```rust
pub fn status(deps: Deps<EthereumCustomQuery>) -> Result<Binary, ContractError> {
    let client_state = get_eth_client_state(deps.storage)?;
    let status = if client_state.is_frozen {
        "Frozen"
    } else {
        "Active"
    };

    Ok(to_json_binary(&StatusResult {
        status: status.to_string(),
    })?)
}
```

Which is similar to what's in `ibc-go`:

```go
func (cs ClientState) status(
    ctx context.Context,
    clientStore storetypes.KVStore,
    cdc codec.BinaryCodec,
) exported.Status {
    if !cs.FrozenHeight.IsZero() {
        return exported.Frozen
    }
//..snip
}
```

Additionally introduce the `expired` status logic too in the former.

Issue M-7: Light client fork version mismatch during signature verification will cause DOS

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/175

Summary

The light client implementation currently uses the Deneb fork version when computing domains for signature verification, despite the Ethereum network having upgraded to the Pectra fork. This creates a critical mismatch in the signature verification process.

In the current implementation, the code calculates the fork version using:

```
let fork_version = client_state
    .fork_parameters
    .compute_fork_version(compute_epoch_at_slot(
        client_state.slots_per_epoch,
        fork_version_slot,
    ));
```

and

```
pub const fn compute_fork_version(&self, epoch: u64) -> Version {
    match epoch {
        _ if epoch >= self.deneb.epoch => self.deneb.version,
        _ if epoch >= self.capella.epoch => self.capella.version,
        _ if epoch >= self.bellatrix.epoch => self.bellatrix.version,
        _ if epoch >= self.altair.epoch => self.altair.version,
        _ => self.genesis_fork_version,
    }
}
```

However, this doesn't support pectra hardfork whie it being already live on all testnets (holesky and sepolia) and soon on ethereum mainnet.

Impact

fork version will wrongly use deneb and hence will not able to validate BLS aggregated signatures signed by committee on pectra hardfork leading to DOS.

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/packages/ethereum/ethereum-light-client/src/verify.rs#L300-L305

Tool Used

Manual Review

Recommendation

Add pectra in the `compute_fork_version`, `ForkParameters` and other required places.

Issue L-1: `ICS26Router` contract doesn't protect relayers from malicious IBC apps

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/154

Summary

`ICS26Router.sol` fully trusts IBC apps, though they can be freely registered by anyone via `addIBCApp`. This leaves relayers unprotected from possible malicious behavior on the side of IBC apps. Examples of such behavior are consuming an unbounded amount of gas, up to the block gas limit, or returning excessively large acks. As in general relayers are assumed to relay all IBC packets, absence of protection leaves them vulnerable to gas exhaustion attacks, leading to direct financial losses. All calls to at least untrusted IBC apps (those not added by the admin), but better to all, should be protected, e.g. by using the `ExcessivelySafeCall` library.

Vulnerability Detail

`ICS26Router.sol` allows anyone to register IBC applications via a call to addIBCApp; thus IBC apps are in general untrusted. At the same time the relayers have to relay all IBC packets, and to invoke the following contract methods:

- recvPacket
- ackPacket
- timeoutPacket

All these methods perform internally a call to the corresponding IBC app: `onRecvPacket`, `onAckPacket`, and `onAckPacket`, respectively. The matter is complicated by the fact that there is no a priori limit set on the amount of gas an IBC app may consume, so in order to function at all relayers have to call the above methods with large amounts of gas supplied, leaving them widely exposed to possible malicious behavior on the side of IBC apps.

Impact

There are two possible impacts: gas stealing, and DoS.

1. Gas stealing: a malicious IBC app may exploit this vulnerability by performing expensive computations for free. In this scenario, an attacker sets app a malicious IBC app, e.g. which they want to perform periodic on-chain computations (e.g. trading), and send IBC transactions to this app from the Cosmos side, making the app function for free at the the expense of relayers.

2. Denial-of-Service: a malicious IBC app may either consume all gas supplied for the transaction via computations, or, as in the case of `recvPacket`, return an excessively large acknowledgment, and thus exhaust all available gas.

Tool Used

Manual Review

Recommendation

We can recommend two approaches:

1. Make IBC apps registration permissioned

For the launch, make method `addIBCApp` permissioned, i.e. allow only an admin to register IBC applications. Notice that it still doesn't really protect relayers, as unexpected interactions may still happen also with permissioned applications; this rather reduces the probability of unexpected scenarios.

2. Protect relayers from untrusted IBC apps calls

This mitigation is similar to the mitigation of another finding (”`ICS26Router::recvPacket` can be DoSed via gas griefing attack...”), but slightly different, namely:

- In the gas griefing finding, the caller of `recvPacket` was malicious, causing the internal call to fail via supplying too little gas
- In this finding, the caller is a honest relayer, but the IBC app is malicious, causing the relayer to spend too much gas
- Another difference: this finding has to protect not only from gas exhaustion, but also from the memory expansion, so the solution provided in the previous finding has to be adapted.

Thus, we recommend the same mitigation, but with adaptations:

- Extend the call to ICS26Router::addIBCApp with the additional parameter, `minGas`, specifying the minimum amount of gas with which `onRecvPacket` should be called. Store this information, and make it publicly available for relayers. This is important, in order for relayers to be able to pro-actively determine whether they are interested to relay that particular IBC packet, as well to be able to bound the gas supplied for the transaction.
- During the call to `onRecvPacket` / `onAckPacket` / `onTimeoutPacket`, control the amount of gas supplied; see e.g. how it's done in the OptimismPortal2 contract. The SafeCall library won't work though, as data has to be received from `onRecvPacket`, so we recommend to employ the ExcessivelySafeCall library instead.

Discussion

gjermundgaraba

> at the same time the relayers have to relay all IBC packets

I believe there is a misconception of the roles and expectations of relayers. Relayers are not expected to (nor do they) pick up any and all packets for all applications (which in the past was any channels in the context of what they relay). It is also permissionless to add CosmWasm IBC applications on many chains, but it doesn't mean that relayers automatically start relaying packets from those applications.

This finding is only an issue if the relayers would have to relay all packets without discrimination. They already filter out packets today which are deemed spam or otherwise malicious in nature by verifying things like packet size and gas usage.

Issue L-2: Timeout timestamp validation inconsistency between IBC-Go and Solidity implementation

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/157

Description

There is an inconsistency in how timeout timestamps are validated between the IBC-Go and Solidity implementations.

IBC-Go Implementation:

```
timeout := time.Unix(int64(msg.TimeoutTimestamp), 0)
if timeout.Before(sdkCtx.BlockTime()) {
    return nil, errorsmod.Wrap(types.ErrTimeoutElapsed, "timeout is less than the
     ↪  current block timestamp")
}
```

The IBC-Go implementation allows the timeout timestamp to be equal to or greater than the current block timestamp.

Solidity Implementation:

```
require(
    msg_.timeoutTimestamp > block.timestamp,
     ↪  IBCInvalidTimeoutTimestamp(msg_.timeoutTimestamp, block.timestamp)
);
```

The Solidity implementation requires the timeout timestamp to be strictly greater than the current block timestamp.

Impact

This inconsistency means that a valid IBC packet with timeoutTimestamp == blockTimestamp would:

- Be accepted by the IBC-Go implementation
- Be rejected by the Solidity implementation

Since there is no financial loss but rather an inconsistency leading to different functional implementations, we consider this a Low-severity finding.

Recommendation

We recommend modifying the IBC-Go implementation to align with Solidity's handling. This is because if the timeout timestamp equals the block timestamp, it would always trigger a timeout.

Issue L-3: Unlike IBC-Go, IBC-Solidity doesn't enforce limits on `Memo` field of `FungibleTokenPacketDataV2`

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/159

Summary

`IBC-Go` implementation enforces restrictions on the `Memo` field of `FungibleTokenPacketDataV2` packet, both when the packet is being sent, and when it's being received. `IBC-Solidity`, on the other hand, doesn't validate `Memo` field upon sending or receiving. As a result, a packet which is legitimately sent from the Solidity side, but which includes a too large `Memo` field, will fail upon receiving on the Go side, resulting in user frustration, delays, the necessity to resend the packet, etc.

Vulnerability Detail

Function ibc-go/modules/apps/transfer/keeper/relay.go::OnRecvPacket calls FungibleTokenPacketDataV2::ValidateBasic, which in particular ensures that `Memo` field is doesn't exceed 32768 bytes:

```
if len(ftpd.Memo) > MaximumMemoLength {
    return errorsmod.Wrapf(ErrInvalidMemo, "memo must not exceed %d bytes",
     ↪  MaximumMemoLength)
}
```

`IBC-Solidity`, on the other hand, allows a user to send ICS-20 packets with unrestricted `Memo` field in functions ICS20Transfer.sol::sendTransfer and ICS20Transfer.sol::permitSendTransfer.

As a result, an ICS-20 transfer which is successfully accepted on the Ethereum side, will fail on the Cosmos side.

Impact

At minimum, unsuspecting users will get failed transfers, delays, and it will be necessary for them to resend failed packets. More severe consequences may result from third parties building integrations on top of IBC with dynamic `Memo` fields, which would work for small use cases, but will stop to function when reaching a certain level of complexity.

Tool Used

Manual Review

Recommendation

- Implement on the Solidity side the same level of validation for members of `FungibleTokenPacketDataV2` data structure as it exists on the Go side; this includes the aforementioned `Memo` field, but also possibly the `Receiver` field.
- Properly document the enforced restrictions in user-facing documentation.

Discussion

srdtrk

The reason why we perform these checks in CosmosSDK is because CosmosSDK chains don't consume gas properly and are prone to DOS attacks by repeatedly sending very large transactions. (This can lead to a chain halt in extreme scenarios). These are not enforced by the IBC specs.

Ethereum doesn't have this problem since it accounts for the gas properly and cannot be DOSed by large packets.

Issue L-4: freezing the client will always fail due to reverting

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/160

Description

Inside `handleSP1UpdateClientAndMembership`, the following code is executed:

```
else if (updateResult == ILightClientMsgs.UpdateResult.Misbehaviour) {
    clientState.isFrozen = true;
    revert CannotHandleMisbehavior();
} // else: NoOp
```

When `updateResult` is `Misbehaviour`, the client state is set to frozen. However, immediately after, the function reverts.

This suggests two possible intentions:

- If the function is meant to revert in this case, setting `clientState.isFrozen = true` is redundant since the state change is reverted by the next line of code.
- If the goal is to freeze the client, this approach fails because the function reverts before the state change can persist.

Impact

If the first intention applies, this is merely a redundant check and should be classified as `Low/Info` severity.

However, if the latter is intended, the client fails to be frozen, violating a core invariant of punishing misbehavior. This should be considered `Medium` severity.

Recommendation

Depending on what the team has intended the appropriate recommendation should be made.

Discussion

srdtrk

PR addressing this: https://github.com/cosmos/solidity-ibc-eureka/pull/346

Issue L-5: Inconsistency in trusting period validation between IBC-go and solidity implementations

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/161

Description

In the IBC-go implementation, the following check is performed:

```
if (cs.TrustingPeriod >= cs.UnbondingPeriod) {
    return errorsmod.Wrapf(
        ErrInvalidTrustingPeriod,
        "trusting period (%s) should be < unbonding period (%s)",
        cs.TrustingPeriod, cs.UnbondingPeriod
    );
}
```

If `TrustingPeriod` is greater than or equal to `UnbondingPeriod`, the function reverts.

However, in the `SP1ICS07Tendermint.sol` codebase, we find this check:

```
require(
    clientState.trustingPeriod <= clientState.unbondingPeriod,
    TrustingPeriodTooLong(clientState.trustingPeriod, clientState.unbondingPeriod)
);
```

Here, the `trustingPeriod` is required to be less than or equal to the `unbondingPeriod`.

This creates a minor inconsistency:

- In IBC-go, the function reverts when `TrustingPeriod` is equal to or greater than `UnbondingPeriod`.
- In Solidity, the function reverts when `TrustingPeriod` is greater than `UnbondingPeriod`.

As a result, when `trustingPeriod == unbondingPeriod`, the IBC-go implementation fails, while the Solidity implementation passes.

Recommendation

We would recommend ensuring consistency between implementations

Discussion

srdtrk

PR addressing this: https://github.com/cosmos/solidity-ibc-eureka/pull/345

Issue L-6: Hardcoded to 0 `clock_drift` in SP1 programs may lead to interruptions in IBC packet flows

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/162

Summary

In the SP1 program `update-client`, the `clock_drift` submitted to IBC-rs verification methods is hard-coded to be zero. It is crucial that clock drift is configured properly (and greater than 0); otherwise it will be in many cases impossible to construct a proof of the state of a Cosmos chains for the Ethereum light client, because SP1 proof construction may fail due to drifting of clocks between the party constructing the proof, and the corresponding Cosmos chain.

The disjoint interaction between SP1 proof construction using the local time of the party constructing it, and the later validation of that proof by the `SP1ICS07Tendermint` light client on Ethereum, with the `ALLOWED_SP1_CLOCK_DRIFT` of 30 minutes (which is a large value), poses additional security challenges which are hard to foresee.

Vulnerability Detail

In sp1-programs/update-client/src/lib.rs `clock_drift` is hard-coded to be `Duration::default()` (which is 0), and then thus constructed options are passed to IBC-rs `verify_header` method:

```
pub fn update_client(
    client_state: ClientState,
    trusted_consensus_state: ConsensusState,
    proposed_header: Header,
    time: u64,
) -> UpdateClientOutput {
    let client_id = ClientId::new(TENDERMINT_CLIENT_TYPE, 0).unwrap();
    let chain_id = ChainId::from_str(&client_state.chainId).unwrap();
    let options = Options {
        trust_threshold: client_state.trustLevel.clone().into(),
        trusting_period: Duration::from_secs(client_state.trustingPeriod.into()),
        clock_drift: Duration::default(),
    };

    let ctx = types::validation::ClientValidationCtx::new(time,
     ↪  &trusted_consensus_state);

    verify_header::<_, sha2::Sha256>(
        &ctx,
        &proposed_header,
        &client_id,
        &chain_id,
        &options,
        &ProdVerifier::default(),
    )
    .unwrap();
```

This is problematic, due to the following check, which these calls reach via first `IBC-rs/ibc-clients/ics07-tendermint`, and then in tendermint-rs/light-client-verifier/src/predicates.rs:

```
fn is_header_from_past(
    &self,
    untrusted_header_time: Time,
    clock_drift: Duration,
    now: Time,
) -> Result<(), VerificationError> {
    let drifted = (now + clock_drift).map_err(VerificationError::tendermint)?;

    if untrusted_header_time < drifted {
        Ok(())
    } else {
        Err(VerificationError::header_from_the_future(
            untrusted_header_time,
            now,
            clock_drift,
        ))
    }
}
```

In order to understand the security implications of this, it is important to understand the whole execution context.

1. When a new header from a Cosmos chain is retrieved, the `update-client` SP1 program is run (presumably by a relayer). The value of `time` passed to the SP1 program is presumably the local relayer time.
2. The obtained proof is submitted to Ethereum, where the value of `time` from the proof is validated as follows, with ALLOWED_SP1_CLOCK_DRIFT defined to be 30 minutes:

```
require(time <= block.timestamp, ProofIsInTheFuture(block.timestamp, time));
require(block.timestamp - time <= ALLOWED_SP1_CLOCK_DRIFT,
 ↪  ProofIsTooOld(block.timestamp, time));
```

What consequences does this have? First, in step 1, the proof construction may fail with the `header_from_the_future` error if the local time of the relayer and the Cosmos chain time drift from each other. The first time this is discovered, when the clock drifts say by 5 seconds, the relayer may submit to the SP1 program the local time + 5 seconds. When it fails the second and the third time, say with 10 and 15 seconds, a relayer will be tempted to seek a general solution to the problem, and submit e.g. the timestamp of the incoming header as the local `time`, thus effectively fooling the `is_header_from_past` check. Without a known bound for clock drift, provided by the light client, the `ALLOWED_SP1_CLOCK_DRIFT`, i.e. 30 minutes, defines the maximum clock drift that the proof may have. How the clock drift should be configured is best explained in the Hermes relayer documentation on Handling Clock Drift:

> IBC light client security model requires that the timestamp of a header included in client updates for some client is within `[now - client.trusting_period, now + client.max_clock_drift)`. ... IBC light client security model requires that the clocks of the reference and host chains are roughly synchronized. Hermes uses the `clock_drift` and `max_block_time` configuration parameters to determine how much clock drift is tolerable between the reference and host chains.
>
> - `clock_drift` is a correction parameter that specifies how far in the future or past a chain's clock may be from the current time.
> - `max_block_time` is the maximum amount of time that can occur before a new block is created on the chain.
>
> The `clock_drift` parameter values on both the reference and host chains, and `max_block_time` of the host chain are summed to get the `max_clock_drift` when creating a client on the host chain. This can be summarized more succinctly in the following equation:
>
> `client.max_clock_drift = reference.clock_drift + host.max_block_time + host.clock_drift`
>
> Thus, when configuring these values in Hermes' config.toml, keep in mind that this is how these parameters will be used. If the total clock drift is too small, then we run the risk of client updates being rejected because a new block won't have been created yet. It's better to err on the side of total clock drift being larger than smaller, however, if this value ends up being too large, then this becomes a security vulnerability.

The last paragraph is of particular interest. Typical `clock_drift` values are of the order of several seconds; see e.g. Clock Sync Issues #2536, comment, where it's recommended to increase the clock drift to 15 seconds. 30 minutes for this value seems to be excessively large.

It is also worth noting that in the `IBC-go` light client implementation, `clock_drift` is part of the client state, is configurable, and is verified to be strictly greater than 0 upon initialization:

```
if cs.MaxClockDrift <= 0 {
    return errorsmod.Wrap(ErrInvalidMaxClockDrift, "max clock drift must be greater
     ↪  than zero")
}
```

Impact

The first impact is that proof construction by a verifier may fail, due to drifting of clocks between a Cosmos chain, and the local clock of the party constructing the proof. This may negatively impact (interrupt) IBC flows.

While we can't exactly point to what vulnerabilities may arise from a large clock drift value (`ALLOWED_SP1_CLOCK_DRIFT`) of 30 minutes, of interest are e.g. the Security Advisory Alderfly on ”forward lunatic attack”, or the general attacks described in Fork accountability. It's worth keeping in mind that a large clock drift also effectively reduces the trusting period by that value (or even double that value; see the Light Client TLA+ spec).

Of interest is also the interplay between disjoint checking of the timing parameters when constructing the proof, and the delayed validation of those when accepting the proof in the light client. Such interaction scenario has not been explored so far in the research on the light client security, and the effects of it are unknown.

Tool Used

Manual Review

Recommendation

To address the first issue (possibility of proof construction failures), we would recommend the following:

- Make `clock_drift` a configurable parameter of the Tendermint light client in Solidity, also accessible for external parties. This value can then be used by the party constructing the proof.
- Make `clock_drift` an input to SP1 programs, as well as part of the proof public values.
- When receiving a relayer transaction, validate the value of `clock_drift` from the proof public values against that specified upon light client creation.

To address the possible issues arising from the large clock drift value (`ALLOWED_SP1_CLOCK_DRIFT`) of 30 minutes, we would recommend the following:

- Consider the possibility of reducing this value. 30 minutes seems excessively large; we have not been able to find a justification for this value. Is it possible to reduce it? Would e.g. 5 minutes suffice?
- Consult with parties performing light client research on the consequences of the disjoint construction and validation of the proof, and what effects large values of clock drift may have upon this interaction.

Issue L-7: Processing the same amount of packets in the same frequency in solidity-ibc-eureka and ibc-go would have their sequence used up way earlier for channels in the solidity side

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/165

Summary

There is a mismatch in the amount of packets that should be processed, i.e have no explicit handling of sequence overflow on the solidity side since the maximum value of uint32 is 2^32 - 1 = 4,294,967,295. Unlike the Go implementation which uses uint64.

Vulnerability Detail

Each client has its own sequence counter:

```
// In IBCStoreStorage
mapping(string clientId => uint32 sequence) prevSequenceSends;
```

When sending a packet, the sequence is automatically incremented:

```
function nextSequenceSend(string calldata clientId) internal returns (uint32) {
    IBCStoreStorage storage $ = _getIBCStoreStorage();
    uint32 seq = $.prevSequenceSends[clientId] + 1;
    $.prevSequenceSends[clientId] = seq;
    return seq;
}

// The ICS26Router uses this sequence when creating a new packet:
uint32 sequence = nextSequenceSend(msg_.sourceClient);

IICS26RouterMsgs.Packet memory packet = IICS26RouterMsgs.Packet({
    sequence: sequence,
    sourceClient: msg_.sourceClient,
    destClient: counterpartyId,
    // ...
});
```

Now in the Go implementation, sequences are correctly handled using uint64:

```
// GetNextSequenceSend returns the next send sequence from the sequence path
func (k *Keeper) GetNextSequenceSend(ctx context.Context, clientID string) (uint64,
↪   bool) {
    store := k.storeService.OpenKVStore(ctx)
    bz, err := store.Get(hostv2.NextSequenceSendKey(clientID))
    if err != nil {
        panic(err)
    }
    if len(bz) == 0 {
        return 0, false
    }
    // ... decode and return sequence as uint64
    return sdk.BigEndianToUint64(bz), true
}
```

This is also sufficiently back tracked, cause when receiving packets on the solidity side we are expected to work with sequences of `uint64` type

Since the Path generation and verification use uint64 to maintain protocol compatibility with the ics-024host: https://github.com/cosmos/ibc/tree/main/spec/core/ics-024-host-requirements#path-space

In code implementation:

Among other paths, we always use uint64, for e.g when getting path for acknowledging or receipt which is used from `ackPacket()` `timeoutPacket()`:

```
    function packetAcknowledgementCommitmentPathCalldata(
        string memory clientId,
|>        uint64 sequence
    )
        internal
        pure
        returns (bytes memory)
    {
        return abi.encodePacked(clientId, uint8(3), uint64ToBigEndian(sequence));
    }

    function packetReceiptCommitmentPathCalldata(
        string memory clientId,
|>        uint64 sequence
    )
        internal
        pure
        returns (bytes memory)
    {
        return abi.encodePacked(clientId, uint8(2), uint64ToBigEndian(sequence));
    }
```

Despite the above, the solidity still enforces a sequence of uint32 in it's packets

Impact

The Solidity implementation will hit sequence overflow at 2^32 - 1 = 4,294,967,295 packets whereas the Go implementation can handle up to 2^64 - 1.

Meaning we have the Solidity implementation to be limited to 4.29 billion packets, whereas the implementation should be able receive and verify packets with sequences up to uint64 from other chains.

This is info level cause likelihood is low as this requires a very high number of packets, but the impact would be severe as it could break IBC communication, using a theoretical example here is if we have:

Low likelihood here is justified cause even if we process 10,000 packets/day we still need a ~ dozen years to hit the max.

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/ibc-go/modules/core/04-channel/v2/keeper/keeper.go#L152-L162 https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/utils/ICS24Host.sol#L50-L75

Tool Used

Manual Review

Recommendation

1. Change the sequence type in Solidity to match Go's implementation:

```
// In IBCStoreStorage
mapping(string clientId => uint64 sequence) prevSequenceSends;
```

2. Update all related functions to use uint64:

```
function nextSequenceSend(string calldata clientId) internal returns (uint64) {
    IBCStoreStorage storage $ = _getIBCStoreStorage();
    uint64 seq = $.prevSequenceSends[clientId] + 1;
    $.prevSequenceSends[clientId] = seq;
    return seq;
}
```

This would ensure consistency with the Go implementation and prevent any potential sequence overflow issues.

Issue L-8: Stale documentation claims valid IBC packets are no longer allowed to be processed

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/166

Summary

The IBC implementation in both Go and Solidity enforces strict timeout validation that requires non-zero timeout timestamps, contradicting the documented behavior that suggests timeout can be disabled by setting it to 0. This inconsistency between implementation and documentation could lead to confusion and failed transactions when users attempt to follow the documented behavior.

Vulnerability Detail

The IBC documentation in the Go implementation states: https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/ibc-go/modules/apps/transfer/types/msgs.go#L65

> // NOTE: timeout height or timestamp values can be 0 to disable the timeout.

However, both implementations enforce strict timeout validation:

- Solidity Implementation (when sending packet in `ICS26Router.sol`):

```
require(
    msg_.timeoutTimestamp > block.timestamp,
    IBCInvalidTimeoutTimestamp(msg_.timeoutTimestamp, block.timestamp)
);
```

2. Go Implementation also checks against this:

```
// timeoutTimestamp must be greater than current block time
timeout := time.Unix(int64(msg.TimeoutTimestamp), 0)
if timeout.Before(sdkCtx.BlockTime()) {
    return nil, errorsmod.Wrap(types.ErrTimeoutElapsed, "timeout is less than the
     ↪  current block timestamp")
}

// timeoutTimestamp must be less than current block time + MaxTimeoutDelta
if timeout.After(sdkCtx.BlockTime().Add(types.MaxTimeoutDelta)) {
    return nil, errorsmod.Wrap(types.ErrInvalidTimeout, "timeout exceeds the maximum
     ↪  expected value")
}
```

This creates a situation where:

- Users following the documentation might attempt to disable timeouts by setting timestamp to 0
- Such transactions will be rejected by both implementations
- There is no actual way to disable packet timeouts as documented

Impact

Disabling the timestamp validation is no longer possible, which would then break potential integration issues for applications built assuming the documented behavior.

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/ibc-go/modules/core/04-channel/v2/keeper/msg_server.go#L24-L32

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/ICS26Router.sol#L120-L124

Tool Used

Manual Review

Recommendation

If indeed timestamp validation is no longer possible to be disabled, then update the documentation to reflect the actual implementation:

```
// NOTE: timeout timestamp must be greater than the current block timestamp
// and cannot be disabled by setting to 0.
```

Alternatively, modify the implementations to allow timeout disabling as documented

```
// Allow 0 as special case to disable timeout
require(
    msg_.timeoutTimestamp == 0 || msg_.timeoutTimestamp > block.timestamp,
    IBCInvalidTimeoutTimestamp(msg_.timeoutTimestamp, block.timestamp)
);
```

Which would also need to be translated to when receiving the packets.

Issue L-9: EVM IBC implementation should scrutinise the memo fields against callback instructions

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/173

Summary

The Solidity IBC implementation accepts and processes transfer packets with non-empty memo fields, silently ignoring any callback instructions that might be present. Since callback functionality on the EVM side is not implemented, this creates an asymmetry between Cosmos and EVM chains that could lead to unexpected behavior for users.

Vulnerability Detail

Callbacks are intended to be based on an opting in logic, this can be seen even hinted in the Actor flow shared for this audit.

Where there has been an explicit mention that **if and only if** the user explicitly specifies the contract address in the callback memo of the transfer packet, then the callback logic is invoked.

Now on the ibc-go side, this is fully ensured, i.e if the user does indeed opt in for the callbacks then the callback logic is indeed invoked:

```go
func (im IBCMiddleware) OnRecvPacket(..snip)

{
    // ..snip

    sdkCtx := sdk.UnwrapSDKContext(ctx)
    cbData, err := types.GetCallbackData(
        packetData, payload.GetVersion(), payload.GetDestinationPort(),
        sdkCtx.GasMeter().GasRemaining(), im.maxCallbackGas,
         ↪  types.DestinationCallbackKey,
    )
    // OnRecvPacket is not blocked if the packet does not opt-in to callbacks
    if err != nil {
|>                  return recvResult
    }

    callbackExecutor := func(cachedCtx sdk.Context) error {
        // reconstruct a channel v1 packet from the v2 packet
        // in order to preserve the same interface for the contract keeper
        packetv1 := channeltypes.Packet{
            Sequence:           sequence,
            SourcePort:         payload.SourcePort,
            SourceChannel:      sourceClient,
            DestinationPort:    payload.DestinationPort,
            DestinationChannel: destinationClient,
            Data:               payload.Value,
            TimeoutHeight:      clienttypes.Height{},
            TimeoutTimestamp:   0,
        }
        // wrap the individual acknowledgement into the
        ↪  channeltypesv2.Acknowledgement since it implements the
        ↪  exported.Acknowledgement interface
        // since we return early on failure, we are guaranteed that the ack is a
        ↪  successful acknowledgement
        ack := channeltypesv2.NewAcknowledgement(recvResult.Acknowledgement)
|>      return im.contractKeeper.IBCReceivePacketCallback(cachedCtx, packetv1, ack,
↪   cbData.CallbackAddress, payload.Version)
    }

        // callback execution errors are not allowed to block the packet lifecycle,
        ↪  they are only used in event emissions
|>  err = internal.ProcessCallback(sdkCtx, types.CallbackTypeReceivePacket, cbData,
↪   callbackExecutor)
    types.EmitCallbackEvent(
        sdkCtx, payload.DestinationPort, destinationClient, sequence,
        types.CallbackTypeReceivePacket, cbData, err,
    )

    return recvResult
        }
```

From the same document we have the provided packets flow documentation where we know that the callback logic is out of scope for the EVM implementation since it's not implemented:

Now despite callbacks being out of scope, the current implementation silently accepts and tries to process packets with non-empty memos:

- In ICS26Router.sol, `recvPacket` passes the core packet information to the `onRecvPacket` after some checks.
- The ICS20Transfer.sol `onRecvPacket` function decodes the packet data including the memo field:

```
IICS20TransferMsgs.FungibleTokenPacketData memory packetData =
    abi.decode(msg_.payload.value, (IICS20TransferMsgs.FungibleTokenPacketData));
```

- The FungibleTokenPacketData struct defines the memo field as optional:

```

|>    /// @param memo Optional memo
    struct FungibleTokenPacketData {
        string denom;
        string sender;
        string receiver;
        uint256 amount;
|>          string memo;
    }
```

- However, the implementation never checks if the memo field is empty, and simply ignores any callback instructions that might be present:

```
// No check for memo.length > 0 to revert the packet receival since callback
↪  instructions are not supported

// Just transfers tokens to receiver
escrow.send(IERC20(erc20Address), receiver, packetData.amount);
return ICS20Lib.SUCCESSFUL_ACKNOWLEDGEMENT_JSON;
```

Unlike the Cosmos implementation, the EVM side has no logic to parse or handle callback instructions in the memo, which means:

1. The memo content is completely ignored
2. No error is returned when callback instructions are present
3. The user receives no indication that their callback intent was not fulfilled

Impact

The asymmetry in how memos with callback instructions are handled creates a potential for confusion and unexpected behavior:

1. Users familiar with Cosmos-to-Cosmos transfers might expect similar behavior when transferring to EVM chains
2. There's no feedback mechanism to inform users that callback functionality is not supported
3. In scenarios where users are interacting with protocols that rely on callbacks, transactions will complete but with only partial functionality

While memos can be used for purposes other than callbacks (which are processed correctly), the specific case of callback instructions being silently ignored could lead to incomplete cross-chain workflows, so users sending packets with callbacks from Cosmos chains to EVM chains will have their transfers completed, but any callback logic will be silently ignored, breaking cross-chain workflows and putting the full packet in a limbo (not fully processed) state_since they wouldn't want the transfer to be processed if the callback wouldn't be attempted.

To attempt to make a case on a more concised impact here would be to consider a case of a classic protocol on the EVM side, say a vault, where in order for users to receive the shares from that vault they need to deposit while sending in the tokens, now in the memo a user could specify that this logic should be called, however cause we don't do anything with it, the tokens are stuck and lost cause in this case the user would not specify their own controlled address as the recipient of the packet and since it gets fully processed to this vault/or the router before calling the vault then the tokens are stuck here and the user can not get them back. Other cases could be if a user attempts to forward while having ETH as the intermediary chain, the memo can't be processed.

Ultimately this can be heavily looked at as a user error, which is why it's low/info.

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka//blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/msgs/IICS20TransferMsgs.sol#L30-L37

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka//blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/ICS20Transfer.sol#L229-L292

Tool Used

Manual Review

Recommendation

Consider parsing the memo field to detect callback instructions (looking for specific callback format data) similar format to what's used on Cosmos chains and reject the packet/let it timeout.

Alternatively, **document clearly** in user-facing materials that callback functionality is not supported on EVM chains and callback instructions in memos will be ignored.

Issue L-10: Blockchain height is `uint32` in IBC-Solidity, which deviates from `uint64` both in IBC-Go and IBC-rs

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/176

Summary

The blockchain height struct is defined in IICS02ClientMsgs.sol as follows:

```
struct Height {
    uint32 revisionNumber;
    uint32 revisionHeight;
}
```

This deviates from the `uint64` employed both in IBC-Go and in IBC-rs.

IBC-Go/modules/core/02-client/types/client.pb.go:

```
type Height struct {
    // the revision that the client is currently on
    RevisionNumber uint64
     ↪  `protobuf:"varint,1,opt,name=revision_number,json=revisionNumber,proto3"
     ↪  json:"revision_number"`
    // the height within the given revision
    RevisionHeight uint64
     ↪  `protobuf:"varint,2,opt,name=revision_height,json=revisionHeight,proto3"
     ↪  json:"revision_height"`
}
```

IBC-rs/ibc-core/ics02-client/types/src/height.rs:

```
pub struct Height {
    /// Previously known as "epoch"
    #[cfg_attr(feature = "serde", serde(default))]
    revision_number: u64,


    /// The height of a block
    revision_height: u64,
}
```

Impact

There are already Cosmos chains with block times below 1 second; e.g. Sei, with block time of 0.4 seconds. With such block times `uint32` would be exhausted in ~54 years. With even smaller block times, which seems feasible, correspondingly faster; e.g. with 0.1 seconds it would be ~14 years.

Besides potential overflows (which is still relatively far in the future), this discrepancy may open a way to other vulnerabilities. Though the type conversions between the Solidity light client, SP1 programs, and IBC-rs verification seems to be handled correctly, there are always possibilities for subtle conversion bugs.

Another issue is that the `Height` struct above is part of the public interface of Solidity light clients and the `ICS26Router` contract. Unlike internal state variables, updating interfaces in case the need arises would require re-instantiating the whole contract setup, thus breaking external dependencies.

Recommendation

We recommend to employ `uint64` instead of `uint32` for the `Height` struct.

Issue L-11: It is impossible to securely connect 2 Ethereum chains with the current IBC-Solidity API

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/177

Summary

In the IBC specification, channel setup is a two-step procedure, for each of the connecting chains:

1. `CreateClient` creates a new client, allocating a new client id;
2. `RegisterCounterparty` submits a client id of the counterparty client, thus allowing to establish a correspondence between two client ids on two chains; this pair then serves as a channel identifier.

This is exactly how it is done in `IBC-Go`; see ibc-go/modules/core/keeper/msg_server.go::CreateClient, and ibc-go/modules/core/keeper/msg_server.go::RegisterCounterparty

Contrary to that, `IBC-Solidity` combines both steps into one, ICS02ClientUpgradeable::addClient. In this setup, it is possible to securely connect a Cosmos chain to an Ethereum chain via the sequence:

1. `CreateClient` on Cosmos
2. `addClient` on Ethereum
3. `RegisterCounterparty` on Cosmos.

So this works for the initial version, which is supposed to connect only Cosmos chains to Ethereum. Unfortunately, if in the future there will be a need to connect two Ethereum chains (e.g. Ethereum mainnet and Optimism), this will be impossible to achieve, as `addClient` needs a known counterparty client id to setup the connection securely.

Impact

It is impossible to securely connect two Ethereum chains using the current IBC-Solidity API. In case such need arises, a refactoring will be needed, splitting the call into two, as in IBC-Go: `CreateClient`, and `RegisterCounterparty`.

Recommendation

While this doesn't influence in any way IBC V2 launch, if connections between Ethereum chains are planned in the future, it might be beneficial to perform the refactoring earlier, rather than later, in order not to introduce breaking changes down the road.

Issue L-12: Large KV pair size can result in out of gas

Source:
https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/issues/178

Summary

In `SP1ICS07Tendermint`'s `verifyMembership`:

```
require(
    output.kvPairs.length > 0 && output.kvPairs.length <= 256,
     ↪  LengthIsOutOfRange(output.kvPairs.length, 1, 256)
);
```

contract allows upto 256 KV pairs in the payload.

However, from the benchmarks test it is evident that average case of 50 KV pairs can consume upto `5,000,000` gas units on ethereum and this number could be higher as some proofs can be more complex than average case.

So it is very likely that 256 KV pairs will suprass block gas limit and will not be processed.

Impact

It breaks the core invaraint offered by the contract. Since relayer can always subimit transfers in smaller quantities, there is no direct loss of funds or damange.

Code Snippet

https://github.com/sherlock-audit/2025-02-interchain-labs-ibc-eureka/blob/c44d895d2609554954b803207619f80f1c9d8be8/solidity-ibc-eureka/contracts/light-clients/SP1ICS07Tendermint.sol#L214-L216

Tool Used

Manual Review

Recommendation

Conduct a proper benchmark for 256 KV pairs and consider lowering to 128 pairs or some other value after inspection

Disclaimers