QuillAudits

Audit Report
July, 2022

For

Asva Labs

Table of Content

Executive Summary

Project Name	Asva IDO Launchpad
Overview	The IDO Launchpad is a platform that aims to raise funds for blockchain products for the public. The users can participate in the projects based on the tokens staked. The tokens here are all ERC20 based, and the platform will be launched on BSC, Polygon and Avalanche blockchains.
Timeline	29th June, 2022 to 19th July, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyze Asava's codebase for quality, security, and correctness. *https://github.com/Asva-Labs-HQ/Asva-IDO-SC/tree/claim/contracts*
Commit hash	f427a8998bde5d91a639a6075693d3d2bda81906
Fixed In Commit hash	*https://github.com/Asva-Labs-HQ/Asva-IDO-SC/tree/claim/contracts* 8a65f75da57537901202e0ba497064456ba46c42

Extra Review: *https://github.com/Asva-Labs-HQ/Asva-IDO-SC/tree/claimv2/contracts*
Note: Extra Review was conducted on 27 June 2023 to Verify the Changes made in contract to verify High Severity Issue "Unhandled case for different decimal tokens".

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	1
Resolved Issues	1	1	1	2

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

High Severity Issues

A1. Unhandled case for different decimal tokens

Line	Contract: TierIDOPool.sol | Function: buy()
172	*uint256 currencyAmount = amount.mul(price).div(10**uint256(currency.decimals()));*

Description

The contracts have an unhandled case where if the decimal value of the token used to purchase the IDOToken does not match, the desired output is not achieved.

Scenario

Let's assume IDOToken is 18 decimal and the purchase token is USDT/USDC. There is a logic in the contract which calculates the amount that the user has to pay and saves the same amount into the sale mapping. When the user comes back to claim, the contract fetches the same value from the sales mapping for the distribution of IDOToken. The problem is that if the purchase token amount is in 6 decimal places, it ultimately converts the IDOToken into 6 decimal places and sends them in that form.

Assumptions/Restrictions

The bug will not be produced if the IDOToken and the purchase token have the same decimal places. This causes the contract to be bound to use the same decimal tokens in the presale. In any case where there is a difference in the decimals of both these tokens, the result will be undesirable.

Status

Resolved

Medium Severity Issues

A2. Potential array out of bounds error

Line	Contract: Claimer.sol | Function: Constructor()

38	*for (uint i = 0; i < times.length; i++) {
	require(percents[i] > 0, 'Claimer: 0% is not allowed');
	require(times[i] > 0, 'Claimer: time must specified');
	claims.push(Claim(times[i], percents[i]));
	totalPercent += percents[i];
	}*

Description

Inside the Claimer's constructor, there is a loop that checks the values contained inside arrays times and percents. The loop checks both the arrays, while running till times.length. If the lengths are different, we may have an array out of bound issue. In case the iteration number is less, one of the arrays may not be traversed completely.

Remediation

Write a require statement that makes sure that the arrays are of equal length.

Status

Resolved

Low Severity Issues

A3. Missing zero address check

Line	Contract: TierIDOPool.sol | Function: setPlatformTokenAddress()
251	*function setPlatformTokenAddress(address _platformToken) external onlyOwner returns (bool) {* *platformToken = IERC20(_platformToken);* *return true;* *}*

Line	Contract: TierIDOPool.sol | Function: addToPoolWhiteList()
293	*function addToPoolWhiteList(address[] memory _users, uint8 _tier) external onlyOwner returns (bool) {* *for (uint256 i = 0; i < _users.length; i++) {* *if (!poolWhiteList[_users[i]]) {* *poolWhiteList[_users[i]] = true;* *addressBelongsToTier[_users[i]] = _tier;* *listWhitelists.push(address(_users[i]));* *}* *}* *return true;* *}*

Line	Contract: AsavaPoolFactory.sol & AsavaClaimFactory.sol | Function: constructor()
87	*constructor(address _asvaInfoAddress, address _platformToken) public {* *AsvaInfo = AsvaInvestmentsInfo(_asvaInfoAddress);* *platformToken = IERC20(_platformToken);* *}*

Description

The function does not check whether the provided address is a non-zero or a zero address.

Remediation

There should be a simple require check to ensure that the input address is non-zero.

Status

Resolved

Informational Issues

A4. Wrong NatSpec description

Line	Contract: AsvaInvestmentsInfo.sol

102	* @dev To get claim added bool
	* Requirements:
	* - asvaId must be a valid id
	*/
	function getIDOAddress(address idoContract) external view returns (bool) {
	return alreadyAdded[idoContract];
	}

Line	Contract: AsvaInvestmentsInfo.sol

110	* @dev To get Claim contract address by asvaId
	*
	* Requirements:
	* - asvaId must be a valid id
	*/
	function getClaimAddressByProjectID(uint256 claimId) external view returns (address) {
	return claimddressByProjectID[claimId];
	}

Description

The NatSpec descriptions given for these functions do not match them.

Status

Acknowledged

A5. Spelling mistakes

Line	Contract: AsvaInvestmentsInfo.sol
18	*mapping(uint256 => address) public claimddressByProjectID;*

Description

A spelling mistake was found in the AsvaInvestmentsInfo.sol contract.

Remediation

Change the spelling to claimAddressByProjectID.

Status

Resolved

A6. Solidity Style Guide: Function Order

Description

According to docs.soliditylang.org, there is a specific order of functions that people are suggested to follow. In the Claimer.sol contract, the view functions in the external group should be at the end. Other than that, AsavaClaimFactory.sol has an internal function above an external function. These two should be switched.

Status

Acknowledged

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Asva codebase. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the End, The Asva Labs Team resolved Almost all Issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Asva Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Asva platform's Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
July, 2022

For

Asva Labs

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com