Zellic

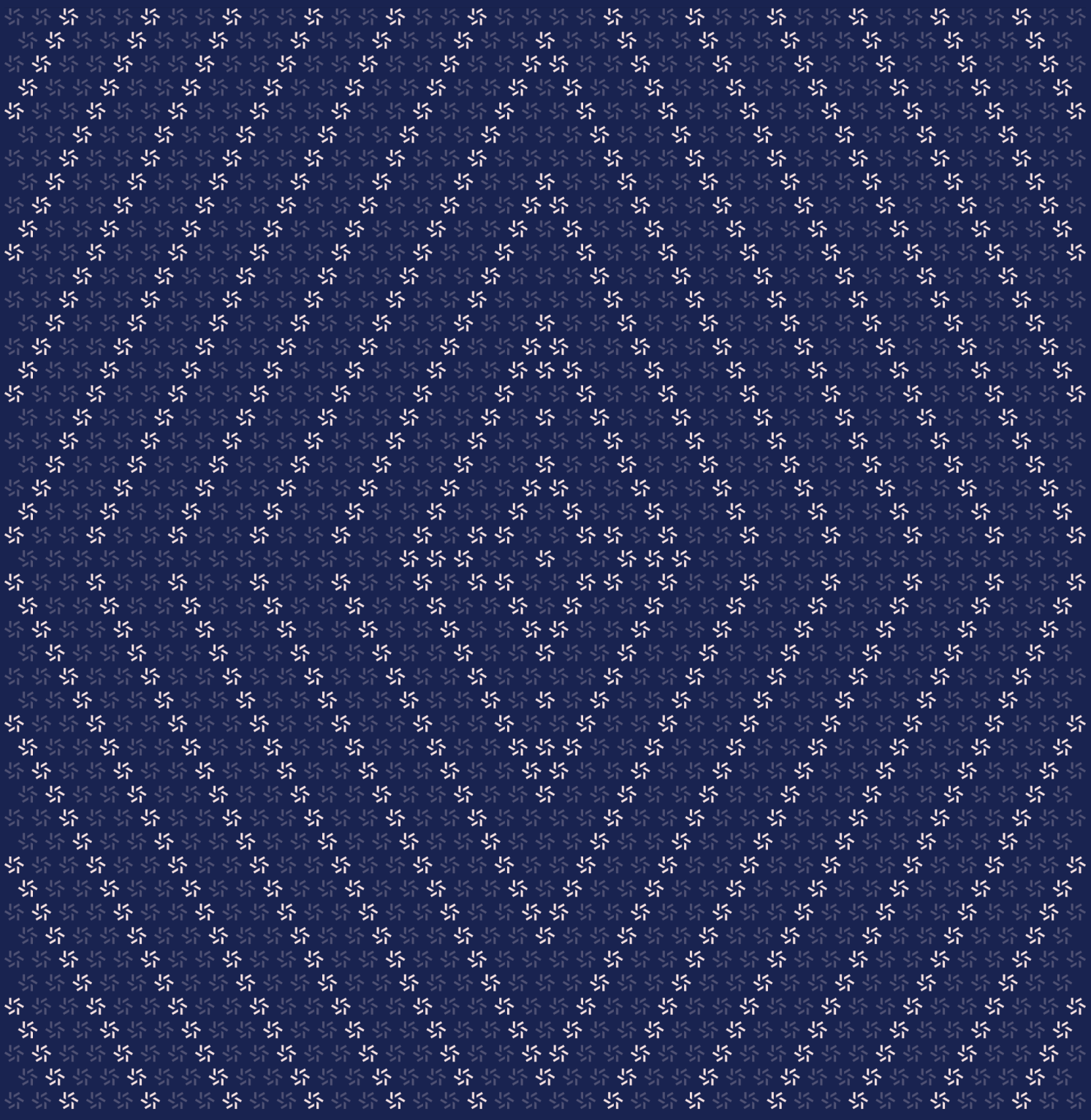
Prepared for
IBC Team
Interchain Labs

Prepared by
Nipun Gupta
Ayaz Mammadov
Zellic

March 25, 2025

IBC Eureka

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Interchain Labs from March 3rd to March 19th, 2025. During this engagement, Zellic reviewed IBC Eureka's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a user send a transfer packet without escrowing/burning the tokens on the sender side?
- Could a user successfully authenticate sending a transfer packet on behalf of another user?
- Is it ensured that a malicious validator set on chain X cannot steal funds on a counterparty chain that did not originate from chain X (i.e., can only steal funds that they had sent out)?
- Could an ERC-20 trick the escrow into thinking it is an IBCERC20.sol minted by the protocol?
- Are there any possible replay attacks on messages?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped IBC Eureka contracts, we discovered five findings. Two critical issues were found. One was of high impact, one was of medium impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Interchain Labs in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	1
Medium	1
Low	0
Informational	1

2. Introduction

2.1. About IBC Eureka

Interchain Labs contributed the following description of IBC Eureka:

> The Inter-Blockchain Communication (IBC) protocol is a blockchain interoperability solution that enables secure, permissionless, and feature-rich cross-chain interactions for seamless data and value transfer without a third-party intermediary. IBC v2 is the evolution of IBC, significantly simplifying the protocol while maintaining expressiveness. V2 makes it easier to connect with diverse blockchain ecosystems, starting with Ethereum and Solana to follow.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no

hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

IBC Eureka Contracts

Types	Solidity, Rust
Platforms	EVM-compatible, Cosmos
Target	solidity-ibc-eureka
Repository	https://github.com/cosmos/solidity-ibc-eureka ↗
Version	`6bb8fcf6af5094487c85f12d9398c8401fd4a1b7`

Programs	`contracts/**`
	`packages/ethereum/ethereum-light-client/**`
	`packages/ethereum/ethereum-trie-db/**`
	`packages/ethereum/ethereum-types/**`
	`packages/ethereum/tree_hash/**`
	`packages/sp1-ics07-tendermint-prover/**`
	`packages/sp1-ics07-tendermint-utils/**`
	`programs/cw-ics08-wasm-eth/**`
	`programs/sp1-programs/**`

Target	skip-go-evm-contracts
Repository	https://github.com/skip-mev/skip-go-evm-contracts ↗
Version	`64c5ba2db3155fee10d5fb5331dd371499c9548f`
Programs	`EurekaHandler`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-weeks. The assessment was conducted by two consultants over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Nipun Gupta
Engineer
nipun@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 3, 2025	Start of primary review period
March 4, 2025	Kick-off call
March 19, 2025	End of primary review period

3. Detailed Findings

3.1. Untrusted input is used as trusted consensus state

Target	ethereum-light-client		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The function `verify_header` is called during the `QueryMsg::VerifyClientMessage`, which is used to verify the client message if the client message is of the type `Header`. The verified header is then used to update the consensus and the client state. The `header` is provided via the user and has to be verified to be correct before using the header to update the states.

The function `verify_header` creates a `TrustedConsensusState` struct via the `consensus_state` and the `header.trusted_sync_committee.sync_committee` where the `consensus_state` is the state stored corresponding to the latest slot and the `header.trusted_sync_committee.sync_committee` is what the user provided, as shown below:

```
pub fn verify_header<V: BlsVerify>(
    consensus_state: &ConsensusState,
    client_state: &ClientState,
    current_timestamp: u64,
    header: &Header,
    bls_verifier: V,
) -> Result<(), EthereumIBCError> {
    let trusted_consensus_state = TrustedConsensusState {
        state: consensus_state.clone(),
        sync_committee: header.trusted_sync_committee.sync_committee.clone(),
    };
```

Here, the `sync_committee` is not validated against the aggregated key stored in the consensus state and hence the trusted sync committee remains untrusted.

There is a similar issue in `update_consensus_state` where the `trusted_slot` comes from the header, which is not verified against the stored consensus state.

Impact

As the sync committee is used to validate the `attested_header`, an incorrect header could be provided, which could be used to prove an incorrect `finalized_root` and state root. This could be used to update the light client with incorrect states.

Recommendations

We recommend verifying the `sync_committee` provided by the header against the `current_sync_committee` or `next_sync_committee` stored in the consensus state of the storage.

Remediation

This issue has been acknowledged by Interchain Labs, and a fix was implemented in commit `1f242e04` ↗.

3.2. Merkle verification could be bypassed

Target	ethereum-light-client		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

As per the sync-protocol specifications, the `update.next_sync_committee` should be validated via the Merkle proof if the `next_sync_committee_branch` is not empty, as shown below:

```
if not is_sync_committee_update(update):
    assert update.next_sync_committee == SyncCommittee()
else:
    if update_attested_period == store_period
    and is_next_sync_committee_known(store):
        assert update.next_sync_committee == store.next_sync_committee
    assert is_valid_normalized_merkle_branch(
        leaf=hash_tree_root(update.next_sync_committee),
        branch=update.next_sync_committee_branch,

    gindex=next_sync_committee_gindex_at_slot(update.attested_header.beacon.slot)
        root=update.attested_header.beacon.state_root,
    )
```

But in the implementation of ethereum-light-client, the Merkle verification is executed in the case `update.next_sync_committee` and `trusted_consensus_state.next_sync_committee()` are both available. As the `trusted_consensus_state.next_sync_committee()` (see Finding 3.1. ↗) comes from user input too, it might be possible to prove the incorrect `next_sync_committee`.

```
if let (Some(next_sync_committee), Some(stored_next_sync_committee)) = (
    &update.next_sync_committee,
    trusted_consensus_state.next_sync_committee(),
) {
    if update_attested_period == stored_period {
        ensure!(
            next_sync_committee == stored_next_sync_committee,
            EthereumIBCError::NextSyncCommitteeMismatch {
                expected: stored_next_sync_committee.aggregate_pubkey,
                found: next_sync_committee.aggregate_pubkey,
            }
```

```
        );
    }
    // This validates the given next sync committee against the attested
    header's state root.
    validate_merkle_branch(
        next_sync_committee.tree_hash_root(),
        update.next_sync_committee_branch.unwrap_or_default().into(),
        NEXT_SYNC_COMMITTEE_BRANCH_DEPTH,
        get_subtree_index(NEXT_SYNC_COMMITTEE_INDEX),
        update.attested_header.beacon.state_root,
    )
    .map_err(|e|
    EthereumIBCError::ValidateNextSyncCommitteeFailed(Box::new(e)))?;
}
```

Impact

As the sync committee is used to validate the `attested_header`, an incorrect header could be provided, which could be used to prove an incorrect `finalized_root` and state root. This could be used to update the light client with incorrect states.

Recommendations

We recommend following the specification and validating the Merkle proof in cases similar to how it is validated in the specification.

Remediation

This issue has been acknowledged by Interchain Labs, and a fix was implemented in commit `d46a8e55` ↗.

3.3. IBC does not work with chains that generate subsecond blocks

Target	SP1ICS07Tendermint		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	High

Description

The `_checkUpdateResult` function ensures that the timestamp of the new valid submitted consensus state (`newConsensusState.timestamp`) is greater than the last trusted consensus state, `trustedConsensusState.timestamp`.

```
function _checkUpdateResult(IUpdateClientMsgs.UpdateClientOutput memory
    output)
    private
    view
    returns (ILightClientMsgs.UpdateResult)
{
    bytes32 consensusStateHash
    = _consensusStateHashes[output.newHeight.revisionHeight];
    if (consensusStateHash == bytes32(0)) {
        // No consensus state at the new height, so no misbehaviour
        return ILightClientMsgs.UpdateResult.Update;
    } else if (
        consensusStateHash != keccak256(abi.encode(output.newConsensusState))
            || output.trustedConsensusState.timestamp
    >= output.newConsensusState.timestamp
    ) {
        // The consensus state at the new height is different than the one in
    the mapping
        // or the timestamp is not increasing
        return ILightClientMsgs.UpdateResult.Misbehaviour;
```

However, this invariant does not hold for chains that can produce more than a block a second.

Impact

IBC clients will be frozen. IBC will not work on chains that generate more than one block a second (subsecond blockchains).

Recommendations

Adjust the timestamp to use finer-grained measures, such as milliseconds or even nanoseconds.

Remediation

This issue has been acknowledged by Interchain Labs, and a fix was implemented in commit b05dea16 ↗.

3.4. Attested header is stored instead of finalized header

Target	ethereum-light-client		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The function `update_consensus_state` is used to return the updated consensus state, updated slot, and the updated client state, which is then stored in the storage. The function uses the slot/timestamp and root values of the attested header instead of the finalized header to update the storage. As per the specification of the sync protocol, the finalized header should be used to update the storage. Furthermore, the slot of the attested header is used to calculate the `update_finalized_period`, as shown below:

```
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 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))
```

Similarly in `verify.rs:verify_header`, the `verify_account_storage_root` function should use the finalized header's `state_root` instead of the attested header's state root.

Impact

The security guarantees of the attested header and the finalized header are different. The attested header represents a block that is highly likely to be a part of the canonical chain but has not reached finality. In the case of a chain reorg, the attested header cannot be trusted.

Recommendations

We recommend using the finalized header for the updates instead of attested header, as per the specification.

Remediation

This issue has been acknowledged by Interchain Labs, and a fix was implemented in commit `537b47c1` ↗.

3.5. Unchecked slippage may lead to sandwich attacks

Target	EurekaHandler		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `swapAndTransfer` function in `EurekaHandler` allows users to swap a token to another token before transferring the output token via ics20. The swap is performed on the `swapRouter` using a low level call with the calldata `swapCalldata`. It is unclear how `swapCalldata` is created as it directly comes from the input and hence might contain the minimum amount of output tokens to be `0` or something less than expected.

Impact

Swaps could be sandwiched causing a loss of funds for users.

Recommendations

Slippage parameters should be verified in the calldata, or an additional min output amount could be added in `swapAndTransfer` which could be used to verify the amount after the swap.

Remediation

This issue has been acknowledged by Interchain Labs.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Centralization risks and trust assumptions

When a router calls an app with the message, the apps should not blindly trust the message because any user could verify the membership by creating a fake light client.

Similarly, there is a centralization risk involved as the light client could be migrated to a new address. This could lead to a malicious/hacked migrator migrating the light-client address to a fake address such that it could prove fake membership. Or a malicious/hacked migrator could intentionally revert the calls to the light client, which might lead to the tokens of users being stuck.

As per the Interchain Labs team, this is the accepted behavior and the apps should perform their due diligence before trusting the migrator.

4.2. One app could be registered with multiple ports

Two `portId`s could point to the same app in ICS26Router. This could only happen in case an app is already registered with a custom `portId`, because then anyone could add a new IBC app that points to the same address with `portId` set to an empty string. The function `addIBCApp` will set the `portId` in this app to be the hex value of the address of the app, and therefore the app could be accessed via two different `portId`s. Here is the relevant code:

```
function addIBCApp(string calldata portId, address app) external {
    string memory newPortId;
    if (bytes(portId).length != 0) {
        _checkRole(PORT_CUSTOMIZER_ROLE);
        newPortId = portId;
    } else {
        newPortId = Strings.toHexString(app);
    }
    //...
}
```

While it is not currently a security issue, if any app tried to backlist another app using the `portId`, it might be possible to bypass the blacklist using the method discussed above if the blacklisted app has been registered via a custom `portId`.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: cs-ics08-wasm-eth

Description

This is the CosmWasm implementation that can be used with ibc-go's 08-wasm light-client wrapper. It handles the client and consensus state and calls into `packages/ethereum-light-client` for all the light-client–related logic.

The `packages/ethereum-light-client` performs stateless verification. It contains all the core logic for verifying Ethereum consensus, proving state — verifying (non)membership — and the headers submitted to update the light client. The state is handled by the CosmWasm code.

The contract has three entry points:

1. `instantiate` — This initializes the initial `client_state` and the consensus state.
2. `sudo` — The sudo entry point could only be called via the trusted modules and are used for verification of membership/misbehavior and updating the state.
3. `query` — This is used to query the state.

The `packages/ethereum-light-client` performs the following verifications:

1. `verify_membership`, which is used to verify the membership of a key in the storage.
2. `verify_misbehaviour`, which verifies if a consensus misbehavior is valid by checking if the two conflicting light-client updates are valid.
3. `verify_header`, which verifies the header of the light client.

Invariants

- During the `SudoMsg::UpdateState` message, the `update_consensus_state` function must be invoked with the current client state, current consensus state, and a new header, resulting in the updated consensus state being stored and, if required, the updated client state being stored as well.
- The `SudoMsg::UpdateStateOnMisbehaviour` message should set the `is_frozen` variable to true in the ETH client state.

- The `SudoMsg::VerifyMembership` should verify the membership proof at the given consensus height and the current client state. And the `SudoMsg::VerifyNonMembership` should perform similar actions but for nonmembership proofs.
- The `QueryMsg::VerifyClientMessage` message should correctly verify the header that will be used for updating the state of the light client. The verification of the header is done in `ethereum_light_client::verify::verify_header`, which must verify the `account_proof` and that the update has been signed by at least two thirds of the sync committee. In the case the type of the `client_message` is of `EthereumMisbehaviourMsg`, it should verify the misbehavior via `ethereum_light_client::misbehaviour::verify_misbehaviour`.
- The `QueryMsg::CheckForMisbehaviour` should return true if the provided `client_message` corresponds to two conflicting headers.
- The `validate_light_client_update` function must perform the verification of the update as per the consensus specification ↗.
- The `update_consensus_state` should also update the consensus state as per the consensus specification.

Test coverage

Cases covered

- Membership verification is working as expected.
- Header verification is working as expected.
- Updating consensus states via `update_consensus_state` returns the expected `updated_slot`, `new_consensus_state`, and `new_client_state`.

Cases not covered

- Unit tests for misbehavior verification.

Attack surface

The ibc-go first performs the verification via the query message and then updates the states via the sudo message. It is crucial for the verification logic to be similar to the specification mentioned in the sync protocol docs. Any deviation from the proposed specification might lead to critical issues in the verification process. We verified the implementation against the specification and found several instances where the implementation did not follow the specification:

- Finding 3.2. ↗: Merkle verification could be bypassed.
- Finding 3.4. ↗: Attested header is stored instead of finalized header.

There were a few instances where the provided user input was used as trusted states, which might lead to issues as the user input is untrusted. The issue is discussed in detail in Finding 3.1. ↗.

5.2. Component: sp1-programs

Description

The SP1 programs are used to create proofs for the update, misbehavior, and membership functionality. The proofs are later used by the SP1ICS07Tendermint contract, which verifies them and performs certain actions. There are four different proof generators, and the verification of all of them is handled by the ibc_client_tendermint module.

The operator calls `generate_proof` in `sp1-ics07-tendermint-prover` for each of these four different proofs, which generates the proofs, and the returned output is either written to a file or printed to stdout.

Invariants

- The proof generation must fail if the verification fails. For example, in case of updating the client, the function `update_client` calls the `verify_header` function, which verifies the proposed header.
- Similarly, proof generation must fail if the verification of membership or misbehavior fails.

Test coverage

Cases covered

- The end-to-end tests cover the proof generation for `misbehaviour`, `membership`, `update_client`, and the `update_client and membership` proof types.

Cases not covered

- N/A.

Attack surface

The verification is handled by the ibc_client_tendermint module, hence the attack surface is minimized. We verified if any of the crucial variables remain unverified and did not find any such unverified variables.

5.3. Component: SP1ICS07Tendermint

Function: `updateClient`

The function verifies the public values via the SP1 verifier and updates the client and the consensus state.

Inputs

- `updateMsg`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid update — otherwise, the SP1 verifier will revert.
  - **Impact**: Updates the client and consensus state.

Branches and code coverage

Intended branches

- ☑ The state should be updated as per the result of the `_checkUpdateResult` call.

Negative behavior

- ☑ The vkey should be equal to `UPDATE_CLIENT_PROGRAM_VKEY`.
- ☑ The time should be in the correct range.
- ☑ The `chainId`, `trustLevel`, `trustingPeriod`, and `unbondingPeriod` of the public client state should match that of the stored client state.
- ☑ The trusted consensus-state hash of the output should match the stored consensus-state hash.
- ☑ The SP1 proof should be valid.

Function call analysis

- `VERIFIER.verifyProof`
  - **What is controllable?** `proof.vKey`, `proof.publicValues`, and `proof.proof`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

Function: `verifyMembership`

The function verifies the public values via the SP1 verifier and verifies that the requested path/value is present in the output paths/values.

Inputs

- `msg_`
  - **Control**: Arbitrary.

- **Constraints**: Must be a valid membership proof — otherwise, the SP1 verifier will revert.
- **Impact**: Returns the timestamp of the trusted consensus state if the requested path/value is found in the output.

Branches and code coverage

Intended branches

- ☑ If the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipProof`, it returns the timestamp of the trusted consensus state if the requested path and value are found in the output.
- ☑ If the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipAndUpdateClientProof`, it first updates the consensus and client state, then verifies the membership proof, and finally returns the timestamp of the trusted consensus state that was updated.

Negative behavior

- ☑ Membership proof value and key should be present in the kvPairs of the output.
- ☑ `msg_.value` should not be empty.
- ☑ The vkey should be equal to `MEMBERSHIP_PROGRAM_VKEY` if the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipProof` and `UPDATE_CLIENT_AND_MEMBERSHIP_PROGRAM_VKEY` if the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipAndUpdateClientProof`.
- ☑ The trusted consensus state hash of the output should match the stored consensus state hash.
- ☑ The SP1 proof should be valid.

Function call analysis

- `VERIFIER.verifyProof`
  - **What is controllable?** `proof.vKey`, `proof.publicValues`, and `proof.proof`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

Function: `verifyNonMembership`

The function verifies the public values via the SP1 verifier and verifies that the requested path and empty value are present in the output paths/values.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid nonmembership proof — otherwise, the SP1 verifier will revert.
  - **Impact**: Returns the timestamp of the trusted consensus state if the requested path and empty values are found in the output.

Branches and code coverage

Intended branches

- ☑ If the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipProof`, it returns the timestamp of the trusted consensus state if the requested path and empty value are found in the output.
- ☑ If the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipAndUpdateClientProof`, it first updates the consensus and client state, then verifies the nonmembership proof, and finally returns the timestamp of the trusted consensus state that was updated.

Negative behavior

- ☑ The membership-proof empty value and key should be present in the kvPairs of the output.
- ☑ The vkey should be equal to `MEMBERSHIP_PROGRAM_VKEY` if the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipProof` and `UPDATE_CLIENT_AND_MEMBERSHIP_PROGRAM_VKEY` if the proof type is `IMembershipMsgs.MembershipProofType.SP1MembershipAndUpdateClientProof`.
- ☑ The trusted consensus-state hash of the output should match the stored consensus-state hash.
- ☑ The SP1 proof should be valid.

Function call analysis

- `VERIFIER.verifyProof`
  - **What is controllable?** `proof.vKey`, `proof.publicValues`, and `proof.proof`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

Function: `misbehaviour`

The function verifies the public values via the SP1 verifier, verifies the misbehavior output, and finally updates the frozen state.

Inputs

- `misbehaviourMsg`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid misbehavior SP1 proof — otherwise, the SP1 verifier will revert.
  - **Impact**: Sets the `isFrozen` state to true.

Branches and code coverage

Intended branches

- ☑ If the proof is valid, it updates the `isFrozen` state to true.

Negative behavior

- ☑ The vkey should be equal to `MISBEHAVIOUR_PROGRAM_VKEY`.
- ☑ It should make sure that the trusted consensus state from header1 and header2 is known.
- ☑ The SP1 proof should be valid.

Function call analysis

- `VERIFIER.verifyProof`
  - **What is controllable?** `proof.vKey`, `proof.publicValues`, and `proof.proof`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

5.4. Component: ICS20Transfer

Function: `sendtransfer`

This function initiates an ICS20 transfer.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: `msg_.amount > 0`.
  - **Impact**: The `msg_` contains the transfer parameters.

Branches and code coverage

Intended branches

- ☑ It gets the escrow if it already exists for said token — otherwise, it deploys a new beacon escrow.
- ☑ It calls the escrow's callback.
- ☑ If the token is native, it generates a new denom.
- ☑ If token is returning back to origin, it burns escrow tokens.

Function call analysis

- `_transferfrom(_msgsender(), address(escrow), msg_.denom, msg_.amount)`
  - **what is controllable?** `msg_.denom` and `msg_.amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `escrow.recvcallback(msg_.denom, _msgsender(), msg_.amount)`
  - **what is controllable?** `msg_.denom` and `msg_.amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `OnRecvPacket`

This is a recv callback, executed when receiving an ICS20 IBC packet.

Inputs

- `msg_`
  - **Control**: Arbitrary.

- **Constraints**: Several constraints ensuring valid packet details such as versioning and port — also, packet amount > 0.
- **Impact**: The received IBC ICS20 message.

Branches and code coverage

Intended branches

- ☑ If the packet is a forwarded IBC token, it removes the prefix.
- ☑ If the packet is a native-forwarded IBC token, it removes the prefix and sends back native tokens.
- ☑ If token is returning back to origin, it burns escrow tokens.
- ☑ If receiving a new token, it generates a new IBC-ERC-20 and mints relevant tokens.

Function call analysis

- `IBCERC20(erc20Address).mint(address(escrow), packetData.amount)`
  - **what is controllable?** `packetData.amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_getOrCreateIBCERC20(newDenom, address(escrow))`
  - **what is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `onAckPacket`

This is a packet-acknowledgment callback, refunding tokens back to the sender and minting the tokens if they were burned from the escrow.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Contains the parameters required for the acknowledgement.

Branches and code coverage

Intended branches

- ☑ It refunds tokens and mints if tokens that were sent back were burned instead of escrowed.

Function call analysis

- `IBCERC20(erc20Address).mint(address(escrow), packetData.amount)`
  - **what is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `escrow.send(IERC20(erc20Address), refundee, packetData.amount)`
  - **what is controllable?** `packetData.amount`, granted the relevant packet was sent initially.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `onTimeoutPacket`

This is a time-out–acknowledgment callback, refunding tokens back to the sender and minting the tokens if they were burned from the escrow.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Contains the parameters required for the timeout call.

Branches and code coverage

Intended branches

- ☑ It refunds tokens and mints if tokens that were sent back were burned instead of escrowed.

Function call analysis

- `IBCERC20(erc20Address).mint(address(escrow), packetData.amount)`
  - **what is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `escrow.send(IERC20(erc20Address), refundee, packetData.amount)`
  - **what is controllable?** `packetData.amount`, granted the relevant packet was sent initially.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **what happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.5. Component: ICS26Router

Function: `sendPacket`

This function sends an IBC packet, committing a packet to storage.

Inputs

- `msg_`
  - **Control**: Only the IBC app can send this.
  - **Constraints**: Time-out must be valid.
  - **Impact**: The IBC message.

Branches and code coverage

Intended branches

- ☑ The packet should be committed.

Negative behavior

- ☑ Counterparty should exist.
- ☑ Sequence should increase as expected.
- ☑ Same packet cannot be committed/sent twice.

Function: `RecvPacket`

This function receives a packet and commits a packet acknowledgment.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: Packet must have payload length equal to 1.
  - **Impact**: The received IBC message.

Intended branches

- ☑ The SP1 proof must be verified to ensure that the packet exists on the counterparty chain.
- ☑ Packet acknowledgment is committed.
- ☑ Relevant callback is called.

Negative behavior

- ☑ The `timeoutStamp` should not be expired.
- ☑ Destination chain info should match the source client.
- ☑ It cannot receive twice (`setPacketReceipt`).

Function: `ackPacket`

This function acknowledges the IBC packet — doing so deletes the packet commitment.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: Counterparty info must match.
  - **Impact**: The message to verify and acknowledge.

Intended branches

- ☑ Packet commitment is deleted.
- ☑ Relevant callback is called.
- ☑ Verify the counterparty has committed the packet acknowledgment.

Negative behavior

- ☑ Packet cannot be acknowledged twice.

Function: `timeoutPacket`

This function times out the IBC packet — doing so deletes the packet commitment.

Inputs

- `msg_`
  - **Control**: Arbitrary.
  - **Constraints**: Counterparty info must match.
  - **Impact**: The message to time out.

Intended branches

- ☑ Relevant callback is called.
- ☑ Counterparty should exist.
- ☑ Packet commitment is deleted.
- ☑ If the commitment must exist, it must exist past the time-out timestamp.

Negative behavior

- ☑ Commitment must not exist in the counterparty chain.

5.6. Component: EurekaHandler

Function: `transfer`

This function initiates an ICS20 transfer.

Inputs

- `amount`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens to transfer.
- `transferParams`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: These are the transfer parameters which contain the details about the transfer.
- `fees`
  - **Control**: Fully controlled by caller.

- **Constraints**: `block.timestamp < fees.quoteExpiry`.
- **Impact**: This is the fees for the transfer.

Branches and code coverage

Intended branches

- ☑ The tokens are transferred to this contract and then approved to the `ics20Transfer` contract. Finally `sendTransferWithSender` is called on the `ics20Transfer` contract.

Negative behavior

- ☐ Revert if fee quote is expired.

Function: `swapAndTransfer`

Swaps the token to an output token and initiates an ICS20 transfer.

Inputs

- `swapInputToken`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The input token to swap.
- `swapInputAmount`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of input tokens to swap.
- `swapCalldata`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: Call data for the swap function call.
- `transferParams`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: These are the transfer parameters which contain the details about the transfer.
- `fees`
  - **Control**: Fully controlled by caller.
  - **Constraints**: `block.timestamp < fees.quoteExpiry`.

- **Impact**: This is the fees for the transfer.

Branches and code coverage

Intended branches

- ☑ The function first transfers the input swap token to itself, performs a swap to the `transferParams.token` token and then initiates an ICS20 transfer.

Negative behavior

- ☐ Revert if fee quote is expired.
- ☑ Revert if the output amount after the swap doesn't cover the total fees.

Function: `lombardTransfer`

This function initiates an ICS20 transfer for lbtc.

Inputs

- `amount`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of tokens to transfer.
- `transferParams`
  - **Control**: Fully controlled by caller.
  - **Constraints**: N/A.
  - **Impact**: These are the transfer parameters which contain the details about the transfer.
- `fees`
  - **Control**: Fully controlled by caller.
  - **Constraints**: `block.timestamp < fees.quoteExpiry`.
  - **Impact**: This is the fees for the transfer.

Branches and code coverage

Intended branches

- ☑ The function transfers the amount and fees from the caller, gets the voucheramount from `lbtcVoucher` and initiates an ICS20 transfer.

Negative behavior

- ☐ Revert if fee quote is expired.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Cosmos Chain.

During our assessment on the scoped IBC Eureka contracts, we discovered five findings. Two critical issues were found. One was of high impact, one was of medium impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.