CredShields

Smart Contract Audit

Jun 5th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Stable Jack between May 5th, 2024, and May 15th, 2024. And a retest was performed on May 30th, 2024.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

- Aditya Dixit (Research Team Lead)
- Shreyas Koli (Auditor)
- Naman Jain (Auditor)

Prepared for

Stable Jack

Table of Contents

CRED SHIELDS

1. Executive Summary

Stable Jack engaged CredShields to perform a smart contract audit from May 5th, 2024, to May 15th, 2024. During this timeframe, Thirty-three (33) vulnerabilities were identified. **A retest was performed on May 30th, 2024, and all the bugs have been addressed.**

During the audit, Five (5) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Stable Jack" 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
Smart Contract	3	2	11	6	2	9	**33**
	3	**2**	**11**	**6**	**2**	**9**	**33**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Smart Contract ’s scope during the testing window while abiding by the policies set forth by Smart Contract ’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 Stable Jack'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 Stable Jack can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Stable Jack can future-proof its security posture and protect its assets.

2. Methodology

Stable Jack engaged CredShields to perform a Stable Jack 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 May 5th, 2024, to May 15th, 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/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/

Table: List of Files in Scope

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

Stable Jack 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, Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

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 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 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, Thirty-three (33) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Unfair Token Minting	Critical	Business Logic
Unauthorized Fund Drainage via _distributeRebalancePoolRewards() Function	Critical	Business Logic
Incorrect Amount Passed to wsAVAX Deposit in _mintHelper() Function	Critical	Business Logic
Rounding Error in Token Minting	High	Rounding Error
Inconsistent Price Validation in _fetchTwapPrice()	High	Business Logic

Lack of Minting Pause Validation in mintBothTokens() Function	Medium	Missing Input Validation
Lack of Collateral Ratio Validation in mintBothTokens() Function	Medium	Missing Input Validation
Unused Strategy Contract Invocation in _transferBaseToken() Function	Medium	Invalid External Call
Missing Price Feed Validation	Medium	Missing Input Validation
Chainlink Oracle Min/Max price validation	Medium	Missing Input Validation
Lack of Fee Calculation in mintBothTokens() Function	Medium	Missing Input Validation
Potential Integer Overflow in Unlock() Function	Medium	Arithmetic Overflow
Potential Integer Overflow in Reward Rate Calculation	Medium	Arithmetic Overflow
Transaction Reversion Due to Invalid Oracle Price in _fetchTwapPrice()	Medium	Business Logic
Incorrect Usage of submit Function in mintaToken() Function	Medium	Business Logic
Revert Risk Due to Incorrect Input Handling in mintaToken() Function	Medium	Business Logic
Missing Zero Address Validations	Low	Missing Input Validation
Use of Multiple Pragma Versions	Low	Missing Best Practices
Floating and Outdated Pragma	Low	Floating Pragma

Use Ownable2Step	Low	Missing Best Practices
Missing Parameter Validation in Constructor	Low	Missing Input Validation
Missing Events in Important Functions	Low	Missing Best Practices
Dead Code	Informational	Code With No Effects
Require with Empty Message	Informational	Code optimization
Large Number Literals	Gas	Gas & Missing Best Practices
Gas Optimization in Require Statements	Gas	Gas Optimization
Use of SafeMath	Gas	Gas Optimization
Multiplication/Division by 2 should use Bit-Shifting	Gas	Gas Optimization
Code Optimization by using max and min	Gas	Gas Optimization
State Variable Can Be Marked As Constants	Gas	Gas Optimization
Splitting Require Statements	Gas	Gas Optimization
Gas Optimization in Increments	Gas	Gas Optimization
Gas Optimization for State Variables	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	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Vulnerable	Contracts use outdated pragma
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
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	Vulnerable	Bug ID #23
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

4. Remediation Status

Stable Jack is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on May 30th, 2024, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Unfair Token Minting	Critical	Fixed
Unauthorized Fund Drainage via _distributeRebalancePoolRewards() Function	Critical	Fixed
Incorrect Amount Passed to wsAVAX Deposit in _mintHelper() Function	Critical	Fixed
Rounding Error in Token Minting	High	Fixed
Inconsistent Price Validation in _fetchTwapPrice()	High	Fixed
Lack of Minting Pause Validation in mintBothTokens() Function	Medium	Fixed
Lack of Collateral Ratio Validation in mintBothTokens() Function	Medium	Fixed
Unused Strategy Contract Invocation in _transferBaseToken() Function	Medium	Fixed

Missing Price Feed Validation	Medium	Fixed
Chainlink Oracle Min/Max price validation	Medium	Fixed
Lack of Fee Calculation in mintBothTokens() Function	Medium	Fixed
Potential Integer Overflow in Unlock() Function	Medium	Fixed
Potential Integer Overflow in Reward Rate Calculation	Medium	Fixed
Transaction Reversion Due to Invalid Oracle Price in _fetchTwapPrice()	Medium	Fixed
Incorrect Usage of submit Function in mintaToken() Function	Medium	Fixed
Revert Risk Due to Incorrect Input Handling in mintaToken() Function	Medium	Fixed
Missing Zero Address Validations	Low	Fixed
Use of Multiple Pragma Versions	Low	Won't Fix
Floating and Outdated Pragma	Low	Won't Fix
Use Ownable2Step	Low	Won't Fix
Missing Parameter Validation in Constructor	Low	Fixed
Missing Events in Important Functions	Low	Fixed
Dead Code	Informational	Fixed

Require with Empty Message	Informational	Fixed
Large Number Literals	Gas	Won't Fix
Gas Optimization in Require Statements	Gas	Fixed
Use of SafeMath	Gas	Won't Fix
Multiplication/Division by 2 should use Bit-Shifting	Gas	Fixed
Code Optimization by using max and min	Gas	Won't Fix
State Variable Can Be Marked As Constants	Gas	Won't Fix
Splitting Require Statements	Gas	Fixed
Gas Optimization in Increments	Gas	Won't Fix
Gas Optimization for State Variables	Gas	Fixed

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID # 1 [Fixed]

Unfair Token Minting

Vulnerability Type
Business Logic

Severity
Critical

Description
The contract provides three functions for minting tokens using base tokens (SAVAX). `mintBothWithSAVAX()` Mints both xTokens and aTokens. `mintxTokenWithSAVAX()` Mints only xTokens. `mintaTokenWithSAVAX()` Mints only aTokens. The issue lies in the fact that `mintxTokenWithSAVAX()` and `mintaTokenWithSAVAX()` mint more tokens individually than `mintBothWithSAVAX()` does for the same amount of SAVAX. This discrepancy creates an unfair situation for users who choose to mint both tokens simultaneously through `mintBothWithSAVAX()`, as they receive fewer tokens than if they had used the other two functions separately.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/sAVAX/sAVAXGateway.sol#L119-L134
- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/sAVAX/sAVAXGateway.sol#L174-L187
- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/sAVAX/sAVAXGateway.sol#L157-L171
- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/sAVAX/sAVAXGateway.sol#L139-L154

Impacts

This unfair distribution can lead to an economic imbalance, where users minting tokens separately gain an advantage over those minting both tokens simultaneously. Users who are aware of this discrepancy may feel unfairly treated and lose trust in the protocol.

Remediation

It is recommended to ensure that the minting logic for `mintBothWithSAVAX()`, `mintxTokenWithSAVAX()`, and `mintaTokenWithSAVAX()` is consistent and fair. This means that the total amount of tokens minted using `mintBothWithSAVAX()` should be equivalent to the combined amount of tokens minted using `mintxTokenWithSAVAX()` and `mintaTokenWithSAVAX()` for the same amount of SAVAX

Test-Case

```
context("mintBothWithSAVAX() gives less token than mintXToken() & mintaToken()", async
() => {
       beforeEach(async () => {
         await avax.connect(signer1).mint(signer1.address, ethers.parseEther("1000"));
         await avax.connect(signer2).mint(signer2.address, ethers.parseEther("1000"));
         await avax.connect(signer3).mint(signer3.address, ethers.parseEther("1000"));
         await avax.connect(signer4).mint(signer4.address, ethers.parseEther("1000"));
         await avax.connect(user).mint(user.address, ethers.parseEther("1000"));
         await avax.mint(deployer.address, ethers.parseEther("10"));

         await avax.connect(signer1).approve(market.getAddress(), MaxUint256);
         await avax.connect(signer2).approve(market.getAddress(), MaxUint256);
         await avax.connect(signer3).approve(market.getAddress(), MaxUint256);
         await avax.connect(signer4).approve(market.getAddress(), MaxUint256);
         await avax.connect(user).approve(market.getAddress(), MaxUint256);
         await avax.approve(market.getAddress(), MaxUint256);
         await market.mint(ethers.parseEther("1"), deployer.address, 0, 0);

                    await  market.connect(signer1).mintXToken(ethers.parseEther("500"),
signer1.address, 1);
                    await  market.connect(signer1).mintaToken(ethers.parseEther("500"),
signer1.address, 1);
```

```
                    await  market.connect(signer2).mintXToken(ethers.parseEther("500"),
signer2.address, 1);
                    await  market.connect(signer2).mintaToken(ethers.parseEther("500"),
signer2.address, 1);

                    await  market.connect(signer3).mintXToken(ethers.parseEther("500"),
signer3.address, 1);
                    await  market.connect(signer3).mintaToken(ethers.parseEther("500"),
signer3.address, 1);

                    await  market.connect(signer4).mintXToken(ethers.parseEther("500"),
signer4.address, 1);
                    await  market.connect(signer4).mintaToken(ethers.parseEther("500"),
signer4.address, 1);

       });

       it("mint using mintBothWithSAVAX()", async () => {
                          await   avax.connect(user).transfer(sAVAXGateway.getAddress(),
ethers.parseEther("1"));
                                        await    sAVAXGateway.mintBothWithSAVAX(0,    0,
ethers.parseEther("1"),user.address);

                             console.log("xToken   balance   of   user   :   ",   await
xToken.balanceOf(user.address));
                             console.log("aToken   balance   of   user   :   ",   await
aToken.balanceOf(user.address));
       });

       it("mint using mintXToken", async () => {
            await market.connect(user).mintXToken(ethers.parseEther("1"), user.address,
0);
            await market.connect(user).mintaToken(ethers.parseEther("1"), user.address,
0);

                             console.log("xToken   balance   of   user   :   ",   await
xToken.balanceOf(user.address));
```

```
                        console.log("aToken balance of user : ", await
aToken.balanceOf(user.address));
      });
    });
```

Test-Case Output

```
  sAVAXGateway.spec
    mint aToken
Balance of Treasury Before minting :  1000000000000000000n
Balance of user after calling mintaToken with 1wei : 0n
Balance of Treasury After minting :  1000000000000000001n
      ✔ should succeed
    mintBothWithSAVAX() gives less token than mintXToken() & mintaToken()
xToken balance of user :  496848209783156833 08n
aToken balance of user :  503151790216843166 91n
      ✔ mint using mintBothWithSAVAX()
xToken balance of user :  985000000000000000 0n
aToken balance of user :  997500000000000000 0n
      ✔ mint using mintXToken
```

Retest

This vulnerability has been addressed by removing mintBothTokens() and minting both tokens separately.

https://github.com/stable-jack/v1-jack-contracts/blob/2734a3836c1339e995f979f839009ac942175fab/contracts/jack/sAVAX/sAVAXGateway.sol#L146-L147

Bug ID # 2 [Fixed]

Unauthorized Fund Drainage via _distributeRebalancePoolRewards() Function

Vulnerability Type
Business logic

Severity
Critical

Description
In the `_distributeRebalancePoolRewards` function, the contract mistakenly approves an unnecessary allowance to `msg.sender` before transferring funds to the rebalance pool. This allowance allows any attacker to call `transferFrom` from the base token and drain all the funds from the contract without proper authorization.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/9dd1a5da99c6a2f3e7377fe54bb01e664866ddc2/contracts/jack/sAVAX/sAVAXTreasury.sol#L31

Impacts
An attacker can exploit this vulnerability to drain all funds from the contract by calling `transferFrom` from the base token, utilizing the unnecessary allowance approved to `msg.sender`.

Remediation
To mitigate this vulnerability it is recommended to remove the unnecessary approval of allowance to `msg.sender` in the `_distributeRebalancePoolRewards` function to prevent unauthorized fund transfers.

Test-Case

```
it("Test #8: Harvest Function Scenario 1", async () => {
```

```
                    await  market.connect(signer1).mintXToken(ethers.parseEther("500"),
signer1.address, 1);
                    await  market.connect(signer1).mintaToken(ethers.parseEther("500"),
signer1.address, 1);

                    await  market.connect(signer2).mintXToken(ethers.parseEther("500"),
signer2.address, 1);
                    await  market.connect(signer2).mintaToken(ethers.parseEther("500"),
signer2.address, 1);

                    await  market.connect(signer3).mintXToken(ethers.parseEther("500"),
signer3.address, 1);
                    await  market.connect(signer3).mintaToken(ethers.parseEther("500"),
signer3.address, 1);



                    await  market.connect(signer4).mintXToken(ethers.parseEther("500"),
signer4.address, 1);
                    await  market.connect(signer4).mintaToken(ethers.parseEther("500"),
signer4.address, 1);



                        await  pool.connect(signer1).deposit(ethers.parseEther("100"),
signer1.address);
                        await  pool.connect(signer2).deposit(ethers.parseEther("200"),
signer2.address);
                        await  pool.connect(signer3).deposit(ethers.parseEther("300"),
signer3.address);
                        await  pool.connect(signer4).deposit(ethers.parseEther("400"),
signer4.address);

          await avax.mint(await treasury.getAddress(), ethers.parseEther('1500'));

          console.log("balance of treasury: ", await treasury.totalBaseToken());
               console.log("real balance of treasury: ", await avax.balanceOf(await
treasury.getAddress()));
          console.log("balance of signer4", await avax.balanceOf(signer4));
          await treasury.connect(signer4).harvest();
```

```
                await avax.connect(signer4).transferFrom(await treasury.getAddress(),
signer4.address, ethers.parseEther("100"));
           console.log('-----------------after attack-------------');
           console.log("balance of treasury: ", await treasury.totalBaseToken());
                console.log("real balance of treasury: ", await avax.balanceOf(await
treasury.getAddress()));
           console.log("balance of signer4", await avax.balanceOf(signer4));
      });
```

Test-Case-Output

```
  StaleJack.spec
    test 1
balance of treasury:  4965000000000000000000n
real balance of treasury:  6465000000000000000000n
balance of signer4 0n
-----------------after attack-------------
balance of treasury:  4965000000000000000000n
real balance of treasury:  4865000000000000000000n
balance of signer4 100000000000000000000n
      ✔ Test #8: Harvest Function Scenario 1 (115ms)


  1 passing (1s)
```

Retest

This vulnerability has been fixed by removing approve call to msg.sender in sAVAXTreasury.sol .
https://github.com/stable-jack/v1-jack-contracts/blob/postaudit/contracts/jack/sAVAX/sAVAXTreasury.sol#L28-L34

Bug ID #3 [Fixed]

Incorrect Amount Passed to wsAVAX Deposit in _mintHelper() Function

Vulnerability Type
Logical Error Vulnerability

Severity
Critical

Description
In the `_mintHelper()` function, there is a vulnerability related to passing an incorrect amount argument to the deposit function when minting `wsAVAX` tokens. The amount passed to the deposit function should correspond to the returned value after performing the token swap. However, the function currently uses the initial user input amount instead of the swapped amount, potentially leading to incorrect minting of `wsAVAX` tokens.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/sAVAX/sAVAXGateway.sol#L251

Impacts
If the wrong amount is used for minting `wsAVAX` tokens, it may result in the minted tokens not accurately reflecting the value obtained through the token swap, leading to inconsistencies in token balances and potential financial losses.

Remediation
To mitigate the vulnerability, it is essential to ensure that the correct amount is passed to the deposit function when minting `wsAVAX` tokens in the `_mintHelper()` function. This can be achieved by updating the function to use the returned value after performing the token swap as the amount argument for the deposit function.

Retest

This vulnerability has been fixed by passing swappedAmount to deposit() .
https://github.com/stable-jack/v1-jack-contracts/blob/main/contracts/jack/sAVAX/sAVAXGateway.sol#L290

Bug ID #4 [Fixed]

Rounding Error in Token Minting

Vulnerability Type
Rounding Error

Severity
High

Description
The `mintaToken()` function in the contract uses a formula to calculate the amount of tokens a user will receive for a given amount of base tokens. However, due to rounding errors in Solidity, if the user inputs a very small amount of base tokens (e.g., 1 wei), the formula may result in zero tokens being minted for the user. This occurs because the division operation may truncate the result to zero due to rounding. This vulnerability also affects in case of redeeming, Minting, and liquidation.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L232-L258
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L261-L299
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L302-L345
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L375-L420
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L423-L506
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L509-L558

Impacts

Users may lose their base tokens without receiving any corresponding tokens in return due to the rounding error in the `mintaToken` function. This can lead to financial losses for users and create a negative experience.

Remediation

Set a minimum input limit for base tokens in the `mintaToken` function to prevent users from losing tokens due to rounding errors. Revert transaction if `StableCoinMath` returns zero amount.

Test-Case

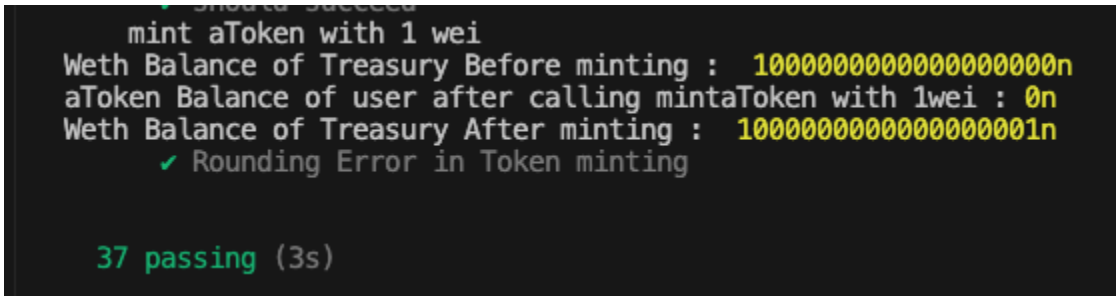
```
context("mint aToken with 1 wei", async () => {
   beforeEach(async () => {
     await oracle.setPrice(10);
     await treasury.initializePrice();
     await weth.connect(signer).deposit({ value: ethers.parseEther("10") });
     await weth.deposit({ value: ethers.parseEther("10") });

     await weth.connect(signer).approve(market.getAddress(), MaxUint256);
     await weth.approve(market.getAddress(), MaxUint256);
     await market.mint(ethers.parseEther("1"), deployer.address, 0, 0);
   });

   it("Rounding Error in Token minting", async () => {

            console.log("Weth Balance of Treasury Before minting : ", await
weth.balanceOf(treasury.getAddress()));
     await market.connect(signer).mintaToken( 1, signer.address, 0);
      console.log("aToken Balance of user after calling mintaToken with 1wei :", await
aToken.balanceOf(signer.address));
            console.log("Weth Balance of Treasury After minting : ", await
weth.balanceOf(treasury.getAddress()));
   });
 });
```

Test-Case-Output



Retest

This vulnerability has been addressed by reverting the transaction if StableCoinMath returns zero.

Bug ID #5 [Fixed]

Inconsistent Price Validation in _fetchTwapPrice()

Vulnerability Type
Business Logic

Severity
High

Description
The function `_fetchTwapPrice()` fetches the Time-Weighted Average Price (TWAP) from an oracle and processes it based on the type of swap specified by the SwapKind enum. The function makes a call to `getPrice()` from the oracle, which returns a boolean `_isValid` indicating if the price is valid. However, there is inconsistent validation of the `_isValid` flag depending on the SwapKind.

For `SwapKind.MintaToken` and `SwapKind.MintXToken`, the function properly checks if `_isValid` is true and reverts the transaction if it is not. However, for `SwapKind.None`, there is no validation on `_isValid`, and the function proceeds with an invalid price, which can result in incorrect transactions and potential exploitation.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/testing/contracts/jack/Treasury.sol#L667-L679

Impact
Transactions using SwapKind.None may proceed with invalid prices, leading to incorrect financial calculations. Malicious actors could exploit this flaw to execute transactions at incorrect prices, potentially causing financial losses to the protocol and its users.

Remediation
To mitigate this vulnerability, Modify the `_fetchTwapPrice()` function to ensure that `_isValid` is checked for all swap kinds, including `SwapKind.None`.

Retest

`_fetchTwapPrice()` was redesigned. This Vulnerability has been fixed.

Bug ID # 6 [Fixed]

Lack of Minting Pause Validation in `mintBothTokens()` Function

Vulnerability Type
Missing Validation Vulnerability

Severity
Medium

Description
The `mintBothTokens()` function allows minting both `aToken` and `XToken` simultaneously. However, unlike the `mintaToken()` and `mintXToken()` functions, it lacks a validation check to ensure that minting is not paused before proceeding. This omission presents a vulnerability as it bypasses the pause mechanism intended to prevent minting during certain conditions.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/Market.sol#L348-L373

Impact
The absence of the pause validation in the `mintBothTokens()` function can have a significant impact. It allows users to exploit the vulnerability, enabling them to mint both aToken and XToken even when minting is supposed to be paused. This could result in an undesired inflation of tokens or disrupt the carefully balanced token economics.

Remediation
To address this vulnerability, it is recommended to implement a consistent pause validation mechanism across all minting functions, including the `mintBothTokens()` function. This can be achieved by adding a require statement to check

Retest

This vulnerability has been addressed by removing `mintBothTokens()` function from `market.sol`.

Bug ID # 7 [Fixed]

Lack of Collateral Ratio Validation in mintBothTokens() Function

Vulnerability Type
Missing Validation Vulnerability

Severity
Medium

Description
In the `mintaToken()` function, there is a validation check based on the `_collateralRatio` to determine if minting should be paused. However, this validation is absent in the `mintBothTokens()` function, allowing users to mint both `aToken` and `XToken` without considering the current collateral ratio. This omission poses a vulnerability as it bypasses an essential safeguard against minting tokens when the collateral ratio is below a certain threshold.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/Market.sol#L348-L373

Impact
The absence of collateral ratio validation in the `mintBothTokens()` function can lead to several significant impacts.

- Firstly, it allows users to mint tokens without considering the system's financial stability, potentially leading to an imbalance in the collateralization ratio and increasing the risk of insolvency.
- Secondly, it undermines the effectiveness of the collateral ratio as a control mechanism, as minting can occur even when the system is not financially stable according to predetermined thresholds.

Remediation

To mitigate this vulnerability, it is recommended to add collateral ratio validation into the `mintBothTokens()` function. This can be achieved by adding a `require` statement to check if the collateral ratio exceeds the specified threshold before allowing minting to proceed.

Retest

This vulnerability has been addressed by removing `mintBothTokens()` function from `market.sol`.

Bug ID # 8 [Fixed]

Unused Strategy Contract Invocation in _transferBaseToken() Function

Vulnerability Type
Invalid External Call

Severity
Medium

Description
The `_transferBaseToken()` function contains an invocation to `IAssetStrategy(strategy).withdrawToTreasury(_diff)`, assuming the presence of a strategy contract. However, as per client specifications, there will not be any strategy contract used in the system. If the strategy contract does not exist, this external call will revert, potentially leading to the loss of funds or a deadlock situation.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L146-L149
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L500-L503
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L507
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L594

Impacts
In the absence of a strategy contract, this line of code will result in a revert, potentially disrupting the execution flow and confusing for developers interacting with the contract.

Remediation

Since the client has confirmed that there will not be any strategy contract utilized in the system, the unused invocation to `IAssetStrategy(strategy).withdrawToTreasury(_diff)` should be removed from the `_transferBaseToken` function. This will eliminate the risk of unintended behavior and simplify the codebase.

Retest
This has been fixed and the strategy logic is now removed.

Bug ID# 9 [Fixed]

Missing Price Feed Validation

Vulnerability Type
Input Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
The contract was found to be using `latestRoundData()` without proper input validations on the returned parameters which might result in a stale and outdated price.

Vulnerable Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/price-oracle/ChainlinkPriceOracle.sol#L38
- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L58-L62

Impacts
Having oracles with functions to fetch price feed without any validation might introduce erroneous or invalid price values that could result in an invalid price calculation further in the contract.

Remediation

It is recommended to have input validations for all the parameters obtained from the Chainlink price feed. Here's a sample implementation:

```
(uint80 roundID ,answer,, uint256 timestamp, uint80 answeredInRound) = AggregatorV3Interface(chainLinkAggregatorMap[underlying]).latestRoundData();

require(answer > 0, "Chainlink price <= 0");
require(answeredInRound >= roundID, "Stale price");
require(timestamp != 0, "Round not complete");
```

Retest

This Vulnerability is fixed by adding validation on returned parameters of latestRoundData()

Bug ID# 10 [Fixed]

Chainlink Oracle Min/Max price validation

Vulnerability Type
Input Validation

Severity
Medium

Description
Chainlink has a library `AggregatorV3Interface` with a function called `latestRoundData()`. This function returns the price feed among other details for the latest round.
Chainlink aggregators have a built in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset.

Vulnerable Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/price-oracle/ChainlinkPriceOracle.sol#L38
- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L58-L62

Impacts
This would allow user to store their allocations with the asset but at the wrong price.

Remediation
The contract should check the returned answer/price against the `minPrice`/`maxPrice` and revert if the answer is outside of the bounds.

```
if (answer >= maxPrice or answer <= minPrice) revert(); // eg
```

Retest

Client's Comment: sAVAX/USD and AVAX/USD do not have min/max Answer values. From the chainlink docs, it is recommended to create our own monitoring alerts. Hexagate will be very useful here. So the bug can be considered fixed in a way that: we will not check minPrice and maxPrice, since they have no utility in the protocol. Since there is no min/max Price now, if the price is not valid, it reverts

Instead, we are going to check if the Oracle is safe in two ways:

1. Depeg of sAVAX/AVAX
2. Hexagate monitoring, alert us + make the price invalid if chainlink oracle stalls/ volatility > 25% etc

Bug ID # 11 [Fixed]

Lack of Fee Calculation in `mintBothTokens()` Function

Vulnerability Type
Missing Validation Vulnerability

Severity
Medium

Description
The `mintBothTokens()` function lacks `fee` calculation functionality present in the `mintaToken()` and `mintXToken()` functions. This absence means that users can mint both aToken and XToken without being subject to any fees. This oversight presents a vulnerability as it deviates from the established fee structure and may lead to inconsistencies in token minting processes.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/Market.sol#L348-L373

Impact
The absence of fee calculation in the `mintBothTokens()` function can have several significant impacts.

- Firstly, it undermines the revenue generation mechanism of the system, potentially leading to financial losses for the platform.
- Secondly, it creates inconsistencies in the token minting process, as minting both aToken and XToken through `mintBothTokens()` would not incur fees compared to minting them individually through mintaToken and mintXToken.

Remediation
To mitigate this vulnerability, it is recommended to add fee calculation functionality into the `mintBothTokens()` function. This can be achieved by adapting the fee calculation logic

from the `mintaToken()` and `mintXToken()` functions to accommodate the minting of both tokens simultaneously.

Retest

This vulnerability has been addressed by removing `mintBothTokens()` function from `market.sol`.

Bug ID # 12 [Fixed]

Potential Integer Overflow in Unlock() Function

Vulnerability Type
Arithmetic Overflow Vulnerability

Severity
Medium

Description
The `unlock()` function introduces a potential vulnerability stemming from typecasting the `_amount` parameter from `uint256` to `uint128`. This typecasting operation may lead to underflow risks under certain circumstances, particularly when the _amount parameter is large enough to exceed the maximum value representable by a `uint128`.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/jack/rebalance-pool/RebalancePool.sol#L402

Impacts
In the event of an underflow, the unlocked token amount may be incorrectly calculated, potentially resulting in the loss of funds or the inability to access rightful token holdings.

Remediation
Evaluate the necessity of using a uint128 data type for storing the unlocked token amount. If possible, consider utilizing a wider integer type, such as uint256, to accommodate larger values and mitigate the risk of underflow.

Retest
This Vulnerability has been fixed by using uint256 data type in unlock struct.

Bug ID # 13 [Fixed]

Potential Integer Overflow in Reward Rate Calculation

Vulnerability Type
Arithmetic Overflow Vulnerability

Severity
Medium

Description
Within the `_notifyReward` function, there's a potential vulnerability highlighted regarding integer overflow during reward rate calculation. Specifically, the vulnerability arises from the typecasting operation where a `uint256` value is typecasted to `uint128`, which can lead to unexpected behavior if the result exceeds the maximum value represented by `uint128`.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L802

Impacts
If an overflow occurs during the reward rate calculation, it can result in incorrect reward rates being distributed to users, potentially leading to financial losses or unfair distribution of rewards. Moreover, an attacker could exploit this vulnerability to manipulate the reward rate calculation in their favor.

Remediation
It is recommended to evaluate the data types used for storing reward rates and ensure they can accommodate the expected range of values without risk of overflow. If necessary, consider using wider integer types or alternative approaches to handle large values.

Retest
This is now fixed by using uint256 instead of uint128.

Bug ID # 14 [Fixed]

Transaction Reversion Due to Invalid Oracle Price in _fetchTwapPrice()

Vulnerability Type
Business Logic

Severity
Medium

Description
In the `_fetchTwapPrice()` function, there is an internal call to the `getPrice()` function of an external price oracle contract. The `getPrice()` function returns four values: `_isValid`, `_safePrice`, `_minPrice`, and `_maxPrice`. The `_isValid` value is a boolean indicating whether the oracle price is considered valid. In the `_fetchTwapPrice()` function, there is a requirement that `_isValid` must be true for certain types of swaps (MintaToken and MintXToken). However, the `getPrice()` function always returns `_isValid` as false, causing the transaction to always revert for these swap types due to the require statement.

Affected Code
- https://github.com/stable-jack/v1-jack-contracts/blob/d71819e6582524c8e80bac0e0164e730cc3c882d/contracts/jack/oracle/JacksAVAXTwapOracle.sol#L94

Impacts
All transactions involving `MintaToken` and `MintXToken` swap types will always revert, rendering these functionalities unusable

Remediation
Ensure that the `getPrice()` function of the price oracle contract returns a valid `_isValid` value when appropriate. This may involve reviewing and fixing the logic within the oracle contract itself.

Retest

This vulnerability has been addressed by correcting the logic in the `getPrice()` function.

Bug ID # 15 [Fixed]

Incorrect Usage of submit Function in mintaToken() Function

Vulnerability Type
Business logic

Severity
Medium

Description
The `mintaToken` and `mintXToken` functions in the contract attempt to convert ETH to AVAX by calling the submit function of the ILstAVAX contract. However, it mistakenly provides an `address (address(0))` as input to the submit function, which does not accept any inputs. As a result, this transaction will always revert, as the submit function does not expect or require any input parameters.

Affected Code
- https://github.com/stable-jack/v1-jack-contracts/blob/9dd1a5da99c6a2f3e7377fe54bb01e664866ddc2/contracts/jack/sAVAX/sAVAXGateway.sol#L62
- https://github.com/stable-jack/v1-jack-contracts/blob/9dd1a5da99c6a2f3e7377fe54bb01e664866ddc2/contracts/jack/sAVAX/sAVAXGateway.sol#L74

Impacts
Users who call the `mintaToken` function to convert ETH to AVAX will encounter a transaction revert due to the incorrect usage of the submit function. This prevents users from successfully converting their ETH to AVAX as intended.

Remediation
Modify the `mintaToken` function to correctly call the submit function of the `ILstAVAX` contract without providing any input parameters. The submit function does not require any inputs, so removing the `address(0)` parameter will resolve the issue

Retest

sAVAXGateway is now redesigned

Bug ID # 16 [Fixed]

Revert Risk Due to Incorrect Input Handling in mintaToken() Function

Vulnerability Type
Business logic

Severity
Medium

Description
In the `mintaToken` and `mintXToken` functions, the contract directly passes `msg.value` as the input to the `mintaToken` and `mintXToken` function of the IJackMarket contract. However, if the submit function of the `ILstAVAX` contract returns fewer tokens than the amount of ETH sent (msg.value), the call to `mintaToken` will revert. This vulnerability arises because the `mintaToken` function does not adjust the input based on the tokens received from the submit function.

Affected Code
- https://github.com/stable-jack/v1-jack-contracts/blob/9dd1a5da99c6a2f3e7377fe54bb01e664866ddc2/contracts/jack/sAVAX/sAVAXGateway.sol#L64
- https://github.com/stable-jack/v1-jack-contracts/blob/9dd1a5da99c6a2f3e7377fe54bb01e664866ddc2/contracts/jack/sAVAX/sAVAXGateway.sol#L76

Impacts
If the ILstAVAX contract returns fewer tokens than the amount of ETH sent by the user, the call to mintaToken will revert. Reverted transactions create a poor user experience and may discourage users from further interaction with the contract.

Remediation
Modify the `mintaToken` function to adjust the input value based on the tokens received from the submit function of the `ILstAVAX` contract. Calculate the amount of `AVAX` tokens received and pass this value as input to the mintaToken function

Retest

sAVAXGateway is now redesigned

Bug ID # 17 [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 Variables and Line Numbers

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L546-L550
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L554-L565
- https://github.com/stable-jack/v1-jack-contracts/blob/testing/contracts/jack/Market.sol#L662-L666
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L670-L674
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L678-L682
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L121-L126
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalancer/Rebalancer.sol#L16

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L522-L526
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L538-L542
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L546-L550
- https://github.com/stable-jack/v1-jack-contracts/blob/testing/contracts/helpers/PlatformFeeSpliter.sol#L208

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 is fixed by adding zero address validations wherever necessary.

Bug ID# 18 [Won't Fix]

Use of Multiple Pragma Versions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The contracts were found to be using multiple Solidity compiler versions across different Solidity files. This is not a good coding practice because different versions of the compiler have different caveats, breaking changes and introducing vulnerabilities.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackRebalancePoolSplitter.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackRebalancePoolRegistry.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackRebalancePool.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackMarket.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackTokenWrapper.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/interfaces/jack/IJackTreasury.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L3
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolRegistry.sol#L3

Impacts

Having different pragma versions across multiple contracts increases the chances of introducing vulnerabilities since each solidity version has its own set of issues and coding practices. Some major version upgrades may also break the contract logic if not handled properly.

Remediation

Instead of using different versions of the Solidity compiler with different bugs and security checks, it is better to use one version across all contracts.

Retest

Client's Comment: The most important contracts of the protocol: Treasury, Market, StableCoinMath, LeveragedToken, sAVAXGateway, Oracles are using 0.7.6 version, so no floating pragma for the essentials.

Bug ID # 19 [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. `,0.7.6`, `0.7.6`, `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
- Almost Every Files

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 `0.8.24` pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

Client's Comment: The most important contracts of the protocol: Treasury, Market, StableCoinMath, LeveragedToken, sAVAXGateway, Oracles are using 0.7.6 version, so no floating pragma for the essentials.

Bug ID # 20 [Won’t Fix]

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/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/ChainlinkPriceOracle.sol#L10
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/misc/PlatformFeeDistributor.sol#L10
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L16
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L27
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalancer/Rebalancer.sol#L11
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/helpers/PlatformFeeSpliter.sol#L10
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L15

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L61

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:

Client's Comment: At this stage Ownable2Step does not work for us since our solidity version is ^0.7.6 and Ownable2Step has ^0.8.0.

Bug ID # 21 [Fixed]

Missing Parameter Validation in Constructor

Vulnerability Type
Missing Validation

Severity
Low

Description
The `addRewardToken()` function in the given contract performs thorough validation on the parameters to ensure the integrity of the reward tokens being added. However, the constructor does not perform similar validation for the `RewardInfo[] _rewards` parameter. This discrepancy can lead to potential issues such as duplicate reward tokens or invalid percentages being set during contract initialization, which could undermine the intended safeguards implemented in the `addRewardToken()` function.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/ea558b579f4dc802a8bf57f65065e59263e3d10b/contracts/misc/PlatformFeeDistributor.sol#L80-L82

Impacts
If invalid data is passed to the constructor, the contract may be initialized with an inconsistent or undesirable state. Without validation, the gauge and treasury percentages may exceed acceptable limits, leading to incorrect calculations and potential financial discrepancies

Remediation
It is recommended to Implement the same validation checks in the constructor that are used in the `addRewardToken()` function to ensure the integrity of the initial reward tokens.

Retest:

This issue has been addressed by removing PlatformFeeDistributor.sol contract.

Bug ID # 22 [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/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/ReservePool.sol#L105-L107
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/helpers/PlatformFeeSpliter.sol#L124-L155
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L185-L195
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L571-L578
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L93-L112
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L172-L177

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L171-L214
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L222-L244
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L256-L258
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L261-L269
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L279-L294
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L299-L303
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L605-L623
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L664-L686
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L754-L783
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L819-L827
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L832-L885

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

Events has been added to necessary functions.

Bug ID # 23 [Fixed]

Dead Code

Vulnerability Type
Code With No Effects - SWC-135

Severity
Informational

Description
It is recommended to keep the production repository clean to prevent confusion and the introduction of vulnerabilities. The functions and parameters, contracts, and interfaces that are never used or called externally or from inside the contracts should be removed when the contract is deployed on the mainnet.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L146-L149
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L448
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L500C1-L503
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L507
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L657-L663

Impacts
This does not impact the security aspect of the Smart contract but prevents confusion when the code is sent to other developers or auditors to understand and implement. This reduces the overall size of the contracts and also helps in saving gas.

Remediation

If the library functions are not supposed to be used anywhere, consider removing them from the contract.

Retest

This vulnerability has been fixed by removing the strategy and computeMultiple

Bug ID # 24 [Fixed]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple `require` statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the `require` statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L49

Impacts
Having a short descriptive message in the `require` statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation
It is recommended to add a descriptive message, no longer than 32 bytes, inside the **require** statement to give more detail to the user about why the condition failed.

Retest
This vulnerability has been fixed by adding statement in require().

Bug ID # 25 [Won't Fix]

Large Number Literals

Vulnerability Type
Gas & Missing Best Practices

Severity
Gas

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.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L63-L88

Impacts
Having a large number literals in the code increases the gas usage of the contract during its deployment and when the functions are used or called from the contract.
It also makes the code harder to read and audit and increases the chances of introducing code errors.

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
It is recommended to use numbers in the form "35 * 1e7 * 1e18" or "35 * 1e25".
The numbers can also be represented by using underscores between them to make them more readable such as "35_00_00_000"

Retest

This won’t be fixed as it does not affect security.

Bug ID # 26 [Fixed]

Gas Optimization in Require Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The `require()` statement takes an input string to show errors if the validation fails.
The strings inside these functions that are longer than `32 bytes` require at least one additional MSTORE, along with additional overhead for computing memory offset and other parameters. For this purpose, having strings lesser than 32 bytes saves a significant amount of gas.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L188
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L191

Impacts
Having longer require strings than `32 bytes` cost a significant amount of gas.

Remediation
It is recommended to shorten the strings passed inside `require()` statements to fit under `32 bytes`. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Retest
Require strings are now using less than 32 bytes to optimize gas.

Bug ID # 27 [Won't Fix]

Use of SafeMath

Vulnerability Type
Gas Optimization

Severity
Gas

Description
SafeMath library is found to be used in the contract. This increases gas consumption more than traditional methods and validations if done manually.
Also, Solidity `0.8.0` and above includes checked arithmetic operations by default, rendering SafeMath unnecessary.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L17
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/oracle/JacksAVAXTwapOracle.sol#L15
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L63

Impacts:
This increases the gas usage of the contract.

Remediation
We do not recommend using the SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.
The compiler should be upgraded to `Solidity version 0.8.0+` which automatically checks for overflows and underflows.

Retest

Client's Comment: No changes for now, the current implementation is fine.

Bug ID # 28 [Fixed]

Multiplication/Division by 2 should use Bit-Shifting

Vulnerability Type
Gas Optimization

Severity
Gas

Description
In Solidity, the EVM (Ethereum Virtual Machine) executes operations in terms of gas consumption, where gas represents the computational cost of executing smart contract functions. Multiplication and division by two can be achieved using either traditional multiplication and division operations or bitwise left shift (<<) and right shift (>>) operations, respectively. However, using bit-shifting operations is more gas-efficient than using traditional multiplication and division operations.

- x * 2 can be replaced with x << 1.
- x / 2 can be replaced with x >> 1.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L108
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L187
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L239
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L511

Impacts
Gas consumption directly affects the cost of executing smart contracts on the Ethereum blockchain. Using bit-shifting operations for multiplication and division by two reduces the gas cost from 5 to 3, leading to more cost-effective and efficient smart contract execution.

This optimization is particularly relevant in scenarios where gas efficiency is crucial, such as high-frequency operations or resource-intensive contracts.

Remediation
It is recommended to use left and right shift instead of multiplying and dividing by 2 to save some gas.

Retest
This is fixed by implementing bit shifting to optimize gas.

Bug ID # 29 [Won't Fix]

Code Optimization by using max and min

Vulnerability Type
Gas Optimization

Severity
Gas

Description:
In Solidity contract code, optimizing expressions involving powers of 2, such as `2**32`, by using the built-in `type(uint32).max` and `type(uint32).min` constants can lead to improved code readability and gas efficiency. The original code utilizes `2**32` to calculate the maximum storage capacity of a uint32 data type, but this expression can be replaced with more expressive and gas-efficient alternatives.

Affected Code:

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L60

Impacts:
Using 2**32 in code can hinder readability and result in higher gas costs. Gas consumption is a crucial factor in determining the cost of executing smart contracts on the Ethereum blockchain. Optimizing such expressions contributes to more concise and understandable code, while also potentially reducing the gas fees associated with contract deployment and execution.

Remediation:
To optimize code involving powers of 2, developers should replace expressions like 2**32 with `type(uint32).max` for maximum values and type(uint32).min for minimum values. It is essential to note that `type(uint32).max` is equivalent to `2**32 - 1`.

Retest

Client's Comment: 2^254 is a large number and does not directly correspond to any built-in type's maximum value.

Bug ID # 30 [Won't Fix]

State Variable Can Be Marked As Constants

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract has defined state variables whose values are never modified throughout the contract.
The variables whose values never change should be marked as constant to save `gas`.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolRegistry.sol#L29

Impacts
Not marking unchanging state variables as constant in the contract can waste gas.

Remediation
Make sure that the values stored in the variables flagged above do not change throughout the contract. If this is the case, then consider setting these variables as `constant`.

Retest
Client's Comment: No changes are needed.

Bug ID # 31 [Fixed]

Splitting Require Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Require statements when combined using operators in a single statement usually lead to a larger deployment gas cost but with each runtime calls, the whole thing ends up being cheaper by some gas units.

Affected Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L156
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L386-L389
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L520-L523
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L625-L631
- https://github.com/stable-jack/vhttps://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Market.sol#L625-L631-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L135
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/common/math/LogExpMath.sol#L146
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L612-L615

Impacts

The multiple conditions in one `require` statement combine require statements in a single line, increasing deployment costs and hindering code readability.

Remediation
It is recommended to separate the `require` statements with one statement/validation per line.

Retest
This has been fixed properly.

Bug ID# 32 [Won't Fix]

Gas Optimization in Increments

Vulnerability Type
Gas optimization

Severity
Gas

Description
The contract uses two for loops**,** which use post increments for the variable `"i"`.
The contract can save some gas by changing this to `++i.`
`++i` costs less gas compared to `i++` or `i += 1` for unsigned integers. In **i**`++,` the compiler has to create a temporary variable to store the initial value. This is not the case with `++i` in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Vulnerable Code

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/ChainlinkPriceOracle.sol#L55
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/misc/PlatformFeeDistributor.sol#L80
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/misc/PlatformFeeDistributor.sol#L100
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/misc/PlatformFeeDistributor.sol#186
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/price-oracle/twap/ChainlinkTwapOracleV3.sol#L138
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolRegistry.sol#L39
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolRegistry.sol#L48

- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/helpers/PlatformFeeSpliter.sol#L132
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L81
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L101
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePoolSplitter.sol#L193
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/gateways/JackGateway.sol#L199
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L446
- https://github.com/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/rebalance-pool/RebalancePool.sol#L653

Impacts

Using `i++` instead of `++i` costs the contract deployment around 600 more gas units.

Remediation

It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest

Client's Comment: No changes are needed.

Bug ID # 33 [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/stable-jack/v1-jack-contracts/blob/995c6a69efd89f0800ce6bfb515c8149c47a7ee3/contracts/jack/Treasury.sol#L502

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
Fixed by removing the function.

6. Disclosure

The Reports provided by CredShields is 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.