CODESPECT

lstOlas contracts

SECURITY ASSESSMENT REPORT

October 21, 2025

Prepared for:

stOLAS

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

- **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.
- **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.
- **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

- **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.
- **Medium** - Likely only under certain conditions or moderately incentivized.
- **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

- **Critical** - Must be addressed immediately if already deployed.
- **High** - Must be resolved before deployment (or urgently if already deployed).
- **Medium** - It is recommended to fix.
- **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of lstOLAS. lstOLAS is a liquid staking protocol for OLAS tokens that enables cross-chain staking operations, allowing users to earn rewards while maintaining liquidity through the stOLAS token.

The scope of this audit covers the lstOLAS contracts on Ethereum mainnet, as well as contracts that will be deployed on the Base and Gnosis chains.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of smart contracts, execution testing.

CODESPECT found twelve points of attention, four classified as `High`, three classified as `Medium`, four classified as `Low`, and one classified as `Informational`. All of the issues are summarised in Table 2.

Organisation of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the functionality of the code in scope.
- **Section 7** presents the issues.
- **Section 8** presents the additional notes of the auditors.
- **Section 9** discusses the documentation provided by the client for this audit.
- **Section 10** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
High	0	4	0
Medium	0	3	0
Low	0	4	0
Informational	0	1	0
Total	**0**	**12**	**0**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	lstOLAS
Type of Project	Liquid Staking Protocol
Duration of Engagement	15 Days
Duration of Fix Review Phase	2 Days
Draft Report	October 13, 2025
Final Report	October 21, 2025
Repository	olas-lst
Commit (Audit)	e9d19b7fb08288f829cba8243d91e80ebde745a6
Commit (Final)	3f80329d4d6e06cd9a49f1f14c48b0f39d3c5533
Documentation Assessment	Medium
Test Suite Assessment	Medium
Auditors	Kalogerone, 0xluk3, 0xsynthrax

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	**LoC**
1	Proxy.sol	37
2	Beacon.sol	36
3	BeaconProxy.sol	34
4	Implementation.sol	31
5	l1/Depository.sol	451
6	l1/stOLAS.sol	190
7	l1/Treasury.sol	126
8	l1/Lock.sol	99
9	l1/DefaultDepositProcessorL1.sol	88
10	l1/Distributor.sol	69
11	l1/UnstakeRelayer.sol	39
12	l1/bridging/LzOracle.sol	216
13	l1/bridging/BaseDepositProcessorL1.sol	54
14	l1/bridging/GnosisDepositProcessorL1.sol	25
15	l2/StakingTokenLocked.sol	420
16	l2/StakingManager.sol	282
17	l2/DefaultStakingProcessorL2.sol	225
18	l2/ActivityModule.sol	152
19	l2/Collector.sol	145
20	l2/StakingHelper.sol	40
21	l2/ModuleActivityChecker.sol	15
22	l2/bridging/BaseStakingProcessorL2.sol	44
23	l2/bridging/GnosisStakingProcessorL2.sol	42
	Total	**2860**

5.2 Findings Overview

	Finding	Severity	Update
1	The `LzOracle` contract doesn't assign any value to the `depository` address	High	Fixed
2	Tokens cannot be retrieved back after lock period is over	High	Fixed
3	Users are able to request to unstake before their stake gets completed in L2	High	Fixed
4	`Collector.sol` can't set protocol fees	High	Fixed
5	Attacker can create and activate malicious staking proxies	Medium	Fixed
6	L2 bridge doesn't correctly queue the hash of failed `stake` and `unstake` low level calls	Medium	Fixed
7	The L2 Processor doesn't correctly redirect the tokens of failed deposits	Medium	Fixed
8	Bad actor can retire proxies that have fund shortage	Low	Fixed
9	L1 bridging refunds are sent to `tx.origin` and will not work for non-EOA callers	Low	Fixed
10	Wrong gas calculations in `lzCreateAndActivateStakingModel(...)` call	Low	Fixed
11	`increaseLock` can be used to lock contract balance by anyone	Low	Fixed
12	ERC4626 compliance issues	Info	Fixed

6 System Overview

lstOLAS is a cross-chain liquid staking protocol for OLAS tokens that enables users to stake on L1 (Ethereum) and have their stakes deployed to staking contracts across multiple L2 chains. The protocol introduces stOLAS, a yield-bearing wrapper token that represents staked OLAS.

L1 Components:

- **stOLAS** – An ERC4626 vault token that represents staked OLAS. Users deposit OLAS and receive shares that appreciate in value as staking rewards accumulate in the L2 Staking Proxies.
- **Depository** – This contract manages "staking models" (capacity-limited staking proxies on different L2 chains), allocates incoming OLAS deposits across active models, and coordinates cross-chain stake/unstake operations through bridge-specific deposit processors.
- **Treasury** – Handles user withdrawal requests by issuing time-locked ERC-6909 withdrawal tokens and managing the redemption process, automatically triggering L2 unstakes through the Depository when vault reserves are insufficient to cover withdrawals.
- **Deposit Processor** – Bridge-specific adapters (Gnosis, Base) that handle the technical details of sending OLAS tokens and encoded messages across different bridge implementations to their corresponding L2 chains.
- **Distributor** – Receives bridged OLAS rewards from L2 chains and splits them according to a configurable lock factor. A portion goes to the Lock contract for veOLAS governance power, while the remainder tops up the stOLAS vault balance to benefit all stakers.
- **Lock** – Manages the protocol's veOLAS position by locking a portion of rewards for up to 4 years, enabling the protocol to participate in OLAS governance through proposals and voting.
- **UnstakeRelayer** – Receives OLAS from retired staking models on L2s and channels it into stOLAS as reserve balance, keeping funds available for future staking.
- **LzOracle** – A LayerZero-based oracle contract that uses cross-chain read capabilities to verify L2 staking contract parameters remotely and create/close staking models in the Depository without manual intervention, ensuring L1 model configurations stay synchronized with actual L2 staking contract states.

L2 Components:

- **StakingManager** – Automatically creates, deploys, and manages autonomous services that get staked into Staking Proxy contracts whenever sufficient OLAS arrives from L1.
- **ActivityModule** – A module contract deployed per service that tracks activity through nonces, enables the multisig to claim staking rewards periodically, and drains those rewards back to the Collector for bridging to L1.
- **StakingTokenLocked** – The actual staking contract instances that lock service NFTs and distribute OLAS rewards based on whether services meet minimum activity thresholds during checkpoint periods.
- **Collector** – Accumulates staking rewards or unstake requests, applies a configurable protocol fee to reward operations, and coordinates bridging accumulated OLAS back to the L1.
- **Staking Processor** – Bridge-specific receivers (Gnosis, Base) that process incoming stake/unstake messages from L1, execute the appropriate StakingManager operations, and handle bridging OLAS tokens back to L1 when needed.

6.1 Deposit Flow Process

Users deposit OLAS tokens via the `Depository` contract:

```
function deposit(
    uint256 stakeAmount,
    uint256[] memory chainIds,
    address[] memory stakingProxies,
    bytes[] memory bridgePayloads,
    uint256[] memory values
) external payable returns (uint256 stAmount, uint256[] memory amounts);
```

Users can also deposit by only entering the desired `stakeAmount`, as the rest of the function parameters are optional. The `Depository` calls the `stOLAS`'s deposit function and mints the calculated shares.

Then, for each L2 staking model (if selected), the `Depository` transfers the OLAS tokens to the `DepositProcessor` and calls its `sendMessage(...)` function:

```
function sendMessage(address _target, bytes calldata _message, uint32 _minGasLimit) external payable;
```

The `DepositProcessor` bridges OLAS and sends the stake message to L2. The L2 `StakingProcessor` receives the tokens and the message and calls `StakingManager`'s `stake(...)` function:

```
function stake(address stakingProxy, uint256 amount, bytes32 operation) external virtual;
```

The `StakingManager` creates/deploys services as needed and stakes it in the desired `StakingTokenLocked` contract.

6.2 Withdrawal Flow Process

Users withdraw OLAS tokens via the `Treasury` contract and get minted an ERC-6909 withdrawal request token, which requires users to wait `withdrawDelay` period before finalizing their withdrawal:

```
function requestToWithdraw(
    uint256 stAmount,
    uint256[] memory chainIds,
    address[] memory stakingProxies,
    bytes[] memory bridgePayloads,
    uint256[] memory values
) external payable returns (uint256 requestId, uint256 olasAmount);
```

The `Treasury` redeems stOLAS by calling the `stOLAS`'s redeem function. If the vault + reserve balance is insufficient, it calls the `Depository`'s unstake function. The `Depository` calculates the unstake amounts and bridges the unstake messages to L2 using the `DepositProcessor` again.

The L2 `StakingProcessor` receives the unstake message and calls `StakingManager`'s `unstake(...)` function:

```
function unstake(address stakingProxy, uint256 amount, bytes32 operation) external;
```

The `StakingManager` unstakes the services and sends OLAS to the `Collector`, which bridges tokens back to the L1.

7 Issues

7.1 [High] The `LzOracle` contract doesn't assign any value to the `depository` address

File(s): `LzOracle.sol`

Description: The `depository` variable is declared as:

```
// Depository address
address public immutable depository
```

However, it is never assigned in the constructor (nor at declaration). Since it is not set, any external calls to `IDepository(depository)` would silently target a nonexistent address.

Impact: Any calls to the `depository` address will fail.

Recommendation(s): Assign a value at constructor or hardcode correct address in the contract.

Status: Fixed

Update from lstOLAS: PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.2 [High] Tokens cannot be retrieved back after lock period is over

File(s): `Lock.sol`, reference: `veOLAS.sol`

Description: The `Lock` contract creates `veOLAS` locks but lacks any mechanism to withdraw the underlying OLAS tokens after the lock period expires. When `setGovernorAndCreateFirstLock(...)` is called, the `Lock` contract locks `OLAS` tokens in the `veOLAS` contract, for instance:

```
function setGovernorAndCreateFirstLock(address _olasGovernor) external {
    // ...

    // Approve OLAS for veOLAS
    IToken(olas).approve(ve, olasAmount);
    // Create lock
    IVEOLAS(ve).createLock(olasAmount, MAX_LOCK_TIME);

    // ...
}
```

The `veOLAS` contract's `withdraw()` function allows withdrawal only after the lock expires, and it checks `mapLockedBalances[msg.sender]` - meaning the `Lock` contract address would need to call it since it is recorder as the balance owner:

```
// Line 402
    function createLock(uint256 amount, uint256 unlockTime) external {
        _createLockFor(msg.sender, amount, unlockTime);
    }
// ...

// Line 510
function withdraw() external {
    LockedBalance memory lockedBalance = mapLockedBalances[msg.sender];
    // Only msg.sender (the Lock contract) can withdraw their locked balance
}
// ...
```

However, the Lock contract provides no function to call `veOLAS`'s `withdraw()` after the lock period ends.

Impact: `OLAS` tokens locked in `veOLAS` through the `Lock` contract will be permanently unrecoverable after the lock time expires. The tokens cannot be withdrawn by anyone, not even the contract owner.

Recommendation(s): Add a withdraw function to the `Lock` contract that can be called by the owner after the lock expires.

Status: Fixed

Update from lstOLAS: We will make lock time increase variable tunable in Lock contract. Meaning when it's set to 0 by the DAO, the lock will not be extended. This will ultimately release the lock. **withdraw()** function will also be required. PR with fix: https://github.com/kupermind/olas-lst/pull/41

7.3 [High] Users are able to request to unstake before their stake gets completed in L2

File(s): `Treasury.sol`

Description: Users are able to instantly request to unstake their `stOLAS`, even before their stake is completed in L2. This can create issues in the case that their stake fails to be completed in L2. There are 2 scenarios where this could happen:

a. The L2 Processor contract is `paused`;
b. The low-level call from the L2 Processor to the staking proxy fails;

In both cases, the stake request gets queued. When finally redeemed, the tokens get sent back to the L1 `UnstakeRelayer` contract. However, users are able to `requestToWithdraw(...)` on the `Treasury` contract before this happens. Take the following scenario on an empty vault:

a. User stakes 100 `OLAS` and gets minted 100 `stOLAS`;
b. The `DefaultStakingProcessorL2` contract is paused, this will queue his `STAKE` request and will return the amount to L1 when redeemed;
c. User unstakes his 100 `stOLAS`, making the vault's `stakedBalance` to be `0`;
d. On L2, the unstake process will complete and won't really change anything;
e. The 100 `OLAS` from his initial failed stake reach the L1 `UnstakeRelayer` contract. When the `relay()` function gets called, the vault's `totalReserves` becomes 100;
f. User is unable to complete his withdrawal, while he has burned his `stOLAS` shares;
g. Next user who deposits will enjoy those 100 `OLAS` in the `totalReserves`. Additionally, the initial user will be able to complete his withdrawal later at the cost of someone else's unstake, when tokens will get sent to the `Treasury`;

If the `stakedBalance` from other users can cover the user's withdrawal, the impact is not that high. Follow a similar scenario on a non-empty vault:

a. User stakes 100 `OLAS` and gets minted 100 `stOLAS`;
b. The `DefaultStakingProcessorL2` contract is paused, this will queue his `STAKE` request and will return the amount to L1 when redeemed;
c. User unstakes his 100 `stOLAS`;
d. On L2, the unstake process will eventually be executed and send 100 `OLAS` to the `Treasury` contract;
e. The 100 `OLAS` from his initial failed stake reach the L1 `UnstakeRelayer` contract. When the `relay()` function gets called, the vault's `totalReserves` increases by 100 and the `stakedBalance` decreases by 100;
f. The user is able to complete his withdrawal, but there are `100` OLAS in the `totalReserves` left to be staked again;

Impact: If the vault is empty, not all unstake operations will be able to get completed ever. If the vault is not empty, this issue essentially unnecessarily unstakes twice the request amount.

Recommendation(s): Consider implementing a timer where users are not able to request to unstake after they have staked.

Status: Fixed

Update from lstOLAS: When STAKE fails for any other reason other than insufficient OLAS balance:

- funds will be routed to the Collector address with the "target" address recorded;
- when UNSTAKE* takes place and fails, the redeem() function will try to unstake again, and if failed - claim the operation to be processed by Collector giving a "target" address. If that goes through, the queue hash is erased and operation considered successful;
- there cannot be more UNSTAKE* operations executed than the full deposit for "target" as recorded on L1, thus it can be recoverable;
- On the StakingManager side there's probably a need to check on Collector's balance for a specified target as "balance" amount might not be enough, since funds partially reside on the Collector side;

PR with fix: https://github.com/kupermind/olas-lst/pull/43

7.4 [High] `Collector.sol` can't set protocol fees

File(s): `Collector.sol`

Description: The `changeProtocolFactor(...)` function is supposed to set the protocol fee which is collected from the rewards. However, the function call is going to revert because of wrong zero value check:

```
function changeProtocolFactor(uint256 newProtocolFactor) external {
    // Check for ownership
    if (msg.sender != owner) {
        revert OwnerOnly(msg.sender, owner);
    }

    // Check for zero value
    // @audit attemts to set protocolFactor to any value will revert
    if (protocolFactor == 0) {
        revert ZeroValue();
    }

    protocolFactor = newProtocolFactor;
    // ...
}
```

Instead of checking `newProtocolFactor` for zero value, function performs the check on old `protocolFactor` value, which is 0 by default and is not set during construction. This checking will make any attempts to set protocol fee revert.

Impact: Protocol won't be able to set `protocolFactor` and collect fees from rewards

Recommendation(s): Correctly identify if the `newProtocolFactor` is 0, instead of the old `protocolFactor`. Consider also adding some sanity bounds.

Status: Fixed

Update from lstOLAS: Good catch, PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.5 [Medium] Attacker can create and activate malicious staking proxies

File(s): `StakingTokenLocked.sol`, `LzOracle.sol`

Description: Anybody can create a staking proxy using `StakingFactory::createStakingInstance(...)` function by providing arbitrary `bytes initPayload` parameter, which is going to be used to initialize the proxy. While the function checks legitimacy of some parameters inside the mentioned struct, it fails to verify the following parameters: `stakingManager`, `livenessPeriod` and `activityChecker`. These malicious parameters can be used to harm the protocol and it's users. Possible attack path:

a. Attacker creates a staking proxy and sets the `livenessPeriod` to a very high value;
b. He activates the proxy using the `LzOracle::lzCreateAndActivateStakingModel(...)` function;
c. Attacker monitors the `stOLAS` contract for idle reserves, and stakes them in his staking proxy;
d. Later, when the `StakingTokenLocked::checkpoint()`function is called for reward allocation, the proxy wouldn't allocate any rewards for the stake because of this check in `_calculateStakingRewards()` function:;

```
if (size > 0 && block.timestamp - tsCheckpointLast >= livenessPeriod && lastAvailableRewards > 0)
```

This check will not pass because the `livenessPeriod` value is so big that the `block.timestamp - tsCheckpointLast >= livenessPeriod` statement is never going to be true. The prevention of reward allocation could also be achieved by setting a malicious `activityChecker`, which will always return `false` on `ActivityChecker.isRatioPass()` calls in `_checkRatioPass()` function:

```
// Get the ratio pass activity check
activityData = abi.encodeCall(IActivityChecker.isRatioPass, (currentNonces, lastNonces, ts));
(success, returnData) = activityChecker.staticcall(activityData);

// The return data must match the size of bool
if (success && returnData.length == 32) {
    ratioPass = abi.decode(returnData, (bool));
}
```

Setting a malicious `stakingManager` would just create a staking proxy which is going to revert `STAKE` operations because of this check in `StakingTokenLocked::stake()`:

```
function stake(uint256 serviceId) external {
    // Check for stakingManager address
    if (msg.sender != stakingManager) {
        revert UnauthorizedAccount(msg.sender);
    }
    // ...
}
```

Impact: Attacker staked the user's funds in the proxy that never going to allocate any rewards for that deposit

Recommendation(s): Perform a check on `stakingManager`, `livenessPeriod` and `activityChecker` values

Status: Fixed

Update from lstOLAS: PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.6 [Medium] L2 bridge doesn't correctly queue the hash of failed `stake` and `unstake` low level calls

File(s): `DefaultStakingProcessorL2.sol`

Description: The `_processData(...)` function processes the messages that are received from L1. This function should never revert, so it handles and queues any reverts to be executed at a later time. However, this is not true if the low level `stake` and `unstake` calls fail:

```
function _processData(bytes memory data) internal {
    // ...

    // Status to be emitted for failing scenarios, since reverts cannot be engaged in this function call
    RequestStatus status;
    if (operation == STAKE) {
        if (paused == 1) {
            // Get current OLAS balance
            uint256 olasBalance = IToken(olas).balanceOf(address(this));

            // Check the OLAS balance and the contract being unpaused
            if (olasBalance >= amount) {
                // Approve OLAS for stakingManager
                IToken(olas).approve(stakingManager, amount);

                // This is a low level call since it must never revert
                bytes memory stakeData = abi.encodeCall(IStakingManager.stake, (target, amount, operation));
                (success,) = stakingManager.call(stakeData);
            } else {
                // Insufficient OLAS balance
                status = RequestStatus.INSUFFICIENT_OLAS_BALANCE;
            }
        } else {
            // Contract is paused
            status = RequestStatus.CONTRACT_PAUSED;
        }
    } else if (operation == UNSTAKE || operation == UNSTAKE_RETIRED) {
        // Note that if UNSTAKE* is requested, it must be finalized in any case since changes are recorded on L1
        // This is a low level call since it must never revert
        bytes memory unstakeData = abi.encodeCall(IStakingManager.unstake, (target, amount, operation));
        (success,) = stakingManager.call(unstakeData);
    // ...

    // Check for operation success and queue, if required
    if (success) {
        emit RequestExecuted(batchHash, target, amount, operation);
    } else {
        // Hash of batchHash + target + amount + operation + current target dispenser address
        bytes32 queueHash = getQueuedHash(batchHash, target, amount, operation);
        // Queue the hash for further redeem
        queuedHashes[queueHash] = status;

        emit RequestQueued(batchHash, target, amount, operation, status);
    }

    _locked = 1;
}
```

First, the `status` variable is initialized with the default value of `NON_EXISTENT`. Then, depending on the `operation`, the contract executes a low level call to the `stakingManager` contract to `stake` or `unstake`. This call's `success` is stored to be checked later. However, if `success` is `false`, the `status` variable is never set appropriately. This results in queueing a hash with the `status` of `NON_EXISTENT`. Later, this queued hash can't be executed:

```
function redeem(bytes32 batchHash, address target, uint256 amount, bytes32 operation) external {
    // ...

    // Check if the target and amount are queued
    if (requestStatus == RequestStatus.NON_EXISTENT) {
        revert RequestNotQueued(target, amount, batchHash, operation);
    }
// ...

}
```

Impact: Failed low level calls don't get queued correctly and can't be re-executed.

Recommendation(s): Set the `status` appropriately when the calls fail:

```
if (olasBalance >= amount) {
    // Approve OLAS for stakingManager
    IToken(olas).approve(stakingManager, amount);

    // This is a low level call since it must never revert
    bytes memory stakeData = abi.encodeCall(IStakingManager.stake, (target, amount, operation));
    (success,) = stakingManager.call(stakeData);
+   if (!success) {
+       status = RequestStatus.EXTERNAL_CALL_FAILED;
+   }

} else if (operation == UNSTAKE || operation == UNSTAKE_RETIRED) {
    // Note that if UNSTAKE* is requested, it must be finalized in any case since changes are recorded on L1
    // This is a low level call since it must never revert
    bytes memory unstakeData = abi.encodeCall(IStakingManager.unstake, (target, amount, operation));
    (success,) = stakingManager.call(unstakeData);
+   if (!success) {
+       status = RequestStatus.EXTERNAL_CALL_FAILED;
+   }
```

Status: Fixed

Update from lstOLAS: The status is EXTERNAL_CALL_FAILED by default

Update from CODESPECT: but the `RequestStatus` enum is defined like this:

```
enum RequestStatus {
    NON_EXISTENT,
    EXTERNAL_CALL_FAILED,
    INSUFFICIENT_OLAS_BALANCE,
    UNSUPPORTED_OPERATION_TYPE,
    CONTRACT_PAUSED
}
```

If `status` is not changed, isn't its default value `NON_EXISTENT`?

Update from lstOLAS: Good point! Setting initial status to EXTERNAL_CALL_FAILED would do the trick: `RequestStatus status = RequestStatus.EXTERNAL_CALL_FAILED`; This ensures that if `!success`, the status is already set. PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.7 [Medium] The L2 Processor doesn't correctly redirect the tokens of failed deposits

File(s): `DefaultStakingProcessorL2.sol`

Description: When the process would fail during the bridging, the L2 Processor queues it for later execution. More specifically, if the L2 Processor is `paused` or the low-level stake call to the `Staking Proxy` fails the funds should go to the `Collector` contract under the operation `UNSTAKE_RETIRED`, so they return back to the L1 `UnstakeRelayer`. The `Collector` is not supposed to handle `STAKE` operations. However this is not the case, as the L2 Processor calls the `Collector`'s `topUpBalance(...)` function with `STAKE` as the operation:

```
function redeem(bytes32 batchHash, address target, uint256 amount, bytes32 operation) external {
    // ...

    if (operation == STAKE) {
        // Get the current contract OLAS balance
        uint256 olasBalance = IToken(olas).balanceOf(address(this));
        if (olasBalance >= amount) {
            // Approve OLAS for stakingManager
            IToken(olas).approve(stakingManager, amount);
        } else {
            // OLAS balance is not enough for redeem
            revert InsufficientBalance(olasBalance, amount);
        }

        // If request was queued due to insufficient balance - continue with the stake
        if (requestStatus == RequestStatus.INSUFFICIENT_OLAS_BALANCE) {
            IStakingManager(stakingManager).stake(target, amount, operation);
        } else {
            // Approve OLAS for collector to initiate L1 transfer for corresponding operation by agents
            IToken(olas).approve(collector, amount);

            // Request top-up by Collector for a specific unstake operation
            ICollector(collector).topUpBalance(amount, operation);
        }
```

Impact: Tokens are not correctly directed back to the L1 `UnstakeRelayer` contract.

Recommendation(s): Consider making the following changes:

```
    function redeem(bytes32 batchHash, address target, uint256 amount, bytes32 operation) external {
            // ...

            // If request was queued due to insufficient balance - continue with the stake
            if (requestStatus == RequestStatus.INSUFFICIENT_OLAS_BALANCE) {
                IStakingManager(stakingManager).stake(target, amount, operation);
            } else {
                // Approve OLAS for collector to initiate L1 transfer for corresponding operation by agents
                IToken(olas).approve(collector, amount);

                // Request top-up by Collector for a specific unstake operation
-               ICollector(collector).topUpBalance(amount, operation);
+               ICollector(collector).topUpBalance(amount, UNSTAKE_RETIRED);
            }
```

Status: Fixed

Update from lstOLAS: This is related to issue 07: **[High] Users are able to request to unstake before their stake gets completed in L2**, and must be actioned together accordingly. PR with fix: https://github.com/kupermind/olas-lst/pull/43

7.8 [Low] Bad actor can retire proxies that have fund shortage

File(s): LzOracle.sol, StakingTokenLocked.sol

Description: The LzOracle's contract lzCloseStakingModel(...) function has no access control. The only condition to retire the proxy is the availableRewards == 0 case:

```
function _lzReceive(
    Origin calldata origin,
    bytes32 guid,
    bytes calldata message,
    address, /* executor */
    bytes calldata /* extraData */
) internal override {
    // ...

    } else if (accountChainIdMsgType.msgType == READ_TYPE_CLOSE) {
        // Decode obtained data
        uint256 availableRewards = abi.decode(message, (uint256));

        // Check for correctness of parameters
        if (availableRewards > 0) {
            revert();
        }

        IDepository(depository).LzCloseStakingModel(accountChainIdMsgType.chainId, accountChainIdMsgType.account);

        emit LzCloseStakingModelProcessed(guid, accountChainIdMsgType.chainId, accountChainIdMsgType.account);
    } else {
        // This must never happen
        revert();
    }
}
```

The state mentioned above can occur after checkpoint() function call, if the funds were not enough to cover the allocated rewards. If properly working proxy that was not supposed to be retired gets fund shortage, bad actor can force it to retire. For example:

a. Attacker monitors the availableRewards and rewards that are supposed to be allocated to the services;
b. If at any moment shortage appears, attacker calls the checkpoint() function and sets availableRewards to 0;
c. Calls lzCloseStakingModel() and sets the stakingModel.status to StakingModelStatus.Retired;
d. Calls unstakeRetired(...) and initiates a withdraw;
e. Calls closeRetiredStakingModels(...) and closes the staking model;

Impact: Attacker can close any staking proxy that unintentionally falls into funds shortage.

Recommendation(s): Add access control to lzCloseStakingModel(...) function.

Status: Fixed

Update from lstOLAS: PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.9 [Low] L1 bridging refunds are sent to tx.origin and will not work for non-EOA callers

File(s): LzOracle.sol, DefaultDepositProcessorL1.sol

Description: In the LzOracle contract, the functions lzCreateAndActivateStakingModel(...) and lzCloseStakingModel(...) send LayerZero refunds to payable(tx.origin). The same happens in the DefaultDepositProcessorL1 contract. The sendMessage(...) function sends any leftovers to the tx.origin address. tx.origin is the top-level EOA for the transaction, not necessarily the payer/caller of these functions. When these functions are invoked by smart contract wallets or account abstraction relayers, refunds may be misdirected to a unexpected EOA instead of the actual caller.

Impact: The refunds may be missed in rare cases.

Recommendation(s): Consider returning the funds to initial msg.sender instead.

Status: Fixed

Update from lstOLAS: Since these function are now ownerOnly, refund to msg.sender is safe, and no need to provide an additional refunder address. PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.10 [Low] Wrong gas calculations in `lzCreateAndActivateStakingModel(...)` call

File(s): `LzOracle.sol`

Description: The function `lzCreateAndActivateStakingModel(...)` tries to calculate the gas needed for the oracle call using the `options` variable, provided by the user, but later, during the call, it combines it with an enforced option:

```
MessagingFee memory fee = _quote(READ_CHANNEL, payload, options, false);
require(msg.value >= fee.nativeFee);

MessagingReceipt memory receipt = _lzSend(
  READ_CHANNEL,
  payload,
  combineOptions(READ_CHANNEL, READ_TYPE_CREATE, options), // @audit enforced options can increase
                                                           // the gas cost
  MessagingFee(msg.value, 0),
  payable(tx.origin)
);
```

Impact: The combined options can increase the gas cost and the gas forwarded from the contract might be insufficient, resulting in transaction revert.

Recommendation(s): Use the combined options too during the gas calculation.

Status: Fixed

Update from lstOLAS: PR with fix: https://github.com/kupermind/olas-lst/pull/42

7.11 [Low] `increaseLock` can be used to lock contract balance by anyone

File(s): `Lock.sol`

Description: The `increaseLock(...)` function is the only privileged function in the contract that lacks access control, creating a discrepancy with other, access-controlled ones such as `changeGovernor(...)`, `setGovernorAndCreateFirstLock(...)`, `propose(...)`, `castVote(...)`.

```
function increaseLock(uint256 olasAmount) external returns (bool unlockTimeIncreased) {
    // @audit No access control - anyone can call this

    // Approve OLAS for veOLAS
    IToken(olas).approve(ve, olasAmount);

    // Increase lock amount
    IVEOLAS(ve).increaseAmount(olasAmount);

    // Increase unlock time to a maximum, if possible
    bytes memory increaseUnlockTimeData = abi.encodeCall(IVEOLAS.increaseUnlockTime, (MAX_LOCK_TIME));
    // Note: both success and failure are acceptable
    (unlockTimeIncreased,) = ve.call(increaseUnlockTimeData);
}
```

This function uses OLAS tokens from the `Lock` contract's own balance, which is concerning as it allows an attacker to lock up the contract's funds. An attacker can extend the unlock time to `MAX_LOCK_TIME` (4 years), effectively griefing the owner by preventing timely fund withdrawals.

Impact: Attackers can infinitely keep increasing the unlock time of the locked `OLAS` tokens.

Recommendation(s): Add owner access control to align with the contract's security pattern for other privileged functions.

Status: Fixed

Update from lstOLAS: Same solution as specified in **[High] Tokens cannot be retrieved back after lock period is over** PR with fix: https://github.com/kupermind/olas-lst/pull/41

7.12 [Info] ERC4626 compliance issues

File(s): `stOLAS.sol`

Description: The `stOLAS` contract is supposed to be fully compliant with the ERC4626 vault standard for maximum DeFi composability. However, there is an instance where this is not true. The `maxMint(...)` and `maxWithdraw(...)` functions should always return `0`, because mints and withdrawals are disabled.

Status: Fixed

Update from lstOLAS: Maybe revert?

https://github.com/transmissions11/solmate/blob/main/src/tokens/ERC4626.solL164

https://github.com/transmissions11/solmate/blob/main/src/tokens/ERC4626.solL168

Update from CODESPECT: According to the EIP, these functions "MUST NOT" revert. For full compilance they should return 0. Source: https://eips.ethereum.org/EIPS/eip-4626

Update from lstOLAS: PR with fix: https://github.com/kupermind/olas-lst/pull/42

8 Additional Notes

This section provides supplementary auditor observations regarding the code. These points were not identified as individual issues but serve as informative recommendations to enhance the overall quality and maintainability of the codebase.

- The `Depository.sol` contract implements 2 unused mappings, `mapAccountDeposits` which only gets written but never read and `mapAccountWithdraws` which is completely unused.
- In the `stOLAS.sol` contract, the `deposit(...)` function unnecessarily calls the internal `calculateCurrentBalances()` function, while it can just use the `totalReserves` variable.
- In the `Treasury.sol` contract, the `requestToWithdraw(...)` function unnecessarily transfers the `stOLAS` shares from the `msg.sender` and then calls the `redeem(...)` function. It can directly call the `redeem(...)` function with `msg.sender` as the `tokenOwner`.

Update from lstOLAS: We are going to leave 2 unused mappings, as those will be used from offchain or by other contracts.

The rest of the changes are in this PR: PR-40

9 Evaluation of Provided Documentation

The **lstOLAS** documentation was provided in the form of a code walkthrough where all functionalities and flows were explained. Additionally, the code contained NatSpec comments laying out the intended functionality of each function and the reasoning behind most of the logic. The documentation could be further improved by adding:

- **Official Documentation:** The protocol currently lacks comprehensive official documentation. This absence of formal documentation creates information asymmetry, making it difficult for users and auditors to fully understand the intended behavior of critical functions and validate their correct implementation.
- **Diagrams:** The documentation could be further improved by providing diagrams with flows, especially where cross-chain transfers were involved in a function, like `deposit` and `unstake` in the `Depository` contract.

The documentation provided was overall good and sufficient for the scope of the audit. Nevertheless, the lstOLAS team remained consistently available and responsive, promptly addressing all questions and concerns raised by **CODESPECT** during the audit process.

10 Test Suite Evaluation

10.1 Compilation Output

```
% make build

forge build
Warning: Found unknown config section in foundry.toml: [lint]
This notation for profiles has been deprecated and may result in the profile not being registered in future versions.
Please use [profile.lint] instead or run `forge config --fix`.
[] Compiling...
[] Compiling 141 files with Solc 0.8.30
[] Solc 0.8.30 finished in 5.20s
Compiler run successful with warnings:
...
```

10.2 Tests Output

```
% make tests-hardhat

yarn test:hardhat
yarn run v1.22.22
$ mv contracts/l1/bridging/LzOracle.sol contracts/l1/bridging/LzOracle._sol && hardhat test && mv
↪  contracts/l1/bridging/LzOracle._sol contracts/l1/bridging/LzOracle.sol


  Liquid Staking
    Staking
L1
User approves OLAS for depository: 3000000000000000000000
User deposits OLAS for stOLAS
User stOLAS balance now: 3000000000000000000000
OLAS total assets on stOLAS: 3000000000000000000000
Protocol current veOLAS balance: 1000000000000000000

L2
OLAS rewards available on L2 staking contract: 2000000000000000000000000
Reward before checkpoint 0
Wait for liveness period to pass
Calling checkpoint by agent or manually
Reward after checkpoint 43200500000000000000
Calling claim by agent or manually
Collector balance: 43200500000000000000
Calling relay rewards tokens to L1 by agent or manually

L1
Calling distribute obtained L2 to L1 OLAS to veOLAS and stOLAS by agent or manually
OLAS total assets on stOLAS now: 3004276849500000000000
User approves stOLAS for treasury: 3000000000000000000000
User requests withdraw of small amount of stOLAS: 43200500000000000000
Withdraw requestId: 32477701478199890496035225600
User is minted ERC6909 tokens corresponding to number of OLAS: 43262087345608250000
User to finalize withdraw request after withdraw cool down period
Approve 6909 requestId tokens for treasury
Finalize withdraw
User got OLAS: 43262087345608250000

L1 - L2 - L1
User requests withdraw of all remaining stOLAS: 2999567995000000000000
Withdraw requestId: 32477701533540122717163880449
User is minted ERC6909 tokens corresponding to number of OLAS: 30038442286265439175000
OLAS is not enough on L1, sending request to L2 to unstake and transfer back to L1
Calling relay usntaked tokens to L1 by agent or manually

L1
User to finalize withdraw request after withdraw cool down period
Finalize withdraw
User got OLAS: 30038442286265439175000
```

```
Final user stOLAS remainder: 0
Final OLAS total assets on stOLAS: 0
       E2E liquid staking simple (146ms)

...

 FULL UNSTAKE ITERATION: 39
User requests partial withdraw of stOLAS: 468212790104990781714 76
Failed to parse requestId from log[5], falling back to stack without a transfer event:
Withdraw requestId: 325399337276016062924293407 51
User is minted ERC6909 tokens corresponding to number of OLAS: 508070624531243933626 49
Calling relay unstaked tokens to L1 by agent or manually
User to finalize withdraw request after withdraw cool down period
Approve 6909 requestId tokens for treasury: 508070624531243933626 49
Finalize withdraw
User got OLAS: 508070624531243933626 49
stakedBalance: BigNumber { value: "16" }
vaultBalance: BigNumber { value: "0" }
reserveBalance: BigNumber { value: "0" }
Final user stOLAS remainder: 14
Final OLAS total assets on stOLAS: 16
       Multiple stakes-unstakes (54113ms)
L1
User deposits OLAS for stOLAS
User stOLAS balance now: 5399999999999999999999820
OLAS total assets on stOLAS: 5399999999999999999999820

L2
Collector balance: 2000000000000000000000000
Calling relay unstake retired tokens to L1 by agent or manually

L1
Calling relay unstake retired tokens to stOLAS by agent or manually
stakedBalance before: 2000000000000000000000000
vaultBalance before: 0
reserveBalance before: 3399999999999999999999820
stakedBalance after: 0
vaultBalance after: 0
reserveBalance after: 5399999999999999999999820
Final user stOLAS remainder: 5399999999999999999999820
Final OLAS total assets on stOLAS: 5399999999999999999999820
       Retire models (1790ms)
L1
User deposits OLAS amount: 26666666666666666666666
User deposits OLAS for stOLAS
User stOLAS preview balance: 26666666666666666666666
User stOLAS balance now: 26666666666666666666666
OLAS total assets on stOLAS: 26666666666666666666666
stakedBalance after 1st deposit: 20000000000000000000000
vaultBalance after 1st deposit: 0
reserveBalance after 1st deposit: 6666666666666666666666
User OLAS preview balance: 26666666666666666666666
stakedBalance after last deposit: 200000000000000000000000
vaultBalance after last deposit: 0
reserveBalance after last deposit: 69333333333333333333266
OLAS total assets on stOLAS now: 269333333333333333333266
User OLAS preview balance: 269333333333333333333266
       Check OLAS vs stOLAS amounts (2768ms)
L1
User deposits OLAS amount: 26666666666666666666666
User deposits OLAS for stOLAS
User stOLAS balance now: 26666666666666666666666
stakedBalance after 1st deposit: 0
vaultBalance after 1st deposit: 0
reserveBalance after 1st deposit: 26666666666666666666666
       Deposit without staking


  8 passing (1m)

Done in 74.35s.
```

```
% make tests

forge test -vvv
Warning: Found unknown config section in foundry.toml: [lint]
This notation for profiles has been deprecated and may result in the profile not being registered in future versions.
Please use [profile.lint] instead or run `forge config --fix`.
[] Compiling...
[] Compiling 130 files with Solc 0.8.30
[] Solc 0.8.30 finished in 4.85s
Compiler run successful with warnings:

...

Ran 1 test for test/tests.t.sol:LzOracleFeeTest
[FAIL: EvmError: Revert] setUp() (gas: 0)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 246.70µs (0.00ns CPU time)

Ran 1 test for test/testing.t.sol:LzOracleFeeTest
[PASS] testFees() (gas: 438278)
Logs:
  User options length: 19
  Enforced options length: 19
  Combined options length: 36

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 293.90µs (109.10µs CPU time)

Warning: the following cheatcode(s) are deprecated and will be removed in future versions:
  revertTo(uint256): replaced by `revertToState`
  snapshot(): replaced by `snapshotState`
Ran 6 tests for test/LiquidStaking.t.sol:LiquidStakingTest
[PASS] testE2ELiquidStakingSimple() (gas: 137602)
Logs:
  === Starting setUp ===
  Deployer address: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  Deploying MockERC20...
  MockERC20 deployed at: 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
  Deploying MockStOLAS...
  MockStOLAS deployed at: 0x2e234DAe75C793f67A35089C9d99245E1C58470b
  Deploying MockLock...
  MockLock deployed at: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
  Deploying MockDistributor...
  MockDistributor deployed at: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9
  Deploying MockUnstakeRelayer...
  MockUnstakeRelayer deployed at: 0xc7183455a4C133Ae270771860664b6B7ec320bB1
  Deploying MockDepository...
  MockDepository deployed at: 0xa0Cb889707d426A7A386870A03bc70d1b0697598
  Deploying MockTreasury...
  MockTreasury deployed at: 0x1d1499e622D69689cdf9004d05Ec547d650Ff211
  Deploying MockCollector...
  MockCollector deployed at: 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c
  Deploying MockBeacon...
  MockBeacon deployed at: 0x03A6a84cD762D9707A21605b548aaaB891562aAb
  Deploying MockStakingManager...
  MockStakingManager deployed at: 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF
  === Starting contract initialization ===
  Initializing Lock...
  Lock initialized
  Initializing Distributor...
  Distributor initialized
  Initializing UnstakeRelayer...
  UnstakeRelayer initialized
  Initializing Depository...
  Depository initialized
  Initializing Treasury...
  Treasury initialized
  Initializing Collector...
  Collector initialized
  Initializing StakingManager...
  StakingManager initialized
  Setting up stOLAS managers...
  stOLAS managers set
  Setting up depository treasury...
  Depository treasury set
  Funding Lock with initial OLAS...
  Lock funded with 1 ether
  Setting governor and creating first lock...
  Governor set and first lock created
  Funding StakingManager...
  StakingManager funded with 1 ether
  === Contract initialization completed ===
  === setUp completed successfully ===
  === E2E Liquid Staking Simple Test ===
  L1
  User deposits OLAS for stOLAS
  L2
  Test completed successfully

[PASS] testMaxNumberStakes() (gas: 321481)
Logs:
  === Starting setUp ===
  Deployer address: 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  Deploying MockERC20...
  MockERC20 deployed at: 0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f
  Deploying MockStOLAS...
  MockStOLAS deployed at: 0x2e234DAe75C793f67A35089C9d99245E1C58470b
  Deploying MockLock...
  MockLock deployed at: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
  Deploying MockDistributor...
  MockDistributor deployed at: 0x5991A2dF15A8F6A256D3Ec51E99254Cd3fb576A9
  Deploying MockUnstakeRelayer...
  MockUnstakeRelayer deployed at: 0xc7183455a4C133Ae270771860664b6B7ec320bB1
  Deploying MockDepository...
  MockDepository deployed at: 0xa0Cb889707d426A7A386870A03bc70d1b0697598
  Deploying MockTreasury...
  MockTreasury deployed at: 0x1d1499e622D69689cdf9004d05Ec547d650Ff211
  Deploying MockCollector...
  MockCollector deployed at: 0xA4AD4f68d0b91CFD19687c881e50f3A00242828c
  Deploying MockBeacon...
  MockBeacon deployed at: 0x03A6a84cD762D9707A21605b548aaaB891562aAb
  Deploying MockStakingManager...
  MockStakingManager deployed at: 0xD6BbDE9174b1CdAa358d2Cf4D57D1a9F7178FBfF
  === Starting contract initialization ===
  Initializing Lock...
  Lock initialized
  Initializing Distributor...
  Distributor initialized
  Initializing UnstakeRelayer...
  UnstakeRelayer initialized

...

  Setting governor and creating first lock...
  Governor set and first lock created
  Funding StakingManager...
  StakingManager funded with 1 ether
  === Contract initialization completed ===
  === setUp completed successfully ===
  === Two Services Deposit, One Unstake, More Deposit, Full Unstake Test ===
  L1
  Test completed successfully

Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.25ms (3.17ms CPU time)

Ran 3 test suites in 7.20ms (2.79ms CPU time): 7 tests passed, 1 failed, 0 skipped (8 total tests)

Failing tests:
Encountered 1 failing test in test/tests.t.sol:LzOracleFeeTest
[FAIL: EvmError: Revert] setUp() (gas: 0)

Encountered a total of 1 failing tests, 7 tests succeeded
make: *** [Makefile:21: tests] Error 1
```

10.3 Notes on the Test Suite

The provided test suite includes comprehensive testing of the main flows, like depositing and staking, focusing primarily on correct logic behavior of common usage scenarios. However, some functionalities remain completely untested which led to some simple High severity findings, like an immutable variable not being set. Expanding the test suite to include every contract functionality, would significantly improve the robustness of the codebase.

Furthermore CODESPECT advises against using mock contracts for testing integrations, if possible, as those by definition exhibit limited behavior compared to the original systems they are mimicking.