arbitrary./execution

Milkomeda

LIQUID STAKING SECURITY ASSESSMENT

June 14, 2023

Prepared For:
Sebastien Guillemot, Milkomeda
Prepared By:
Arbitrary Execution

Changelog:

December 27, 2022	*Initial report delivered*
February 28, 2023	*Final report delivered*
June 14, 2023	*Follow-on addendum delivered*

TABLE OF CONTENTS

EXECUTIVE SUMMARY

This report contains the results of Arbitrary Execution's security assessment of the Milkomeda Liquid Staking smart contracts. Milkomeda is a protocol that brings EVM capabilities to non-EVM blockchains. The protocol allows users to bridge assets from Cardano and Algorand to an EVM-compatible sidechain. The Liquid Staking smart contracts will enable users to stake their wrapped assets on the Milkomeda sidechain in order to accrue layer 1 (L1) rewards. Two Arbitrary Execution (AE) security researchers conducted this review over a 1.5-week period, from December 12, 2022 to December 21, 2022. The audited commit was `d7496b1c4bce8dba49f3309ce0aed846d2b10fe9` in the `main` branch of the `dcSpark/liquid-staking` repository. The complete list of files in scope is located in Appendix B. This repository was private at the time of the engagement, so hyperlinks may not work for readers without access.

The team performed a detailed, manual review of the codebase with a focus on Milkomeda's `StakedMilkAda`, `StakingSmartContract`, and `Pillage` contracts. In addition to manual review, the team used Slither with AE's proprietary Slither detectors for automated static analysis.

The assessment resulted in findings ranging in severity from critical to note (informational). A critical finding in the access control for contract upgrades allows an arbitrary user to perform upgrades. Two high severity findings impact the staking and unstaking calculations. Medium and low severity findings impact the way the protocol generates transaction IDs and handles transactions. The note severity findings contain observations regarding code hygiene, documentation, and other best practices.

FIX REVIEW UPDATE

The Milkomeda team has fixed all major issues identified in the engagement. Three note findings were marked acknowledged, partially fixed, or not fixed. The full breakdown of fixes can be found in the Fixes Summary section.

FIX REVIEW PROCESS

After receiving fixes for the findings shared with the Milkomeda team, the AE team performed a review of each fix. Each pull request was scrutinized to ensure that the core issue was addressed, and that no regressions were introduced with the fix. A summary of each fix review can be found in the *Update* section for a finding. For findings that the Milkomeda team chose not to address, the team's rationale is included in the update.

Two fix review PRs contained additional code changes unrelated to the original finding. These additional changes were reviewed by AE in a follow-on engagement after the fix review.

FOLLOW-ON REVIEW UPDATE

During the months of April and May 2023, Milkomeda engaged AE to review additional changes to the Liquid Staking codebase. The AE team reviewed the changes, relayed feedback to the Milkomeda team, and reviewed the PR implementing the feedback. During the follow-on review, the Milkomeda team identified a bug that was verified by the AE team. The bugfix was also reviewed during this engagement.

No additional findings were identified that impact the security of the Liquid Staking protocol. A full summary of the follow-on review is located in Appendix C.

VULNERABILITY STATISTICS

Severity	Count
Critical	1
High	2
Medium	1
Low	2
Note	14

FIXES SUMMARY

Finding	Severity	Status
C01	Critical	Fixed in pull request #13
H01	High	Fixed in pull request #14
H02	High	Fixed in pull request #35
M01	Medium	Fixed in pull request #16
L01	Low	Fixed in pull request #19
L02	Low	Fixed in pull request #24
N01	Note	Fixed in pull request #17
N02	Note	Acknowledged
N03	Note	Fixed in pull request #18
N04	Note	Fixed in pull request #20
N05	Note	Fixed in pull request #22
N06	Note	Fixed in pull request #21
N07	Note	Fixed in pull request #23
N08	Note	Fixed in pull request #32
N09	Note	Fixed in pull request #33
N10	Note	Fixed in pull request #25
N11	Note	Fixed in pull request #26
N12	Note	Fixed in pull request #27
N13	Note	Fixed in pull request #28
N14	Note	Not fixed

AUDIT OBJECTIVES

AE focuses on common high-level goals for all security audit engagements. During this engagement, the AE team:

- Identified smart contract vulnerabilities
- Evaluated adherence to Solidity best practices

The Milkomeda team provided additional specific goals to guide the engagement. The AE team also:

- Verified access controls for UUPSUpgradeable contract upgrades
- Verified the access controls defined in the `Bridge` contract

AE examined access control through a combination of manual review and writing Foundry unit tests.

OBSERVATIONS

There is little documentation in the `liquid-staking` repository, and existing documentation lives in a separate repository from the code. Updating the `liquid-staking` repository's README.md with additional information would aid developers and auditors in future engagements.

The `StakedMilkAda` contract implements the ERC20 interface, but has modified the behavior of several functions including `transfer`, `transferFrom`, and `balanceOf`. The `stakingSmartContract` and `StakedMilkAda` contracts depend on one another for balance calculations. Implementing an interface is different than being complaint with the specification, and the documentation should make this distinction clear.

The Liquid Staking security model draws inspiration from the Milkomeda Bridge, where validators hold the ability to propose and execute privileged functions through the Bridge multisig contract. The Liquid Staking contracts rely on the same set of validators as the Milkomeda Bridge, but implement their own transaction execution mechanism. The Milkomeda Bridge multisig can add and remove validators, but otherwise cannot interact with the Liquid Staking contracts.

SYSTEM OVERVIEW

TOKENS

Native tokens on the Milkomeda sidechain are wrapped versions of their respective L1 assets. Their names are prefixed with `milk`.

MILKADA

`milkAda` is wrapped ADA. It can be used as a native token on the Milkomeda sidechain.

STMILKADA

`stMilkAda` is a token representing "Staked `milkAda`". Users call the `stake` method on the `StakingSmartContract`, lock up `milkAda`, and receive `stMilkAda` in return. This token represents a user's share of the entire staked `milkAda` pool.

USER CATEGORIES

USERS

Users on the Milkomeda sidechain can transact, interact with contracts, and stake tokens. Users do not hold any special roles in the context of the Liquid Staking smart contracts. Liquid Staking users can either be EOAs or smart contracts.

PRIVILEGED ROLES

VALIDATORS

Validators run the consensus protocol of the Milkomeda sidechain. In the context of the Liquid Staking contracts they can call functions that contain calls to `_assertValidator(msg.sender)`. These protected functions will execute after reaching `quorum` validator votes. Actions that are restricted to validators include:

- Changing the address of the `stakedMilkAda` and `Pillage` contracts in the `StakingSmartContract`
- Pausing and unpausing the `StakingSmartContract` and `StakedMilkAda` contracts
- Performing administrator functions on the `Pillage` contract

SYSTEM COMPONENTS

The liquid Staking system has three major components:

- The `StakingSmartContract`, which is responsible for receiving deposits and relaying information about the staking pool to the `StakedMilkAda` contract.
- The `StakedMilkAda` contract, which issues an ERC20 compatible token to represent staked `milkAda`.
- The `Pillage` contract, which is responsible for taking ("pillaging") rewards from smart contracts that are unable to withdraw their rewards.

The `StakingSmartContract` and `StakedMilkAda` contracts use a double accounting system. The `StakingSmartContract` tracks the total deposited `milkAda`, and the `StakedMilkAda` contract tracks the percentage ownership of each user in relation to the total amount of `milkAda` deposited.

The `StakingSmartContract`, `StakedMilkAda`, and `Pillage` contracts are all `UUPSUpgradeable`, so upgrade logic is handled in the implementation contracts, not the proxy.

ACCESS CONTROL MODEL AND VALIDATOR VOTING

The Liquid Staking contracts draw from the access control model of the Milkomeda Bridge contracts. Administrator transactions are proposed by validators through calls to `_addTransaction` and execute after `quorum` validators confirm the transaction via `_confirmTransaction`.

STAKEDMILKADA

The `StakedMilkAda` contract is an ERC20 contract with additional logic for token transfers. This token tracks the percentage ownership of each user in relation to the staking pool. It coordinates with the `StakingSmartContract` to calculate a user's share of the pool when transferring `stMilkAda` tokens.

STAKINGSMARTCONTRACT

The `StakingSmartContract` tracks the total amount of `milkAda` deposited to the staking pool. It handles deposits and accounting when users transfer their `stMilkAda` tokens. The `StakingSmartContract` is the only address that can call `mint` and `burn` on the `StakedMilkAda` contract.

PILLAGE

The `Pillage` contract enables the Milkomeda DAO to claim staking rewards from inactive accounts as protocol revenue. It has permissions to call the `removeRewardsOnBehalf` function on the `StakingSmartContract`. This function allows the `Pillage` contract to withdraw rewards from smart contract accounts that do not implement an `ableToWithdrawRewards` function.

FINDINGS

CRITICAL SEVERITY

[C01] CONTRACT MANAGEMENT FUNCTIONS LACK ACCESS CONTROL

Contracts in the project use the `authorizeUpgrade` modifier for access control. The `onlyProxy` modifier is used with the intention of limiting administrative functions to the contract itself, after reaching `quorum` votes from validators:

```
function _authorizeUpgrade(
    address newImplementation
) internal override(Bridge, UUPSUpgradeable) onlyProxy {}
```

However, the `onlyProxy` modifier does not enforce this. According to OpenZeppelin's documentation:

> Check that the execution is being performed through a delegatecall call and that the execution context is a proxy contract with an implementation (as defined in ERC1967) pointing to self

As a result, any user is able to call the following functions:

- From `Bridge.sol`:
  - `_executeBridgeUpdate` (public)
- From `StakingSmartContract.sol`:
  - `_executePillagerUpdate` (public)
  - `_executeStakedMilkAdaUpdate` (public)
  - `_executePuase` (public)
  - `_executeUnPuase` (public)
- From `StakedMilkAda.sol`:
  - `_executeStakingSCUpdate` (public)
  - `_executePuase` (public)
  - `_executeUnPuase` (public)
- From `Pillage.sol`:
  - `_executeUpgradeTo` (public)
  - `_executeStakingSCUpdate` (public)

RECOMMENDATION

Consider making contract management functions callable only from the contract itself via the `_confirmTransaction` function.

UPDATE

Fixed in pull request #13 (commit hash `0288bb342568eb03a12192090f4d53b3fb5c9968`), as recommended.

HIGH SEVERITY

[H01] SHARE ARITHMETIC UNFAIRLY FAVORS EARLY DEPOSITORS

The share conversion calculation in the function `getSharesByPooledMilkAda` allows the first depositor to manipulate the amount of shares received by following depositors. If the first depositor stakes 1 wei and accredits some amount of rewards in the `accreditToPool` function, the next depositor's stake will include those rewards in the divisor of its share calculation.

Consider a scenario where Milkomeda wants to deposit 11 Ether as an initial deposit, but before they are able to do so, an attacker deposits 1 wei and accredits 10 Ether. Milkomeda will receive 1 share for the 11 Ether deposit, while the attacker receives 1 share after only contributing 1 wei to the pool. If a 10 Ether reward is dispersed, the attacker can withdraw over 10 Ether.

A common way to mitigate risk from "donation attacks" is by making a sufficiently large first deposit from a trusted party.

RECOMMENDATION

Consider staking some balance of tokens during the initialization of the contract in order to avoid donation attacks.

UPDATE

Fixed in pull request #14 (commit hash `3dc9de8e609f9af121095d4ab269a7976e405e7d`), as recommended.

[H02] INCORRECT ROUNDING AND DEFAULTING WHEN STAKING AND UNSTAKING

The rounding performed during staking and unstaking operations is inconsistent with recommendations from standards such as EIP-4626:

> *If (1) it's calculating how many shares to issue to a user for a certain amount of the underlying tokens they provide or (2) it's determining the amount of the underlying tokens to transfer to them for returning a certain amount of shares, it should round down.*
>
> *If (1) it's calculating the amount of shares a user has to supply to receive a given amount of the underlying tokens or (2) it's calculating the amount of underlying tokens a user has to provide to receive a certain amount of shares, it should round up.*

When staking, the `stake` function defaults the amount of shares to mint to `msg.value` when `getSharesByPooledMilkAda` returns zero. However, when computing the number of shares to burn in `_unstake`, the value returned by `getSharesByPooledMilkAda` is always used.

The calculation in the `getSharesByPooledMilkAda` function also strictly rounds down. Because of this, it is possible for it to return zero, specifically when provided with a small input amount while the total rewards are high compared to the total deposits.

When taken in combination, these issues allow a malicious user to drain the rewards pool. The user can, if necessary, make use of a flash loan to first inflate the amount of rewards. For a sufficiently high value of `totalRewards`, `getSharesByPooledMilkAda` will return zero, and the malicious user will receive `msg.value` shares upon calling `stake`.

Then, by repeatedly staking and unstaking, the user will be able to reclaim both any borrowed funds and funds already in the rewards pool.

RECOMMENDATION

Consider removing the edge case in the `stake` function that mints `msg.value` rewards and following the rounding guidelines in ERC4626.

UPDATE

Fixed in pull request #35 (commit hash `cd000da6d45daae0f43bcfdedac696bde29c9735`), as recommended.

MEDIUM SEVERITY

[M01] TRANSACTION VOTING WITH MISMATCHED TRANSACTION

Transactions are added to the `transactions` mapping via the `_addTransaction` function on line 71. This function either adds a new transaction or returns an existing one, depending on the `_transactionId` provided. When the provided `_transactionId` exists, the `_data` and `_destination` fields passed in are checked against the existing values associated with that `_transactionId`.

However, this check will only revert if both the `_destination` and `_data` fields do not match the existing values. This means if one field is correct and the other is incorrect, the function will not revert.

Moreover, it is possible for a malicious validator to front-run another validator's transaction with a transaction that overlaps the `_transactionId` and one of the two parameters. Since one of the parameters will match, the front-run validator's transaction will not revert and will instead count as a vote towards the malicious transaction.

RECOMMENDATION

Consider changing the check to revert on a mismatch of transaction `destination` or `data`.

UPDATE

Fixed in pull request #16 (commit hash `6571e6a03c07f3ad8e6dcdb822c0395c3d1df13f`), as recommended.

LOW SEVERITY

[L01] INCONSISTENT REWARDS CALCULATIONS

In `StakingSmartContract.sol`, the function `_withdrawRewards` implements the arithmetic for computing rewards based on an amount of shares. The function `removeRewardsOnBehalf` contains similar arithmetic for the same purpose.

The rewards calculation in `_withdrawRewards` contains extra logic to handle the cases where the user's deposit is zero or larger than the calculated `percentageMilkAda`, but the version in `removeRewardsOnBehalf` lacks these checks. This can create situations where the `removeRewardsOnBehalf` function will revert with an arithmetic underflow, while the `_withdrawRewards` function will not.

The underflow occurs when `userDeposits` is greater than the left-hand-side of the subtraction on line 158. Under this condition, the `Pillage` contract will be blocked from reaping rewards on behalf of the account.

RECOMMENDATION

Consider factoring the rewards calculation into a separate helper function, or at a minimum ensuring the calculation is consistent across the `_withdrawRewards` and `removeRewardsOnBehalf` functions.

UPDATE

Fixed in pull request #19 (commit hash `e222237169140e9cc8846c70e8d76560f59f49b1`), as recommended.

Code changes unrelated to `StakingSmartContract.sol` were introduced in pull request #19. These changes were reviewed during the follow-on engagement.

[L02] TRANSACTION IDS ARE INSECURE

In the `Base` contract, the `transactions` mapping holds `Transaction` objects. When transactions are submitted, they are first populated via the `_addTransaction` function and confirmed once via the internal `_confirmTransaction` function. The `_transactionId` is used when referencing transactions, and is passed around to both of these functions.

Because a `_transactionId` is supposed to be a unique identifier, the `_addTransaction` function checks and reverts when a submitted transaction has an existing `_transactionId` but different `_data` and `_destination` values. However, the `_transactionId` parameter is normally provided by a validator rather than uniquely generated based on the transaction.

Since it is possible to submit a `Transaction` with an arbitrary `_transactionId`, a malicious validator can front-run any other validator with a `Transaction` that shares the `_transactionId` but contains different data, causing the behaving validator to revert.

Similarly, the `generateId` function can be front-run, as it is possible to calculate the `_transactionId` that will be associated with the new contract's address.

These front-run attacks makes it possible for a single bad acting validator to prevent other validators from submitting transactions.

RECOMMENDATION

Consider securely generating the `_transactionId` of transactions to remove the potential for denial of service attacks. This can safely be done by hashing a combination of a transaction with something time-based such as `block.timestamp` or `block.hash`.

UPDATE

Fixed in pull request #24 (commit hash `3f448741f4d250f11ac91b44a0227161cc0adbd6`), as recommended.

NOTE SEVERITY

[N01] IMPLEMENTATION VERSUS SPECIFICATION MISMATCH

The following docstrings reference a check on the `ACCREDITOR_ROLE` while no check exists in the corresponding function:

- `StakingSmartContract.sol`, line 129
- `StakingSmartContract.sol`, line 140

The following docstrings reference the `MINTER_ROLE` and `BURNER_ROLE` roles which have been deprecated in favor of the `canMintAndBurn` modifier:

- `StakedMilkAda.sol`, line 76
- `StakedMilkAda.sol`, line 91

The following docstring should be updated because the amount of `stMilkAda` received from the `stake` function is not 1 to 1. (Users receive tokens proportional to their ownership of `milkAda` in the staking pool):

- `StakingSmartContract.sol`, line 180

RECOMMENDATION

Consider updating the docstrings to reflect the implementation.

UPDATE

Fixed in pull request #17 (commit hash `e50aeac68012bb444f85e23bcd43e9ce48dbf7d3`), as recommended.

[N02] NON-COMPLIANCE WITH ERC20 STANDARD

`StakedMilkAda` is an ERC20-compatible token used to represent the amount of staked `MilkAda` and the number of shares an address is entitled to. The project documentation states the following:

> *stMilkADA is fully compatible with the standard ERC20. However, there have been slight modifications to the most known methods from this standard, such as transfer, transferFrom and balanceOf. On top of that, the mint and burn methods have been protected by roles: only the Staking Smart Contract can call those methods. While it could have been possible to inherit the ERC20 contract from Open Zeppelin, it was more than necessary to only comply with the interface (IERC20) since several methods have custom implementations.*

The custom implementations of functions can lead to confusion and may not be handled correctly by other protocols.

One of the functions that is non compliant is the `transfer` function. The ERC20 standard states the following:

> *transfer: Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend.*

However when the `transfer` function is called on `StakedMilkAda`, the amount transferred is not the `amount` passed in, but rather a number of shares calculated from that `amount` parameter.

RECOMMENDATION

Consider importing the OpenZeppelin ERC20 contract and overriding necessary functions. Also consider renaming functions to clearly denote that they deviate from the ERC20 standard.

UPDATE

Acknowledged. Milkomeda's statement for the issue:

> *While it could have been possible to inherit the ERC20 contract from Open Zeppelin, it was more than necessary to only comply with the interface (IERC20) since several methods have custom implementations.*

[N03] INCORRECT ADDRESS RAISED IN ERROR

The `assertValidator` function in the `Bridge` contract reverts with an `InvalidValidator(msg.sender)` error if the `_validator` parameter fails the `isValidatorOnBridge` check. Because this is a `public` function, `msg.sender` will not necessarily match the `_validator` parameter and the error will be misleading.

RECOMMENDATION

Consider passing the `_validator` parameter instead of `msg.sender` into the `InvalidValidator` error.

UPDATE

Fixed in pull request #18 (commit hash `010fedb55dbdf27c8f029993d3efdc1cacfd5e37`), as recommended.

[N04] INCORRECTLY INDEXED EVENT PARAMETERS

In EVM-compatible blockchains, events are used to log notable actions that have occurred during a transaction. Event parameters can be indexed to become topics in log entries, which allows applications to efficiently query for specific events. Indexing is suggested for events that are emitted with repeated values.

The following event parameters may benefit from adjustment:

- The `milkAdaRemoved` parameter in the `RewardsRemovedFromPool` event is a `uint256` and is unlikely to be a helpful event topic.
- The `pillager` parameter in the `SetPillager` event is not indexed, which is inconsistent with the `SetStakedMilkAda` event.

RECOMMENDATION

Consider reviewing event parameters, and use the `indexed` keyword where appropriate.

UPDATE

Fixed in pull request #20 (commit hash `8273bbf680f3b5e2d13c1dd39e01764371574bbf`), as recommended.

[N05] USE OF FLOATING COMPILER VERSION PRAGMA

All contracts in this repository float their Solidity compiler versions (e.g. `pragma solidity ^0.8.9`).

Locking the compiler version prevents accidentally deploying the contracts with a different version than what was used for testing. The current pragma prevents contracts from being deployed with an outdated compiler version, but still allows contracts to be deployed with newer compiler versions that may have higher risks of undiscovered bugs.

It is best practice to deploy contracts with the same compiler version that is used during testing and development.

RECOMMENDATION

Consider locking the compiler pragma to the specific version of the Solidity compiler used during testing and development.

UPDATE

Fixed in pull request #22 (commit hash `d336dd1f5db7fd37618328ae4b17c4fc717f3f7a`), as recommended.

[N06] UNSPECIFIED COMPILER VERSION IN FOUNDRY CONFIGURATION

The Milkomeda team has stated that they intend to use Solidity `0.8.9` for deployment and testing, however no `solc` version is specified in the project's `foundry.toml`. By default Foundry will try auto-detect the compiler version, which may not match the intended version.

RECOMMENDATION

Consider setting the `solc` configuration option in `foundry.toml`.

UPDATE

Fixed in pull request #21 (commit hash `25049c386f2760d8ed032b5f1ff7bf5b039b0147`), as recommended.

[N07] NON-STANDARD NAMING SCHEME FOR PUBLIC FUNCTIONS

The Solidity documentation recommends mixedCase function names for public and external functions. Most of the public functions in the `liquid-staking` project follow this convention, however the following functions do not:

- From `Bridge.sol`:
  - `_assertValidator` (public)
  - `_executeBridgeUpdate` (public)
- From `Pillage.sol`:
  - `_executeStakingSCUpdate` (public)
  - `_executeUpgradeTo` (public)
- From `StakedMilkAda.sol`:
  - `_executePuase` (public)
  - `_executeStakingSCUpdate` (public)
  - `_executeUnPuase` (public)
- From `StakingSmartContract.sol`:
  - `_executePillagerUpdate` (public)
  - `_executePuase` (public)
  - `_executeStakedMilkAdaUpdate` (public)
  - `_executeUnPuase` (public)

RECOMMENDATION

Consider changing the above functions names to mixedCase or changing the visibility of the functions to `internal` or `private`.

UPDATE

Fixed in pull request #23 (commit hash `bf4a1aaafe3e23ddc4a3762e780a1d57dc2e588e`), as recommended.

[N08] HARDHAT PROJECT AND UNIT TESTS ARE NON-FUNCTIONAL

The `liquid-staking` project uses Foundry as a development toolchain but still contains Hardhat specific files. These files appear to be artifacts incorrectly left behind from the migration to Foundry. Among these files include Hardhat tests which are out of date with the current codebase and will not run. The following files are Hardhat specific:

- `hardhat.config.js`
- `README.md`
- `testing.js`
- `utils.js`
- `staking.js`
- `deploy.js`

RECOMMENDATION

Consider porting remaining tests to Foundry and removing Hardhat project files from the repository.

UPDATE

Fixed in pull request #32 (commit hash `2c5e04f6f1f6db4f4d4f2162699e3ce470def97f`), as recommended.

Changes unrelated to Hardhat and unit tests were introduced in pull request #32. These changes were reviewed during the follow-on engagement.

[N09] LACK OF NATSPEC DOCUMENTATION

The following functions within the codebase either lack docstrings or have incomplete NatSpec documentation, such as omitting the `@title`, `@param`, or `@return` comments:

In `Bridge.sol`:

- `_addTransaction`
- `_assertValidator`
- `_confirmTransaction`
- `_executeBridgeUpdate`
- `_upgrade`
- `bridgeValidators`
- `generateId`
- `generateUpgradeId`
- `isConfirmed`
- `isValidatorOnBridge`
- `setBridge`
- `transactionExists`

In `Pillage.sol`:

- `_executeStakingSCUpdate`
- `_executeUpgradeTo`
- `initialize`
- `unstakeAdmin`

In `StakedMilkAda.sol`:

- `_executePuase`
- `_executeStakingSCUpdate`
- `_executeUnPuase`
- `pause`
- `setStakingSCAddress`
- `unpause`

In `StakingSmartContract.sol`:

- `_executePillagerUpdate`
- `_executePuase`
- `_executeStakedMilkAdaUpdate`
- `_executeUnPuase`
- `_unstake`
- `_validateIfSCHasWithdrawRewards`
- `_withdrawRewards`
- `initialize`
- `getSharesByPooledMilkAda`

- `pause`
- `setPillager`
- `stake`
- `unpause`
- `unstakeAdmin`
- `withdrawRewards`
- `withdrawRewardsSC`

Lack of complete documentation makes understanding and interacting with the codebase more difficult.

RECOMMENDATION

The Solidity documentation recommends NatSpec for all public interfaces (everything in the ABI). Consider implementing NatSpec-compliant docstrings for all public and external functions.

A good example is the OpenZeppelin ERC20 contract. It follows the NatSpec guidelines, and provides contract documentation that gives additional information about context and usage.

UPDATE

Fixed in pull request #33 (commit hash `992e2a6103d93dfece283c90d857ccfa369781aa`), as recommended.

[N10] USE OF LONG NUMERIC LITERALS

The `StakingSmartContract` uses a constant `factorPercentage` to scale various rewards calculations. It is currently assigned as a decimal number:

```
uint256 public constant factorPercentage = 100000000000000000000000; // Adds 23-decimal precision
```

Expressions can be stored in `constant` variables as long as the value is constant at compile time (e.g. `10 * (10 ** 2)`). Solidity also supports scientific notation for integer literals (e.g. `1e2`).

Either one of these representations makes the code cleaner, and is less error-prone than manually typing digits.

RECOMMENDATION

Consider using scientific notation or an expression for `factorPercentage`.

UPDATE

Fixed in pull request #25 (commit hash `e8ae3a48268b78f5c7f1d1fbc3b7ec2e74811e39`), as recommended.

[N11] INCONSISTENT USE OF NAMED RETURN VALUES

The following functions use named return values inconsistently with other functions in the codebase:

- The `allowance` function in `StakedMilkAda.sol` declares a named return value that is unused.
- The `getSharesByPooledMilkAda` function in `StakingSmartContract.sol` uses the named return value in some, but not all cases.
- The `unstake` function in `StakingSmartContract.sol` declares a named return value that is unused.
- The `_unstake` function in `StakingSmartContract.sol` declares a named return value that is not set and used in an event emission. This value will always be zero.

Incorrect use of named return variables complicates code and reduces readability.

RECOMMENDATION

Consider using named return values in every codepath when they are defined, and removing any cases of unused named return values.

UPDATE

Fixed in pull request #26 (commit hash `19829216bf4dd7537246da5123a3739d1ec8ab94`), as recommended.

[N12] TYPOGRAPHICAL ERRORS

The following lines contain typographical errors:

- `addresss` should be `address`:
  - `StakedMilkAda.sol`, line 123
  - `StakedMilkAda.sol`, line 144
  - `StakedMilkAda.sol`, line 145
  - `StakedMilkAda.sol`, line 167
  - `StakedMilkAda.sol`, line 182
- `antoher` should be `another`:
  - `StakedMilkAda.sol`, line 121
  - `StakedMilkAda.sol`, line 142
- `balace` should be `balance`:
  - `StakingSmartContract.sol`, line 255
- `decreses` should be `decrease`:
  - `StakedMilkAda.sol`, line 207
- `descreased` should be `deducted`:
  - `StakedMilkAda.sol`, line 208
- `percetage` should be `percentage`:
  - `StakingSmartContract.sol`, line 382
  - `StakingSmartContract.sol`, line 386
  - `StakingSmartContract.sol`, line 388
  - `StakingSmartContract.sol`, line 389
- `Puase` should be `Pause`:
  - `StakedMilkAda.sol`, line 372
  - `StakedMilkAda.sol`, line 379
  - `StakedMilkAda.sol`, line 390
  - `StakedMilkAda.sol`, line 397
  - `StakingSmartContract.sol`, line 426
  - `StakingSmartContract.sol`, line 433
  - `StakingSmartContract.sol`, line 444
  - `StakingSmartContract.sol`, line 451
- `Retrives` should be `Retrieves`:
  - `StakedMilkAda.sol`, line 180
- `shars` should be `shares`:
  - `StakedMilkAda.sol`, line 77
- `stakinSC` should be `stakingSC`:
  - `Bridge.sol`, line 165
- `thet` should be `that`:
  - `Bridge.sol`, line 169
- `trasaction` should be `transaction`:
  - `Bridge.sol`, line 170

- `whenver` should be `whenever`:
  - `StakingSmartContract.sol`, line 201

RECOMMENDATION

Consider making the suggested changes to fix the typographical errors.

UPDATE

Fixed in pull request #27 (commit hash `31caa306b46718ef1bd1e089f6f45e1e1ffcbe4c`) as recommended.

[N13] UNUSED ERROR AND ENUM VALUES

The following error and enum values are defined but unused:

- The enum value `Status.ALREADY_EXECUTED` in Base.sol
- The `CallToBridgeFailure` error in Base.sol

Unused code reduces the overall clarity of a codebase.

RECOMMENDATION

Consider removing error and enum values that are not used.

UPDATE

Fixed in pull request #28 (commit hash `57ac0ef235ae325f4606e7208343fad02946b472`), as recommended.

[N14] LACK OF INITIALIZATION ADDRESS CHECKS

The `initialize` functions in the `StakingSmartContract`, `Pillage`, and `StakedMilkAda` contracts set address storage variables without input validation:

- The `Pillage` contract's `initialize` function sets `bridge` and `stakingSC` without verifying that the `_bridge` and `_stakingSC` parameters are smart contract addresses.
- The `StakedMilkAda` contract's `initialize` function sets `bridge` without verifying that the `_bridge` parameter is a smart contract address.
- The `StakingSmartContract` contract's `initialize` function sets `bridge` and `stakedMilkAda` without verifying that the `_bridge` and `_stakedMilkAda` parameters are smart contract addresses.

In the `Pillage` and `StakedMilkAda` contracts, the `stakingSC` address can be updated with validator majority by calling the `setStakingSC` function. However, the `bridge` and `stakedMilkAda` addresses cannot be updated with a function call in any of the contracts. Initializing the `bridge`, or `stakedMilkAda` address to an address that is not a contract will render the system inoperable until an upgrade is applied that can modify the storage variables.

OpenZeppelin's `Address` contract contains the `isContract` function which can be used to verify an address is a smart contract. A check using the `isContract` function will eliminate the possibility of initializing any of these parameters to the zero address or an EOA.

RECOMMENDATION

Consider adding checks to the three `initialize` functions to ensure that the `_bridge`, `_stakingSC`, and `_stakedMilkAda` parameters are smart contract addresses.

UPDATE

Not fixed. At the time of review, no parameter checks have been added to the initializer functions. The `_stakingSC` parameter is provided by precomputing the address of the `StakingSmartContract`. Although Milkomeda points out in their statement that `_stakingSC` is precomputed, a check can still be added to ensure the parameter is not zero. The other two parameters, `_bridge` and `stakedMilkAda`, can use the `isContract` function to ensure those parameters correspond to existing smart contracts.

Milkomeda's statement for this issue:

> *We need to pre-compute the address of the staking contract before it's deployed, so we can create dead shares and have the StakedMilkAda contract mint the (dead) shares first by validating that the caller is the Liquid Staking address. Due to this, adding initialization address checks (specifically) to the Liquid Staking contract wouldn't achieve the goal of ensuring that the address passed is indeed a smart contract address.*

APPENDIX

APPENDIX A: SEVERITY DEFINITIONS

Severity	Definition
Critical	This issue is straightforward to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
High	This issue is difficult to exploit and is likely to lead to catastrophic impact for client's reputation and can lead to financial loss for client or users.
Medium	This issue is important to fix and puts a subset of users' data at risk and is possible to lead to moderate financial impact.
Low	This issue is not exploitable in a recurring basis and cannot have a significant impact on execution.
Note	This issue does not pose an immediate risk but is relevant to security best practices.

APPENDIX B: FILES IN SCOPE

```
contracts
├── base
│   ├── Base.sol
│   └── Bridge.sol
├── mock
│   ├── MockStakerNoSupport.sol
│   └── MockStakerSupport.sol
├── pillage
│   └── Pillage.sol
├── stakedMilkAda
│   ├── IStakedMilkAda.sol
│   └── StakedMilkAda.sol
└── stakingSC
    ├── IStakingSmartContract.sol
    └── StakingSmartContract.sol
```

APPENDIX C: FOLLOW-ON REVIEW

The Milkomeda team engaged AE to review additional changes to the Liquid Staking codebase after the conclusion of the fix review. AE auditors reviewed additional changes to the Liquid Staking codebase, and relayed feedback to the Milkomeda team.

AE reviewed the following during the follow-on engagement:

- PR #19
- PR #32
- The `AccreditToPool` function at commit `06210a0591087dc64ea3a0d9c09a5c12773d327b`
- The `calculateDeposit` function at commit `06210a0591087dc64ea3a0d9c09a5c12773d327b`

PR #19 addressed the inconsistencies in rewards calculations identified in finding L01 and included other refactoring in `StakingSmartContract.sol` (which was renamed to `LiquidStaking.sol`). This review focused on the changes unrelated to L02. Some code-cleanliness suggestions were relayed to the Milkomeda team, and no findings were identified that impact the security of the protocol.

PR #32 removed Hardhat artifacts identified in finding N08 and included other refactoring. This review focused on the changes unrelated to the Hardhat project artifacts. No findings were identified that impact the security of the protocol.

During the follow-on review, the Milkomeda team identified a bug in the `calculateDeposit` function. The bugfix was introduced in PR #44. AE reviewed the fix, and recommended adding tests to exercise this calculation to ensure correct behavior in the future. AE did not identify other security vulnerabilities in `AccreditToPool` or `calculateDeposit`.

After receiving feedback, the Milkomeda team implemented suggestions in PR #41. AE reviewed this pull request, and relayed a final round of feedback to the Milkomeda team.

With the exception of the bug identified by the Milkomeda team, no findings were identified that impact the security of the protocol.