Formal Verification Report of dcSpark SidechainBridge

Summary

This document describes the specification and verification of the sidechain bridge of the Milkomeda protocol using the Certora Prover. The work was undertaken from 07/09/2022 to 12/10/2022. The latest commit that was reviewed and run through the Certora Prover was ce5827a.

The scope of our verification was the SidechainBridge contract.

The Certora Prover verified that the implementation of the sidechain bridge is correct with respect to the formal rules written by the Certora team. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. The development team at dcSpark promptly corrected all issues, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines the high level specification of Milkomeda's sidechain bridge.

List of Main Issues Discovered

Severity: Critical

Issue:	**Protocol hijack by single validator**
Rules Broken:	transacationDoesNotExistsImpliesNotConfirmed
Description:	In the `removeTransaction()` function, removing a transaction does not set all its confirmations to false. This leaves the `transactionId` that was removed as pre-confirmed, so a following transaction to that `transactionId` will be immediately executed (as it already has enough votes). An attacker could leverage this to call `upgradeContract()` function and take control over the protocol
dcSpark Response:	The vulnerable removeTransaction function was removed in aa1a4.

Severity: Critical

Issue:	Asset value upgrade
Rules Broken:	
Description:	Calling `upgradeAsset()` on an asset from a low-value token to a high-value token would greatly increase the value a user already owns on that `assetId`. This improper increase would cause insolvency.
dcSpark Response:	The vulnerable updateAsset function was removed in 62e64.

Severity: Critical

Issue:	Double execution of same request
Rules Broken:	
Description:	A malicious validator could migrate a proposal that it wants to duplicate and not sign it on-chain. The malicious validator would call the `migrate()` when (`quorum` - 1) signatures are obtained. This would enable them to sign the proposal off-chain and send it to the other side, while the migrated proposal would also likely be signed and sent. Unless the migration process on this side is tracked very tightly by all good validators and the other side's bridge, it would be impossible to distinguish between such a malicious duplication and an actual case of two proposals with the same request.
dcSpark Response:	The vulnerable migrateProposal function was removed in b3179.

Severity: High

Issue:	Insolvency
Rules Broken:	checkRewards

Issue:	Insolvency
Description:	In the `withdrawRewards()` function there is a calculation of the "share" of the rewards for each validator (which is `rewardsPot / validators.length`). Subsequently, there is a for loop on the validators array that sends money to each validator. If the validator's fallback (or receive) function contains a call to the `voteForTransaction` or `executeTransaction` functions that would result in the addition of a validator (by executing the transaction), then the for loop of `withrawRewards()` would repeat over an extra validator and send one share too many. This extra loop iteration would cause insolvency.
dcSpark Response:	The issue was fixed in 07042 by disallowing reentrant calls to all key contract functions.

Severity: High

Issue:	DOS by a validator frontruns transactions
Rules Broken:	nonDOS
Description:	A malicious validator can `voteForTransaction` with the id of the future transaction he wants to prevent and then this transaction will not be able to be submitted. This is a DOS to any transaction that the malicious validator wants to prevent and he can even prevent his removal because he can prevent this transaction.
dcSpark Response:	The bridge contract is only deployed to networks where consensus is controlled by the same set of validators that controls the contract. For example, the Milkomeda C1 sidechain is run by the bridge contract validators operating under the IBFT consensus. Under honest majority assumption a corrupt validator will not be able to block their removal from the contract for instance because they can be ejected as a validator on the consensus level (bypassing any communication via EVM transactions).

Severity: **High**

Issue:	Vote on canceled Unwrapping Proposal
Rules Broken:	

Issue:	Vote on canceled Unwrapping Proposal
Description:	Validators can vote on a canceled Unwrapping Proposal and may lead to its execution if the quorum is reached. This can happen because there is no way to distinguish between a canceled UPT and a confirmed UPT.
dcSpark Response:	The vulnerable migrateProposal function was removed in b3179.

Severity: **High**

Issue:	DOS by Reset all voting on unwrappingProposals
Rules Broken:	
Description:	Validators can reset all votes on an Unwrapping Proposal and lead to a migrating proposals DOS by calling `migrateUnwrappingProposal(uint256)`.
dcSpark Response:	The vulnerable migrateProposal function was removed in b3179.

Severity: **Medium**

Issue:	Vote with empty witness
Rules Broken:	
Description:	A Validator can vote multiple times on the same UnwrappingProposalTransaction by calling to `voteOnUnwrappingProposalTransaction(uint256, bytes)` with an empty bytes array as a witness which can lead to confirmation of an unwrapping `UnwrappingProposalTransaction` by a single validator.
dcSpark Response:	Votes with empty witness disallowed in 00389.

Severity: **Low**

Issue:	unwrapping proposal getting stuck after quorum change
Rules Broken:	

Issue:	unwrapping proposal getting stuck after quorum change
Description:	If an unwrapping proposal has vote count 'x' that is less than the quorum and the quorum is changed to be less than 'x', the unwrapping proposal will never be executed and will remain stucked because it is flaged as not closed but it's voting had reached the quorum
dcSpark Response:	Fixed by adding `confirmUnwrappingProposalTransaction`. to confirm unwrapping proposal transactions that it's votes had reached the quorum

Severity: **Low**

Issue:	Replace to zero validator
Rules Broken:	zeroNotValidator
Description:	The notnull modifier was missing in the `replaceValidator` function meaning a majority could add 0x0 as a validator by mistake.
dcSpark Response:	Fixed in 35a12.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Summary of Formal Verification

Overview of Sidechain bridge contract

The following description is taken from the milkomeda-validator repository:

The sidechain bridge contract is a part of the Milkomeda protocol. The Milkomeda protocol provides cross-chain interoperability between two blockchains: a main chain (currently only a Cardano chain can be configured) and a special EVM chain called the sidechain.

The interoperability is provided by a set of validators. The validators are actors which function akin to a decentralized organization. The validators collectively have the power to do actions on both the mainchain and the sidechain which implement the interoperability.

The validators have more power over the sidechain: they run the consensus protocol of the sidechain (the sidechain is a permissioned Hyperledger Besu network). In addition they are recognized as the owners of the bridge smart contract. In a similar manner they are recognized as owners of a smart contract / multisig address on the mainchain.

Currently the only allowed form of chain interoperability is asset transfer. The asset transfer is implemented by the end user sending an asset to a contract on one of the chains. The asset becomes locked, the validators notice this and release a corresponding asset on the other chain.

List of contracts

There are multiple source files, but the bridge really only consists of two deployed contracts: the SidechainBridge and the Proxy. Moreover, these two contracts function as a proxy-implementation pair, therefore they are often talked about as a single contract: the sidechain bridge contract.

The split into multiple source files via inheritance is only an attempt to organize source code in Solidity. The files are as follows:

- **Types.sol**: holds type definitions for the bridge contract.
- **State.sol**: holds state variable definitions for the bridge contract. Keeping all state in single place is important for the safety of the proxy pattern.
- **Multisig.sol**: holds functions to enable the validator set to work as a decentralized organization: it defines what it means that a certain action can be only done by validator majority; it enables removing and adding new validators, changing quorum, and signing-off and executing any sidechain transaction proposals (e.g. those releasing an asset held by the bridge, thus completing a cross-chain asset transfer).
- **TokenRegistry.sol**: different blockchains have wildly different definitions of assets. Milkomeda protocol will match and enable transfers of assets based on validators' decisions. Once validators vote a mainchain-sidechain asset pair identification into the bridge (which happens via functions in this source file), that asset can be locked and unlocked on the sidechain bridge contract.
- **Proxy.sol**: the main bridge contract which holds all the bridge state, but delegates all the calls to the current logic contract.

- **SidechainBridge.sol**: the main bridge contract which holds all the logic (functions) that can be applied to the bridge state. Beyond the functionality inherited from the source files above, this file defines support for unwrapping requests, i.e. moving assets from sidechain to mainchain. An end user can supply their asset to the contract and make a transfer request -- if the asset is locked successfully, validator will provide an encoded mainchain transaction (unlocking an appropriate mainchain asset) to this (i.e. sidechain) contract and keep adding signatures until it's usable on the mainchain.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).

![](https://i.imgur.com/rDhiM7e.png =20x20) indicates the rule was violated under one of the tested versions of the code.

✍️ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (<due to other issues, low priority>) .

We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax {p} (C1 ~ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1~op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Formal Properties for Sidechain bridge contract

The following properties were written and verified by Certora.

Functions - MultisigHarness.sol

```
IsExecuted(bytes32 id) : bool
```

Getter of bool in transaction struct.

```
getLength() : uint256
```

Getter of length of validators.

```
getQuorum() : uint256
```

Getter of quorum.

```
getIsValidator(address) : bool
```

Getter of the mapping of isValidator.

```
getVote(bytes32, address) : bool
```

Getter if a transaction has been voted by the address given.

```
getValidator(uint256) : address
```

Getter of validator at index given.

```
voteForTransaction_harness(bytes32, address, uint256, bool)
```

Same as `voteForTransaction` but in the harness

```
executeTransaction_harness(bytes32)
```

This function executes the transaction but also can call all the inner functions (add, remove, replace...) that are not modeled by the tool (and now are being modeled).

Functions - SidechainBridgeHarness.sol

```
setAsset(bytes32, address) : bool
```

Sets new asset with a given assets Id in the tokenRegistry.

```
setReqMinValue(bytes32, uint256) : bool
```

Sets the minimumValue for the given asset in the tokenRegistry.

```
setTokenType(bytes32) : bool
```

Sets the given asset as ERC20 in the tokenRegistry.

```
isValidatorExist(address) : bool
```

Returns true if the given validator exists in the validators array.

```
isUnwrappingProposalExists(uint256) : bool
```

Returns true if there is an unwrappingProposal with the given ID.

```
isUnwrappingProposalTransactionExists(uint256) : bool
```

Returns true if there is an encoded transaction for the given uwrappingProposal ID.

```
isUnwrappingProposalWitnessExist(uint256, address) : bool
```

Retuns true if the given witness for the given unwrappingProposal exists.

```
isProposalClosed(uint256) : bool
```

Returns true if the given unwrappingProposal is closed.

```
getValidatorsCount() : uint256
```

Returns the number of validators that exist in the bridge.

Functions - RewardsHarness.sol

```
getBalance() : uint256
```

Returns the balance of the Rewards contract.

```
getRewardsPot() : uint256
```

Returns the rewardsPot balance.

```
withdrawRewards_harness() : void
```

same as withdrawRewards but also can call `executeTransaction` so the reentrancy will be modeled.

Properties

1. Integrity Of Vote On Unwrapping Proposal Transaction ✔ Following a call for voteOnUnwrappingProposalTransaction, if quorum is reached, the proposal should be closed, the reward pot should increase, and the same validator can't vote on unwrapping proposal transaction with a witness that already exists.

```
{
    witnessCountBefore :=
getUnwrappingProposalTransactionWitnessCount(proposalId),
    isWitnessExists := isUnwrappingProposalWitnessExist(proposalId,
e.msg.sender),
    rewardsPotBefore := rewardsPot,
    isProposalClosedBefore := isProposalClosed(proposalId)
}
    voteOnUnwrappingProposalTransaction(proposalId, witness);
{
    quorum = witnessCountAfter ∧ ¬isProposalClosedBefore ⇒
(isProposalClosed(proposalId) ∧ rewardsPotBefore < rewardsPotAfter),
    isWitnessExists ⇒ witnessCountBefore = witnessCountAfter,
    ¬isWitnessExists ⇒ witnessCountAfter = witnessCountBefore + 1
}
```

2. Break the Integrity Of Vote On Unwrapping Proposal Transaction ✔ Rule to verify that no function can break the integrityOfVoteOnUnwrappingProposalTransaction.

```
{
    witnessCountBefore :=
getUnwrappingProposalTransactionWitnessCount(proposalId),
    isWitnessExists := isUnwrappingProposalWitnessExist(proposalId,
e.msg.sender),
    rewardsPotBefore := rewardsPot(),
    isProposalClosedBefore := isProposalClosed(proposalId)
}
    <invoke any method f>,
    voteOnUnwrappingProposalTransaction(proposalId, witness)
{
    isProposalClosedBefore ⇒ isProposalClosed(proposalId),
    witnessCountAfter = quorum ∧ ¬isProposalClosedBefore ⇒
(rewardsPotBefore < rewardsPotAfter ∨ f = withdrawRewards()) ∧
isProposalClosed(proposalId),
    witnessCountAfter < quorum ∧ ¬isProposalClosedBefore ⇒
¬isProposalClosed(proposalId),
    isWitnessExists ⇒ witnessCountBefore = witnessCountAfter,
    ¬isWitnessExists ⇒ witnessCountAfter = witnessCountBefore + 1
}
```

3. Can't Vote More Than Once ✓

Validators cannot vote on the same proposal more than once.

```
{
    witnessCountBefore :=
getUnwrappingProposalTransactionWitnessCount(proposalId),
}
    voteOnUnwrappingProposalTransaction(proposalId, witness),
    voteOnUnwrappingProposalTransaction(proposalId, witness)
{
    witnessCountAfter <= witnessCountBefore + 1
}
```

4. Vote On Closed Unwrapping Proposal Transaction ✓

Voting on a closed unwrapping proposal transaction must have no effect except adding a witness.

```
{
    isProposalClosed(proposalId) := True,
    witnessCountBefore :=
getUnwrappingProposalTransactionWitnessCount(proposalId)
}
    voteOnUnwrappingProposalTransaction(proposalId, witness)
{
    witnessCountBefore = witnessCountAfter + 1,
    rewardsBefore = rewardsAfter,
    isProposalClosed(proposalId) = True
}
```

5. Integrity Of MigrateUnwrappingProposal ✓

After call to migrateUnwrappingProposal, the old unwrapping proposal must be closed and a new unwrapping proposal with the same request must be created.

```
{

}
    migrateUnwrappingProposal(oldProposalId)
{
    ¬isUnwrappingProposalExists(oldProposalId) ∧
    unwrappingProposalTransactionExists(oldProposalId)
}
```

6. Who Can Change Quorum ✓

Rule to verify that only the bridge with addValidator, removeValidator, and changeQuorum can change the quorum.

7. Who Can Change Rewards Pot ✓

Rule to verify that confirmUnwrappingProposalTransaction, executeTransaction or only a validator with voteOnUnwrappingProposalTransaction, voteForTransaction, withdrawRewards can change the rewards pot.

8. Integrity Of Withdraw Rewards ✓ Rule to verify that the rewards are distributed.

```
{
    balanceBefore := getBalance(),
    amountToDistribute := getRewardsPot() − (getRewardsPot() %
validatorsCount)
}
    withdrawRewards_harness(args)
{
    amountToDistribute ≠ 0 ⇒ getBalance() < balanceBefore
}
```

9. Transaction Does Not Exists Implies That Its Not Confirmed ✓

This invariant checks that a transaction that does not exist has no votes.

```
¬transactionExists(id)) ⇒ ¬getVote(id, a)
```

10. Validator Is Valid ✓

Invariant to verify that an address that is in the validators array is marked as a validator.

11. Validator Is Not Zero ✓

Invariant to verify that none of the validators is address zero.

12. Validators Uniqueness ✓

Invariant to verify that the validators array is unique (without duplicated validators).

13. validInValidators ✓

Invariant to verify that each of the validators is valid.