CredShields

Smart Contract Audit

June 20th, 2022 • CONFIDENTIAL

Description

This document details the process and result of the pStake smart contract audit performed by CredShields Technologies PTE. LTD. on behalf of pStake finance between May 24th, 2022, and June 14th, 2022.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

pStake Finance

Table of Contents

1. Executive Summary

pStake Finance engaged CredShields to perform a smart contract audit from May 24th, 2022, to June 14th, 2022. During this timeframe, Thirteen (13) vulnerabilities were identified. **A retest was performed by the CredShields team between 15th June 2022 to 20th June 2022 and all the vulnerabilities were found to be fixed.**

During the audit, zero (0) vulnerability was found that had a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "pStake Finance" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and 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
pStake AVAX Smart Contracts	0	0	0	5	5	3	**13**
	0	**0**	**0**	**5**	**5**	**3**	**13**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in pStake's scope during the testing window while abiding by the policies set forth by "pStake Finance" team.

State of Security

Maintaining a healthy security posture requires constant review and refinement of existing security processes. Running a CredShields continuous audit allows “pStake Finance” internal security team and development team to not only uncover specific vulnerabilities but gain a better understanding of the current security threat landscape.

We recommend running regular security assessments to identify any vulnerabilities introduced after pStake Finance introduces new features or refactors the code.

Reviewing the remaining resolved reports for a root cause analysis can further educate “pStake Finance” internal development and security teams and allow manual or automated procedures to be put in place to eliminate entire classes of vulnerabilities in the future. This proactive approach helps contribute to future-proofing the security posture of pStake Finance’s assets.

2. Methodology

pStake Finance engaged CredShields to perform a smart contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

CredShields team read all the provided documents and comments in the smart-contract code to understand the contract's features and functionalities. The team reviewed all the functions and prepared a mind map to review for possible security vulnerabilities in the order of the function with more critical and business-sensitive functionalities for the refactored code.

The team deployed a self-hosted version of the smart contract to verify the assumptions and validation of the vulnerabilities during the audit phase.

A testing window from May 24th, 2022, to June 14th, 2022, 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/persistenceOne/stkAVAX-contracts • **AddressStore.sol** • **FeeVault.sol** • **LiquidStaking.sol** • **Registry.sol** • **StakedAVAXToken.sol** • **IAddressStore.sol** • **IFeeVault.sol** • **IStakedAVAXToken.sol** • **Account.sol** • **BasisFee.sol** • **Config.sol** • **ExchangeRate.sol** • **FeeDistribution.sol**

Table: List of Files in Scope

2.1.2 Documentation

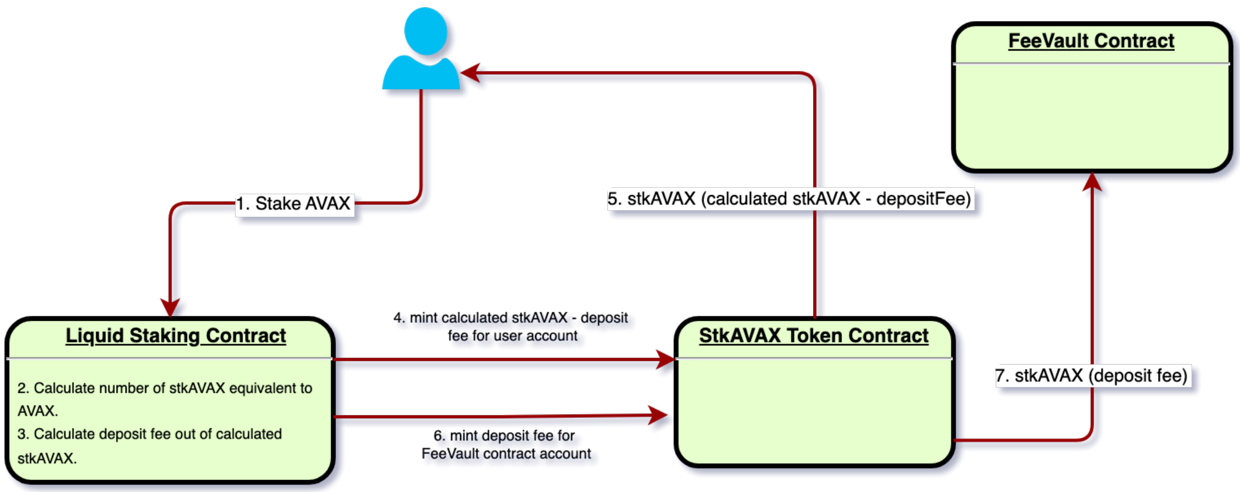
The following documentation was available to the audit team.

https://docs.google.com/document/d/1HpVAWvOfRXcCaSkZ-lTjEmzt1FHm__7O/edit?usp=sharing&ouid=117228556250886958510&rtpof=true&sd=true

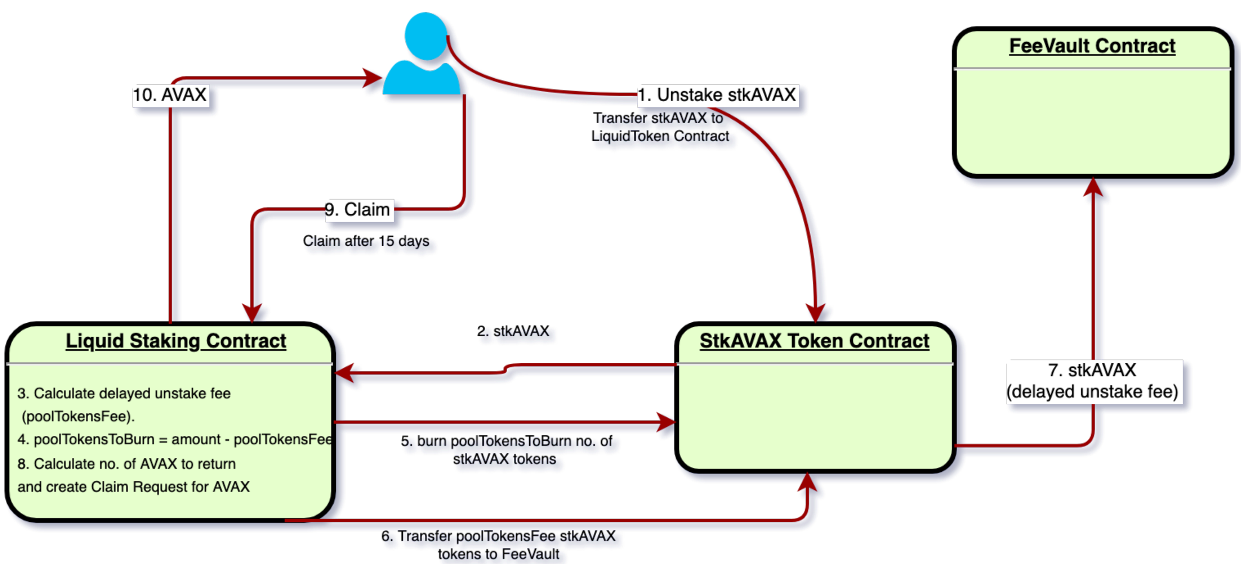
The audit team also mapped all the user flows and privileges and mapped functionalities of the smart contract. The control flow graph can be found below

Control Flow Graph

Staking/Deposit:



Unstaking/Withdrawal:



2.1.3 Audit Goals

CredShields' methodology uses individual tools and methods; however, tools are just used for aids. The majority of the audit methods involve manually reviewing the smart contract source code. The team followed the standards of the SWC registry for testing along with an extended self-developed checklist based on industry standards, but it was not limited to it. The team focused heavily on understanding the core concept behind all the functionalities along with preparing test and edge cases. Understanding the business logic and how it could have been exploited.

The audit's focus was to verify that the smart contract system is secure, resilient, and working according to its specifications. Breaking the audit activities into the following three categories:

- **Security** - Identifying security-related issues within each contract and the system of contracts.
- **Sound Architecture** - Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
- **Code Correctness and Quality** - A full review of the contract source code. The primary areas of focus include:
  - Correctness
  - Readability
  - Sections of code with high complexity
  - Improving scalability
  - Quantity and quality of test coverage

2.2 Retesting phase

pStake Finance is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

Discovering vulnerabilities is important, but estimating the associated risk to the business is just as important.

To adhere to industry guidelines, CredShields follows OWASP's Risk Rating Methodology. This is calculated using two factors - **Likelihood** and **Impact**. Each of these parameters can take three values - **Low**, **Medium**, and **High**.

These depend upon multiple factors such as Threat agents, Vulnerability factors (Ease of discovery and exploitation, etc.), and Technical and Business Impacts. The likelihood and the impact estimate are put together to calculate the overall severity of the risk.

CredShields also define an **Informational** severity level for vulnerabilities that do not align with any of the severity categories and usually have the lowest risk involved.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Overall, the categories can be defined as described below -

1. **Informational**

We believe in the importance of technical excellence and pay a great deal of attention to its details. Our coding guidelines, practices, and standards help ensure that our software is stable and reliable.

Informational vulnerabilities should not be a cause for alarm but rather a chance to improve the quality of the codebase by emphasizing readability and good practices.

They do not represent a direct risk to the Contract but rather suggest improvements and the best practices that can not be categorized under any of the other severity categories.

Code maintainers should use their own judgment as to whether to address such issues.

2. Low

Vulnerabilities in this category represent a low risk to the Smart Contract and the organization. The risk is either relatively small and could not be exploited on a recurring basis, or a risk that the client indicates is not important or significant, given the client's business circumstances.

3. Medium

Medium severity issues are those that are usually introduced due to weak or erroneous logic in the code.

These issues may lead to exfiltration or modification of some of the private information belonging to the end-user, and exploitation would be detrimental to the client’s reputation under certain unexpected circumstances or conditions. These conditions are outside the control of the adversary.

These issues should eventually be fixed under a certain timeframe and remediation cycle.

4. **High**

High severity vulnerabilities represent a greater risk to the Smart Contract and the organization. These vulnerabilities may lead to a limited loss of funds for some of the end-users.

They may or may not require external conditions to be met, or these conditions may be manipulated by the attacker, but the complexity of exploitation will be higher.

These vulnerabilities, when exploited, will impact the client’s reputation negatively.

They should be fixed immediately.

5. **Critical**

Critical issues are directly exploitable bugs or security vulnerabilities. These issues do not require any external conditions to be met.

The majority of vulnerabilities of this type involve a loss of funds and Ether from the Smart Contracts and/or from their end-users.

The issue puts the vast majority of, or large numbers of, users' sensitive information at risk of modification or compromise.

The client's reputation will suffer a severe blow, or there will be serious financial repercussions.

Considering the risk and volatility of smart contracts and how they use gas as a method of payment to deploy the contracts and interact with them, gas optimization becomes a major point of concern. To address this, CredShields also introduces another severity category called “**Gas Optimization**” or “**Gas**”. This category deals with code optimization techniques and refactoring due to which Gas can be conserved.

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 around the engagement or this document.

3. Findings

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, thirteen (13) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Floating Pragma	Low	Floating Pragma (SWC-103)
Unindexed Event Parameters	Informational	Missing best practices
Large Number Literals	Low	Missing best practices
Typo in Comment	Informational	Missing best practices
Struct Input Mismatch	Informational	Missing best practices
Functions should be declared External	Gas	Gas Optimization

Struct Packing	Gas	Gas Optimization
Scientific Notations can save Gas	Gas	Gas Optimization
Missing Input Validation in threshold	Low	Input validation
Missing Input Validation in seq	Informational	Input validation
Redundant Validation in _claim and claimAll	Informational	Missing best practices
Missing Reentrancy Protections	Low	Reentrancy
Multiple Zero Address Validations Missing	Low	Missing Input Validation

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 v0.8.7)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0.8.7 is used
SWC-103	Floating Pragma	Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used anywhere in the code
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 used but has proper access control in place
SWC-107	Reentrancy	Vulnerable	Notable functions such as deposit, claim related function were found to be missing reentrancy guard. However, they had no exploitation scenarios for now.
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	The smart contracts safely uses OpenZeppelin's

			Upgradeable contracts implementation.
SWC-113	DoS with Failed Call	Not Vulnerable	The logic is implemented in such a way that the external calls if they fail, they fail gracefully, i.e. without having a major impact on the functionality of the contract.
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 used which is better than block.number and doesn't affect for a longer duration.
SWC-117	Signature Malleability	Not Vulnerable	ecrecover function is not used anywhere in the code
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	ecrecover is not used anywhere in the code
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 unused variables were 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() has not been used anywhere
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

pStake Finance is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. The fix can be found at https://github.com/persistenceOne/stkAVAX-contracts/pull/19
Also the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Floating Pragma	Low	**Fixed [20/06/2022]**
Unindexed Event Parameters	Informational	**Fixed [20/06/2022]**
Large Number Literals	Low	**Fixed [20/06/2022]**
Typo in Comment	Informational	**Fixed [20/06/2022]**
Struct Input Mismatch	Informational	**Fixed [20/06/2022]**
Functions should be declared External	Gas	**Fixed [20/06/2022]**
Struct Packing	Gas	**Fixed [20/06/2022]**

Scientific Notations can save Gas	Gas	**Fixed [20/06/2022]**
Missing Input Validation in threshold	Low	**Fixed [20/06/2022]**
Missing Input Validation in seq	Informational	**Won’t Fix**
Redundant Validation in _claim and claimAll	Informational	**Fixed [20/06/2022]**
Missing Reentrancy Protections	Low	**Fixed [20/06/2022]**
Multiple Zero Address Validations Missing	Low	**Pending Fix**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID#1 [Fixed]

Floating 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 contracts found in the repository were allowing floating or unlocked pragma to be used, i.e., **^0.8.7**.
This allows the contracts to be compiled with all the solidity compiler versions above **0.8.7**.
The following contracts were found to be affected -

Affected Code

- ^0.8.7 (contracts/AddressStore.sol#3)
- ^0.8.7 (contracts/FeeVault.sol#3)
- ^0.8.7 (contracts/LiquidStaking.sol#3)
- ^0.8.7 (contracts/Registry.sol#3)
- ^0.8.7 (contracts/StakedAVAXToken.sol#3)
- ^0.8.7 (contracts/embedded-libs/Account.sol#3)
- ^0.8.7 (contracts/embedded-libs/BasisFee.sol#3)
- ^0.8.7 (contracts/embedded-libs/Config.sol#3)
- ^0.8.7 (contracts/embedded-libs/ExchangeRate.sol#3)

- ^0.8.7 (contracts/embedded-libs/FeeDistribution.sol#3)
- ^0.8.7 (contracts/interfaces/IAddressStore.sol#3)
- ^0.8.7 (contracts/interfaces/IFeeVault.sol#3)
- ^0.8.7 (contracts/interfaces/IStakedAVAXToken.sol#3)

Impacts

If the smart contract gets compiled and deployed with an older version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older 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 really low therefore this is only informational.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere.

Reference: https://swcregistry.io/docs/SWC-103

Retest:

Strict pragma has been implemented.

Bug ID#2 [Fixed]

Unindexed Event Parameters

Vulnerability Type

Missing best practices

Severity

Informational

Description:

Events in solidity contain two kinds of parameters - indexed and non-indexed. These indexes are also known as "topics" and are the searchable parameters used in events.
In the Ethereum system, events must be easily searched for so that applications can filter and display historical events without undue overhead.
It was noticed that the following event parameters were not indexed making the search for past events cumbersome.

Affected Code

- FeeVault - Line 29,30
  https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/FeeVault.sol#L29-L30

```
event Deposit(address from, uint256 amount);
event Withdraw(address from, address to, uint256 amount);
```

Impacts

This does not impact the security aspect of the Smart contract but affects the ease of use when searching for past events.

Remediation

It should be noted that indexed event parameters take up more gas than non-indexed ones. Keeping that in mind, the contract should add indexed keywords to the searchable parameters to make searching efficient using an event filter.

Retest:
The affected event has been indexed.

Bug ID#3 [Fixed]

Large Number Literals

Vulnerability Type
Missing best practices

Severity
Low

Description
Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations.
The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation also supported by Solidity such as **2e10**.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/embedded-libs/BasisFee.sol#L9

```
library BasisFee {
    error NumeratorMoreThanBasis();

    uint256 internal constant _BASIS = 100000000000;
```

Impacts
Having a large number literal with too many digits is bound to be used incorrectly. Literals with many digits are difficult to read and review. This may also introduce errors in the future if one of the zeroes is omitted while doing code modifications.

Remediation

Scientific notation in the form of **2e10** is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal **MeE** is equivalent to **M * 10**E**. Examples include **2e10**, **2e10**, **2e-10**, **2.5e1**, as suggested in official solidity documentation https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals

Retest:

Scientific notation is in use now.
https://github.com/persistenceOne/stkAVAX-contracts/blob/main/contracts/embedded-libs/BasisFee.sol#L9

Bug ID#4 [Fixed]

Typo in Comment

Vulnerability Type
Missing best practices

Severity
Informational

Description:
A typo was observed in the contract "**LiquidStaking.sol**" on Line 583. The spelling of "**once**" is wrong and written as "**onc**".

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L583

```
/**
    * @dev epochUpdate: This is supposed to be called onc every epoch by
the bot to trigger the necessary movement of...
```

Impacts
This does not impact the security of the smart contract but affects the usability and code review by the auditors and developers.

Remediation
Fix the spelling error - change "onc" to "once".

Retest:
The typo has been fixed.

Bug ID#5 [Fixed]

Struct Input Mismatch

Vulnerability Type
Missing best practices

Severity
Informational

Description
The embedded library **Account.sol** defines a struct called “**Data”** which defines two variables - a **pubkey** and an **addr**.
The values for the struct are being set inside the **“_set()”** function on Line 18 but the assignments are not done in the order in which the struct is defined.
This creates difficulties during code review and analysis.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/embedded-libs/Account.sol#L18-L21

```
library Account {
   using Account for Data;

   struct Data {
       string pubkey;
       address addr;
   }

   function _checkValid(Data calldata self) internal pure {
           require(bytes(self.pubkey).length != 0, "account: pubkey is required");
       require(self.addr != address(0), "account: addr is required");
   }
```

```
    function _set(Data storage self, Data calldata obj) internal {
        self.addr = obj.addr;
        self.pubkey = obj.pubkey;
    }
}
```

Impacts

This does not impact the security of the smart contract but creates confusion for auditors and developers during the audit.

Remediation

Adjust the **"_set()"** function so that it assigns the struct values in the same order in which it's defined in the struct, i.e., assign the **pubkey** first and then the **addr**.

Retest:

The struct alignment has been updated to match the function call.

Bug ID#6 [Fixed]

Functions should be declared External

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public functions that are never called by a contract should be declared external in order to conserve gas.
The following functions were declared as public but were not called anywhere in the contract, making the public visibility useless.

Affected Code

```
function selfDestruct(address _address) public
onlyRole(DEFAULT_ADMIN_ROLE) whenPaused
   {
        selfdestruct(payable(_address));
   }
```

Impacts
Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.
"**public**" functions cost more Gas than "**external**" functions.

Remediation
Use the "**external**" state visibility for functions that are never called from inside the contract.

Retest:

The function has been marked as “external”

https://github.com/persistenceOne/stkAVAX-contracts/blob/main/contracts/StakedAVAXToken.sol#L103-L105

Bug ID#7 [Fixed]

Struct Packing

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Struct variable packing can be improved to save gas costs by arranging the variables in such an order that they get tightly packed in the storage slots.
The main goal here is the reduction of gas requirements when deploying these smart contracts, and saving costs in each place will eventually add up. The consequences of the use of the Tight Variable Packing pattern have to be evaluated before implementing it blindly. The big benefit comes from the substantial amount of gas that can potentially be saved over the lifetime of a contract.

Affected Code

1. https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/embedded-libs/Account.sol#L8-L11

```
struct Data {
    string pubkey;
    address addr;
}
```

This structure costs 72281 gas while deployment whereas if the address variable was used first, it would cost 72269 gas. This saves around 12 units of gas.

Impacts

Smart Contracts are required to have effective Gas usage as they cost real money and each function should be monitored for the amount of gas it costs to make it gas efficient.

Remediation

Change the variable packing order inside the struct to use the address parameter before the string.

Retest:

The struct packing has been updated to save more gas.

Bug ID#8 [Fixed]

Scientific Notations can save Gas

Vulnerability Type

Gas Optimization

Severity

Gas

Description

It is unnecessary to perform power operations when a number can be directly represented in an exponential form as it will save gas.

The contract **LiquidStaking.sol** was defining some constant variables which were not using the scientific notations in the exponential calculations.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L52-L54
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L75
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L78

```
   uint256 private constant _P_CHAIN_AVAX_UNIT = 10**9; // P-chain uses 9
decimal for AVAX
   uint256 private constant _C_CHAIN_AVAX_UNIT = 10**18; // C-chain uses
18 decimals for AVAX
   uint256 private constant _P2C_CONVERSION_RATE = 10**9; // multiplier
when converting P-chain AVAX to C-chain AVAX
   uint256 private constant _MIN_P_CHAIN_DELEGATION_AVAX = 25 *
_P_CHAIN_AVAX_UNIT; // Minimum amount needed for delegation on P-chain
...
```

```
...
   uint256 public constant MIN_AVAX_DEPOSIT = _C_CHAIN_AVAX_UNIT / 10**6;
// 1 micro AVAX
   // @dev MIN_TOKEN_WITHDRAWAL
   // The minimum amount of tokens required to make a withdrawal from the
contract.
   uint256 public constant MIN_TOKEN_WITHDRAWAL = (10**18) / 10**6; // 1
micro stkAVAX
```

Impacts

Using exponential forms or power operations directly increases the code readability and also costs more gas during their usage in calculations.

Remediation

Scientific notation in the form of **2e10** is also supported by Solidity which can be used, where the mantissa can be fractional but the exponent has to be an integer. The literal **MeE** is equivalent to **M * 10**E**.

Examples include **2e10**, **2e10**, **2e-10**, **2.5e1**, as suggested in official solidity documentation https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals

Retest:

Now scientific notation is being used to save gas and for better visibility.

Bug ID#9 [Fixed]

Missing Input Validation in threshold

Vulnerability Type

Input validation

Severity

Low

Description

The Registry.sol contract defines a threshold variable on Line 18 which is getting initialized in the initialize() function which accepts the threshold as the argument.

This variable lacks input validation. There should be some boundaries set on the variable. Right now, the implementation trusts the administrators to pass these values correctly.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/Registry.sol#L23-L29

```
   uint256 public threshold;

   /// @custom:oz-upgrades-unsafe-allow constructor
   constructor() initializer {} // solhint-disable-line no-empty-blocks

     function initialize(NodeConfig[] memory nodes_, uint256 threshold_)
public initializer {
       threshold = threshold_;

       for (uint256 i = 0; i < nodes_.length; ++i) {
           nodes.push(nodes_[i]);
       }
   }
```

Impacts

Threshold stores the threshold values which are passed by the administrators. Errors may be introduced if the admins pass an incorrect or a beyond boundary value causing unexpected behaviors in the contract.

Remediation

Even if the administrators are trusted, there should be proper input validation on all the variables and parameters.

The threshold should have a minimum value of 1 and a maximum value of len(nodes).

Retest:

A boundary value has been added to the threshold input.

Bug ID#10 [Won't Fix]

Missing Input Validation in seq

Vulnerability Type

Input validation

Severity

Informational

Description

The LiquidStaking.sol contract defines a seq variable on Line 577 which is assigned to signLogSeq that defines the latest log sequence number that has been signed.

The variable lacks an input validation and although the function is called only by the BOT_ROLE, there should be an input validation to make sure that it does not assign an incorrect value.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L577-L580

```
        function setSignLogSeq(uint256 seq) external whenNotPaused
onlyRole(BOT_ROLE) {
        signLogSeq = seq;
        emit UpdatedSignLogSeq(signLogSeq);
    }
```

Impacts

The function setSignLogSeq will be called by the bot after it executes the log with the sequence "seq". In case there's an invalid value being passed in the variable, the contract logic will be affected and errors will be introduced.

Remediation

There should be proper input validation on all the variables and parameters. The “seq” variable should have input validations to define the maximum and minimum acceptable values.

Retest:
The team doesn’t want to have a boundary value for now. Also it doesn’t pose any risk as it has access control in place.

Bug ID#11 [Fixed]

Redundant Validation in _claim and claimAll

Vulnerability Type

Missing best practices

Severity

Informational

Description

The contract LiquidStaking.sol defines a function “claimAll()” on Line 474. This function is checking if the user can claim their AVAX using an internal call to “_canBeClaimed()” and then calls the function “_claim()”.

```
...
          while (i < claimRequestCount) {
           if (!_canBeClaimed(claimReqs[msg.sender][i])) {
               i++;
               continue;
           }

           _claim(i);
...
```

When the code flow goes into the function “_claim()”, there is another validation on Line 757 to validate if the AVAX can be claimed which is redundant.

```
...
       if (!_canBeClaimed(req)) {
           revert CantClaimBeforeDeadline();
       }
...
```

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L478-L482
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L757-L758

Impacts

Having redundant codes and validations cost a lot of gas and should be avoided.

Remediation

Instead of having validations for the claim inside both the functions, it is recommended to let it be inside the “claimAll()” function.
Since the “_claim()” is also being used by the claim() function, this validation should also be added to the “claim()”.

Retest:

The function logic has been updated to remove the redundant check.

Bug ID#12 [Fixed]

Missing Reentrancy Protections

Vulnerability Type

Reentrancy

Severity

Low

Description

In a Reentrancy attack, a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

The smart contracts were missing reentrancy protection on all the functions making external calls.

Affected Code

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L409-L435

Impacts

Lacking reentrancy protection could allow threat actors to abuse the functions and reenter the contract.

However, it should be noted that right now there's no impact due to all the state changes being done before the external calls.

This may change in the future during code refactoring and may introduce reentrancy vulnerabilities due to missing validation.

Remediation

Add a Reentrancy guard to all the functions making external calls.

Retest:

Critical functions like deposit(), claim(), tokenRecieved() etc. have reentrancy guard in place.

Bug ID#13

Multiple Zero Address Validations Missing

Vulnerability Type
Missing Input Validation

Severity
Low

Description:
Multiple Solidity 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 Variables and Line Numbers

- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/AddressStore.sol#L21-L22 - **value**
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/FeeVault.sol#L74-L77 - **recipient**
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/FeeVault.sol#L87-L110 - **to**
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/LiquidStaking.sol#L463-L490 - **to**
- https://github.com/persistenceOne/stkAVAX-contracts/blob/3c092460436262e0f5f98b3dff130bdd9d1dcfac/contracts/StakedAVAXToken.sol#L103-L105 - **_address**

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:

-

6. Appendix 1

6.1 Files in scope:

Contracts Description Table:

Contract	Type	Bases		
L	**Function Name**	**Visibility**	**Mutability**	**Modifiers**
StakedAVAXToken	Implementation	ERC777, AccessControlEnumerable, Pausable		
L		Public ❗	🛑	ERC777
L	burn	Public ❗	🛑	onlyRole whenNotPaused
L	operatorBurn	Public ❗	🛑	onlyRole whenNotPaused
L	mint	Public ❗	🛑	onlyRole whenNotPaused
L	selfDestruct	Public ❗	🛑	onlyRole whenPaused
L	pause	Public ❗	🛑	onlyRole
L	unpause	Public ❗	🛑	onlyRole
AccessControlEnumerable	Implementation	IAccessControlEnumerable, AccessControl		
L	supportsInterface	Public ❗		NO ❗
L	getRoleMember	Public ❗		NO ❗

└	getRoleMemberCount	Public ❗		NO ❗
└	_grantRole	Internal 🔒	🛑	
└	_revokeRole	Internal 🔒	🛑	
IAccessControlEnumerable	Interface	IAccessControl		
└	getRoleMember	External ❗		NO ❗
└	getRoleMemberCount	External ❗		NO ❗
IAccessControl	Interface			
└	hasRole	External ❗		NO ❗
└	getRoleAdmin	External ❗		NO ❗
└	grantRole	External ❗	🛑	NO ❗
└	revokeRole	External ❗	🛑	NO ❗
└	renounceRole	External ❗	🛑	NO ❗
AccessControl	Implementation	Context, IAccessControl, ERC165		
└	supportsInterface	Public ❗		NO ❗
└	hasRole	Public ❗		NO ❗
└	_checkRole	Internal 🔒		
└	getRoleAdmin	Public ❗		NO ❗
└	grantRole	Public ❗	🛑	onlyRole
└	revokeRole	Public ❗	🛑	onlyRole
└	renounceRole	Public ❗	🛑	NO ❗
└	_setupRole	Internal 🔒	🛑	
└	_setRoleAdmin	Internal 🔒	🛑	
└	_grantRole	Internal 🔒	🛑	
└	_revokeRole	Internal 🔒	🛑	

Context	Implementation			
└	_msgSender	Internal 🔒		
└	_msgData	Internal 🔒		
Strings	Library			
└	toString	Internal 🔒		
└	toHexString	Internal 🔒		
└	toHexString	Internal 🔒		
ERC165	Implementation	IERC165		
└	supportsInterface	Public ❗️		NO❗️
IERC165	Interface			
└	supportsInterface	External ❗️		NO❗️
EnumerableSet	Library			
└	_add	Private 🔐	🛑	
└	_remove	Private 🔐	🛑	
└	_contains	Private 🔐		
└	_length	Private 🔐		
└	_at	Private 🔐		
└	_values	Private 🔐		
└	add	Internal 🔒	🛑	
└	remove	Internal 🔒	🛑	
└	contains	Internal 🔒		
└	length	Internal 🔒		
└	at	Internal 🔒		
└	values	Internal 🔒		
└	add	Internal 🔒	🛑	

└	remove	Internal 🔒	🛑	
└	contains	Internal 🔒		
└	length	Internal 🔒		
└	at	Internal 🔒		
└	values	Internal 🔒		
└	add	Internal 🔒	🛑	
└	remove	Internal 🔒	🛑	
└	contains	Internal 🔒		
└	length	Internal 🔒		
└	at	Internal 🔒		
└	values	Internal 🔒		
ERC777	Implementation	Context, IERC777, IERC20		
└		Public ❗️	🛑	NO ❗️
└	name	Public ❗️		NO ❗️
└	symbol	Public ❗️		NO ❗️
└	decimals	Public ❗️		NO ❗️
└	granularity	Public ❗️		NO ❗️
└	totalSupply	Public ❗️		NO ❗️
└	balanceOf	Public ❗️		NO ❗️
└	send	Public ❗️	🛑	NO ❗️
└	transfer	Public ❗️	🛑	NO ❗️
└	burn	Public ❗️	🛑	NO ❗️
└	isOperatorFor	Public ❗️		NO ❗️
└	authorizeOperator	Public ❗️	🛑	NO ❗️
└	revokeOperator	Public ❗️	🛑	NO ❗️
└	defaultOperators	Public ❗️		NO ❗️
└	operatorSend	Public ❗️	🛑	NO ❗️

└	operatorBurn	Public ❗️	🛑	NO ❗️
└	allowance	Public ❗️		NO ❗️
└	approve	Public ❗️	🛑	NO ❗️
└	transferFrom	Public ❗️	🛑	NO ❗️
└	_mint	Internal 🔒	🛑	
└	_mint	Internal 🔒	🛑	
└	_send	Internal 🔒	🛑	
└	_burn	Internal 🔒	🛑	
└	_move	Private 🔐	🛑	
└	_approve	Internal 🔒	🛑	
└	_callTokensToSend	Private 🔐	🛑	
└	_callTokensReceived	Private 🔐	🛑	
└	_beforeTokenTransfer	Internal 🔒	🛑	
IERC777	Interface			
└	name	External ❗️		NO ❗️
└	symbol	External ❗️		NO ❗️
└	granularity	External ❗️		NO ❗️
└	totalSupply	External ❗️		NO ❗️
└	balanceOf	External ❗️		NO ❗️
└	send	External ❗️	🛑	NO ❗️
└	burn	External ❗️	🛑	NO ❗️
└	isOperatorFor	External ❗️		NO ❗️
└	authorizeOperator	External ❗️	🛑	NO ❗️
└	revokeOperator	External ❗️	🛑	NO ❗️
└	defaultOperators	External ❗️		NO ❗️
└	operatorSend	External ❗️	🛑	NO ❗️
└	operatorBurn	External ❗️	🛑	NO ❗️

IERC777Recipient	Interface			
L	tokensReceived	External ❗	🛑	NO ❗
IERC777Sender	Interface			
L	tokensToSend	External ❗	🛑	NO ❗
IERC20	Interface			
L	totalSupply	External ❗		NO ❗
L	balanceOf	External ❗		NO ❗
L	transfer	External ❗	🛑	NO ❗
L	allowance	External ❗		NO ❗
L	approve	External ❗	🛑	NO ❗
L	transferFrom	External ❗	🛑	NO ❗
Address	Library			
L	isContract	Internal 🔒		
L	sendValue	Internal 🔒	🛑	
L	functionCall	Internal 🔒	🛑	
L	functionCall	Internal 🔒	🛑	
L	functionCallWithValue	Internal 🔒	🛑	
L	functionCallWithValue	Internal 🔒	🛑	
L	functionStaticCall	Internal 🔒		
L	functionStaticCall	Internal 🔒		
L	functionDelegateCall	Internal 🔒	🛑	
L	functionDelegateCall	Internal 🔒	🛑	
L	verifyCallResult	Internal 🔒		
IERC1820Registry	Interface			

└	setManager	External ❗️	🛑	NO ❗️
└	getManager	External ❗️		NO ❗️
└	setInterfaceImplementer	External ❗️	🛑	NO ❗️
└	getInterfaceImplementer	External ❗️		NO ❗️
└	interfaceHash	External ❗️		NO ❗️
└	updateERC165Cache	External ❗️	🛑	NO ❗️
└	implementsERC165Interface	External ❗️		NO ❗️
└	implementsERC165InterfaceNoCache	External ❗️		NO ❗️
Pausable	Implementation	Context		
└		Public ❗️	🛑	NO ❗️
└	paused	Public ❗️		NO ❗️
└	_pause	Internal 🔒	🛑	whenNotPaused
└	_unpause	Internal 🔒	🛑	whenPaused

Legend

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

6.2 Disclosure:

The Reports neither endorse nor condemn any specific project or team, nor do they guarantee the security of any specific project. The contents of this report do not, and should not be interpreted as having any bearing on, the economics of tokens, token sales, or any other goods, services, or assets.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. There is no warranty or representation made by this report to any Third Party in regards to the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business.

In no way should a third party use these reports to make any decisions about buying or selling a token, product, service, or any other asset. It should be noted that this report is not investment advice, is not intended to be relied on as investment advice, and has no endorsement of this project or team. It does not serve as a guarantee as to the project's absolute security.

CredShields Audit team owes no duty to any third party by virtue of publishing these Reports.