Audited by

CredShields

Smart Contract Audit

October 16th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Lara Protocol between 1st October 2024, and 8th October 2024. A retest was performed on 14th October, 2024.

Author
Shashank (Co-founder, CredShields) shashank@CredShields.com

Reviewers
Aditya Dixit (Research Team Lead), Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor)

Prepared for
Lara Protocol

Table of Contents

1. Executive Summary --------------------

Lara Protocol engaged CredShields to perform a smart contract audit from 1st October 2024 to 8th October 2024. During this timeframe, 13 vulnerabilities were identified. **A retest was performed on 14th October 2024, and all the bugs have been addressed.**

During the audit, 0 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Lara Protocol" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Liquid Staking Contract	0	0	5	4	0	4	**13**
	0	**0**	**5**	**4**	**0**	**4**	**13**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in the Liquid Staking Contract's scope during the testing window while abiding by the policies set forth by Lara Protocol's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Lara Protocol's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Lara Protocol can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Lara Protocol can future-proof its security posture and protect its assets.

2. The Methodology -------------------

Lara Protocol engaged CredShields to perform a Liquid Staking Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from 1st October, 2024, to 8th October, 2024, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

IN SCOPE ASSETS
https://github.com/Lara-staking/liquid-staking/tree/ad76def710199391f684d939570c66d5360ec6ec

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

Lara Protocol is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

<table>
<tr><th colspan="5">Overall Risk Severity</th></tr>
<tr><td rowspan="4"><b>Impact</b></td><td>HIGH</td><td>● Medium</td><td>● High</td><td>● Critical</td></tr>
<tr><td>MEDIUM</td><td>● Low</td><td>● Medium</td><td>● High</td></tr>
<tr><td>LOW</td><td>● None</td><td>● Low</td><td>● Medium</td></tr>
<tr><td></td><td>LOW</td><td>MEDIUM</td><td>HIGH</td></tr>
<tr><td colspan="5"><b>Likelihood</b></td></tr>
</table>

Overall, the categories can be defined as described below -

1. Informational

We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. Low

Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. Medium

Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. High

High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. Critical

Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. Gas

To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- Shashank, Co-founder CredShields shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

3. Findings Summary -------------------

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, 13 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Presale SwapUpperLimit can be bypassed	Medium	Business Logic
Underflow while minting stTARA Token	Medium	Arithmetic Underflow
Inconsistent Staking in stake() Function	Medium	Incorrect Calculation
Incorrect calculation of commissionPart leads to Precision Loss	Medium	Incorrect Calculation
Storage Layout Conflict in ApyOracle contract	Medium	Storage Layout Conflict
Missing Zero Address Validations	Low	Missing Input Validation
Missing Events in Important Functions	Low	Missing Best Practices
Floating and Outdated Pragma	Low	Floating Pragma

Use Ownable2Step	Low	Missing Best Practices
Cheaper Inequalities in if()	Gas	Gas Optimization
Cheaper Inequalities in require()	Gas	Gas Optimization
Gas Optimization for State Variables	Gas	Gas Optimization
Cheaper Conditional Operators	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Vulnerable	Bug ID #2
SWC-102	Outdated Compiler Version	Vulnerable	Bug ID #8
SWC-103	Floating Pragma	Vulnerable	Bug ID #8
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0
SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use
SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.

SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found
SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.
SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found

SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status ------------------

Lara Protocol is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on 14th October, 2024, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Presale SwapUpperLimit can be bypassed	Medium	**Won't Fix** [14th October, 2024]
Underflow while minting stTARA Token	Medium	**Fixed** [14th October, 2024]
Inconsistent Staking in stake() Function	Medium	**Fixed** [14th October, 2024]
Incorrect calculation of commissionPart leads to Precision Loss	Medium	**Fixed** [14th October, 2024]
Storage Layout Conflict in ApyOracle contract	Medium	**Fixed** [14th October, 2024]
Missing Zero Address Validations	Low	**Fixed** [14th October, 2024]
Missing Events in Important Functions	Low	**Fixed** [14th October, 2024]
Floating and Outdated Pragma	Low	**Won't Fix** [14th October, 2024]
Use Ownable2Step	Low	**Fixed** [14th October, 2024]
Cheaper Inequalities in if()	Gas	**Won't Fix** [14th October, 2024]
Cheaper Inequalities in require()	Gas	**Won't Fix** [14th October, 2024]

Gas Optimization for State Variables	Gas	**Fixed** [14th October, 2024]
Cheaper Conditional Operators	Gas	**Fixed** [14th October, 2024]

Table: Summary of findings and status of remediation

5. Bug Reports ------------------------

Bug ID #1 [Won't Fix]

Presale SwapUpperLimit can be bypassed

Vulnerability Type

Business Logic

Severity

Medium

Description:

In the presale contract's swap() function, users can swap their native currency for presale at a fixed rate. The function enforces both a minimum swap amount and an upper limit (denoted by minSwapAmount and swapUpperLimit) to ensure that no user can buy an excessive amount of tokens in this sale. Additionally, the function enforces a restriction that once a user reaches the upper limit, they cannot buy more tokens until after 900 blocks.

However, this restriction is vulnerable to a token transfer bypass. Specifically, a user can circumvent the upper limit check by:

1. Swapping tokens for an amount just below the upper limit.
2. Transferring the tokens from their account to another address.
3. Swapping tokens again in a new transaction since their balance has decreased below the upper limit.
4. Repeating this process to accumulate more tokens than allowed by the swapUpperLimit.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L87-L103

Impacts

Users can bypass the token limit, enabling them to purchase far more tokens than intended. This can skew the token distribution and give an unfair advantage to certain participants.

Remediation

To prevent this vulnerability, the contract must implement a stricter tracking mechanism. Instead of checking the user's balance at the time of the swap, the contract should track the total amount swapped by each user during the presale and enforce the upper limit based on this cumulative amount.

```
mapping(address => uint256) public swappedAmount;

function swap() external payable nonReentrant {

———————————Code————————————————————-

            // Track the total swapped amount to enforce the upper limit
            uint256 newSwapAmount = swappedAmount[ msg.sender ] + msg.value;
            require(newSwapAmount <= swapUpperLimit, "Presale: you can swap max 1000000 TARA");

———————————Code ——————————————-

            // Update the swapped amount and last swap block
            swappedAmount[ msg.sender ] = newSwapAmount;
          if (swappedAmount[ msg.sender ]  >= presaleRate * swapUpperLimit) {
            lastSwapBlock[ msg.sender ] = block.number;
            }
}
```

Retest

Client’s comment: We acknowledge the finding but due to release schedule and monetary constraints weren’t able to audit the codebase quicker. The contract in scope has been already deployed to the Mainnet and presale has been concluded, therefore the contract can never be put in a vulnerable state again.

Bug ID #2 [Fixed]

Underflow while minting stTARA Token

Vulnerability Type

Arithmetic Underflow

Severity

Medium

Description:

In the stake() function, the contract delegates funds to validators using the contract's balance (address(this).balance). However, if the contract's balance is larger than the amount provided by the user, the resulting remainingAmount could be greater than what the user originally staked.

The issue occurs when the contract tries to mint stTARA tokens by subtracting remainingAmount from the user's amount. If the remainingAmount is larger than the amount, this subtraction will cause an underflow.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L211

Impacts

Users might face failed transactions when trying to stake their funds, which means they won't receive their stTARA tokens.

Remediation

To fix this, the contract should check that the remainingAmount is not greater than the user's staked amount before performing the subtraction.

Retest

This vulnerability has been fixed by using the above remediation.

Bug ID #3 [Fixed]

Inconsistent Staking in stake() Function

Vulnerability Type

Incorrect Calculation

Severity

Medium

Description:

The `stake()` function allows users to stake native tokens by passing the desired amount and the corresponding Ether (`msg.value`). The function checks that `msg.value` is at least equal to `amount` with the condition `msg.value < amount`. However, if a user inadvertently sends more Ether than specified in the amount, this extra Ether will be delegated to validators by the `_delegateToValidators()` function.

The issue lies in how the contract handles the extra Ether sent by the user. When the `_delegateToValidators()` function is called, it delegates the contract's entire balance (including the excess Ether), resulting in the user delegating more native tokens than expected. This can lead to users receiving fewer staking tokens (`stTARA`) than they should have, as only the specified amount is minted. At the same time, the excess Ether is still delegated to validators.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L196-L223

Impacts

Users may lose access to extra tokens they sent inadvertently, as the extra tokens will be delegated but not reflected in their staked token balance. This creates a discrepancy between the amount of native tokens staked and the staked tokens (`stTARA`) received.

Remediation

It is recommended to modify the `msg.value` checks to ensure the user sends exactly the correct amount of native tokens for staking:

```
if (msg.value != amount) revert StakeValueIncorrect(msg.value, amount);
```

Retest

This vulnerability has been fixed by implementing the correct validation.

Bug ID #4 [Fixed]

Incorrect calculation of commissionPart leads to Precision Loss

Vulnerability Type

Incorrect Calculation

Severity

Medium

Description:

In the distributeRewardsForSnapshot() function, the contract calculates the generalPart based on the staked balance of the user (delegatorBalance) and the total rewards available for the snapshot (distributableRewards). After that, it calculates the commissionPart using a simple division and multiplication. This calculation is vulnerable to a loss of precision due to integer division. Specifically, if the generalPart is very small (e.g., less than 100 wei), the division generalPart / 100 will result in a zero value, effectively setting commissionPart to zero, regardless of the commissionDiscounts[staker]. To avoid this, mathematical operations should respect the principle of multiplying first before dividing to prevent precision loss. By performing division before multiplication, the contract introduces the possibility of rounding errors, leading to incorrect reward distribution, especially for users with small staking balances.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L306
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L268

Impacts

Users with small staking balances will receive significantly fewer rewards than they are entitled to due to the loss of precision when calculating commissionPart.

Remediation

To prevent the loss of precision, the contract should perform multiplication before division in the commissionPart calculation. This can be achieved by swapping the order of operations:

Retest

This vulnerability has been fixed by implementing the above remediation.

Bug ID #5 [Fixed]

Storage Layout Conflict in ApyOracle contract

Vulnerability Type

Storage Layout Conflict

Severity

Medium

Description

The ApyOracle contract is upgradeable but lacks a storage gap. Without a storage gap, future versions of the contract may introduce new variables that overwrite storage slots in unexpected ways, potentially corrupting the contract's state. This issue arises from Solidity's use of sequential storage slots, where new variables can be written into slots reserved for other purposes in the original version of the contract.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol

Impacts

Without a storage gap, new storage variables introduced in the contract can overwrite the beginning of the storage layout, causing unexpected behavior and potentially severe vulnerabilities.

Remediation

Introduce a storage gap in the contract to reserve space for future storage variables without affecting the contract's storage layout. Or you can use namespace variables.

Retest

This vulnerability has been fixed by introducing a storage gap.

Bug ID #6 [Fixed]

Missing Zero Address Validations

Vulnerability Type

Missing Input Validation

Severity

Low

Description:

The contracts were found to be setting new addresses without proper validations for zero addresses.

Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burned forever.

Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost forever.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/StakedNativeAsset.sol#L76-L78
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L112-L126
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L174-L177
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L47-L53
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L77-L79

Impacts

If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.

Remediation

Add a zero address validation to all the functions where addresses are being set.

Retest

This vulnerability has been fixed by implementing zero address validation.

Bug ID #7 [Fixed]

Missing Events in Important Functions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code
The following functions were affected -

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L109-L111
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L286-L288
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L293-L295
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L318-L320
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L77-L79
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L206-L208
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L152-L154
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L159-L161

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L182-L184
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L189-L191
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L386-L407
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L629-L638
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L55-L61
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L66-L82

Impacts

Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation

Consider emitting events for important functions to keep track of them.

Retest

This issue has been fixed as recommended in remediation.

Bug ID #8 [Won't Fix]

Floating and Outdated Pragma

Vulnerability Type
Floating Pragma (SWC-103)

Severity
Low

Description
Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contract allowed floating or unlocked pragma to be used, i.e., 0.8.20, ^0.8.20. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L5
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L3
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L3
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ReentrancyGuard.sol#L4
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L3
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/StakedNativeAsset.sol#L3
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L5

Impacts
If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.

Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is low.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.26 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

Client’s comments: Taraxa's EVM doesn't support PUSH0 ATM, therefore we can use up to 0.8.25.
We ran several automated test scenarios on 0.8.20 prior to the audit, having the system up and running for months, and hence we decided to stay at 0.8.20.

Bug ID #9 [Fixed]

Use Ownable2Step

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ApyOracle.sol#L15
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L38
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/StakedNativeAsset.sol#L11

Impacts
Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Remediation
It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

Retest

This vulnerability has been fixed by implementing Ownable2StepUpgradeable.

Bug ID #10 [Won't Fix]

Cheaper Inequalities in if()

Vulnerability Type

Gas Optimization

Severity

Gas

Description

The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L307
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L198
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L201
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L236
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L325
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L79

Impacts

Using strict inequalities inside "if" statements costs more gas.

Remediation

It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest

Client’s comments: This level of nuanced gas optimisation is not important as our target chains are DPoS ones, all having small to irrelevant gas costs and the added complexity in checks is not worth the saved gas.

Bug ID #11 [Won't Fix]

Cheaper Inequalities in require()

Vulnerability Type

Gas Optimization

Severity

Gas

Description

The contract was found to be performing comparisons using inequalities inside the require statement. When inside the require statements, non-strict inequalities (>=, <=) are usually costlier than strict equalities (>, <).

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L223
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L422
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L492
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L493
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L495
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L529
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L69
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L90
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L91
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L92
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L94

Impacts

Using non-strict inequalities inside “require” statements costs more gas.

Remediation

It is recommended to go through the code logic, and, **if possible**, modify the non-strict inequalities with the strict ones to save gas as long as the logic of the code is not affected.

Retest

Client’s comments: This level of nuanced gas optimisation is not important as our target chains are DPoS ones, all having small to irrelevant gas costs and the added complexity in checks is not worth the saved gas.

Bug ID #12 [Fixed]

Gas Optimization for State Variables

Vulnerability Type

Gas Optimization

Severity

Gas

Description

Plus equals (+=) costs more gas than the addition operator. The same thing happens with minus equals (-=). Therefore, x +=y costs more gas than x = x + y.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L354
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L393
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L394
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L513
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/Lara.sol#L517

Impacts

Writing the arithmetic operations in x = x + y format will save some gas.

Remediation

It is suggested to use the format x = x + y in all the instances mentioned above.

Retest

This issue has been fixed as recommended in remediation.

Bug ID #13 [Fixed]

Cheaper Conditional Operators

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Upon reviewing the code, it has been observed that the contract uses conditional statements involving comparisons with unsigned integer variables. Specifically, the contract employs the conditional operators x != 0 and x > 0 interchangeably. However, it's important to note that during compilation, x != 0 is generally more cost-effective than x > 0 for unsigned integers within conditional statements.

Affected Code

- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/ERC20Snapshot.sol#L222
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L67
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L79
- https://github.com/Lara-staking/liquid-staking/blob/ad76def710199391f684d939570c66d5360ec6ec/contracts/LaraToken.sol#L89

Impacts
Employing x != 0 in conditional statements can result in reduced gas consumption compared to using x > 0. This optimization contributes to cost-effectiveness in contract interactions.

Remediation
Whenever possible, use the x != 0 conditional operator instead of x > 0 for unsigned integer variables in conditional statements.

Retest
This issue has been fixed by replacing '>' with '!='.

6. The Disclosure -----------------------

The Reports provided by CredShields are not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.

YOUR SECURE FUTURE STARTS HERE

CRED SHIELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHIELDS