BlockApex

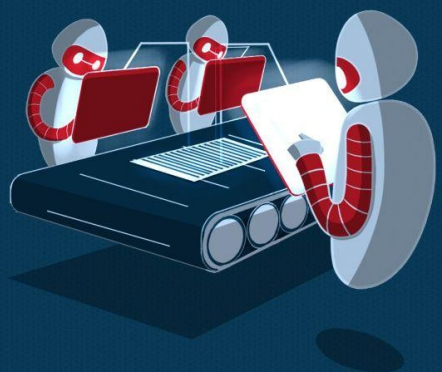
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



Powered by XORD

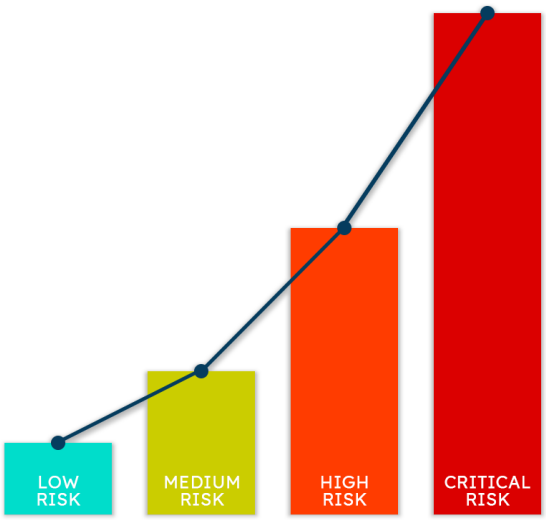
PREFACE

Objectives

The purpose of this document is to highlight the identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below.

The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings



CRITICAL RISK ××××	Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.
HIGH RISK ×××	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.
MEDIUM RISK ××	Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.
LOW RISK ×	Lowest-level vulnerabilities, informational errors, violating code styles/practices usually can't affect smart contract execution; hence they can be ignored.

TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Arable (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which took place on 9th June 2022 .

Name
Arable-MVP V1
Auditor
Kaif Ahmed | Mohammad Memon
Platform
Ethereum/Solidity
Type of review
Manual Code Review | Automated Code Review
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Git repository
https://github.com/ArableProtocol/arable-contracts-o cial/tree/main/contracts/mvp_v1
White paper/ Documentation
Readme.md (Github)
Document log
Initial Audit: 14th June 2022
Final Audit:

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulnerabilities**. Some specific checks are as follows:

Code review		**Functional review**
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset’s integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Arable is the first decentralized synthetic farming protocol, making multi-chain farming accessible and convenient for everyone.

System Architecture

Arable’s goal is to provide an environment where yield farmers can access all the crypto farming assets on a single chain, where the users still receive the same rewards as they would on native chains. At Arable multichain farming is done in a single app on a single chain and it is easy to keep an overview of all your farming activities. The complexity of managing multiple farms and employing involved strategies is greatly reduced. Synthetic assets are essentially tokenized derivatives to support people to buy or sell specific assets on native assets’ price.

Methodology & Scope

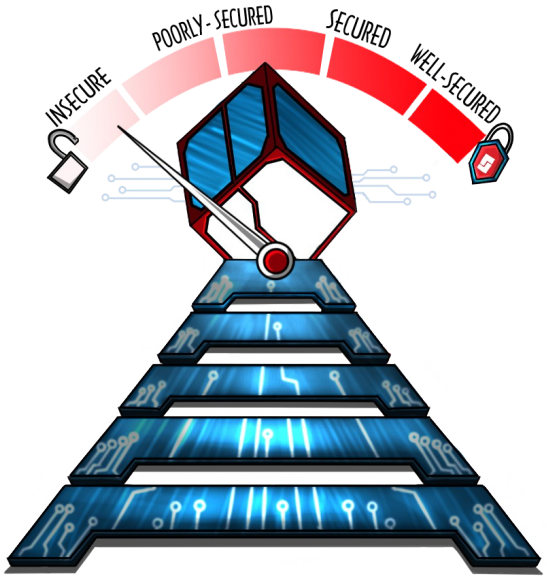
The code came to us in the form of a zip, containing a truffle directory, the contract and the tests. Initially, we ran the contract and tested the functionality of all the functions manually. After that, we moved to Foundry to try all kinds of scenarios. After all the logical and functional testing, we moved to code optimizations and solidity design patterns to ensure consistency and readability.

AUDIT REPORT

Executive Summary

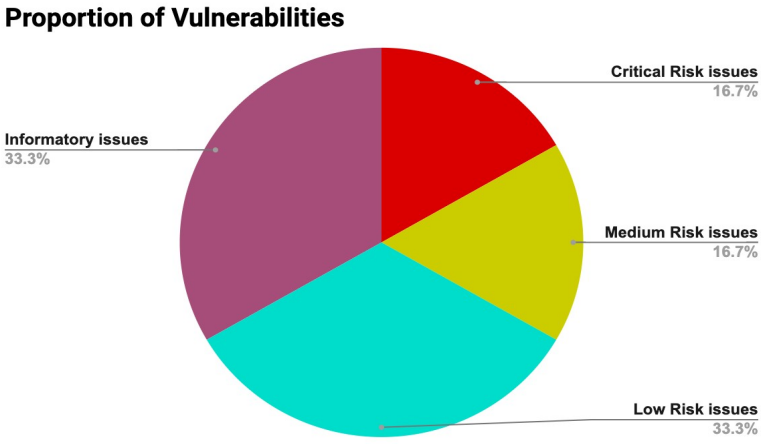
The analysis indicates that all of the functionalities in the contracts audited are **not working properly**.

Our team performed a technique called “Filtered Audit”, where the contract was separately audited by two individuals. After their thorough and rigorous process of manual testing, an automated review was carried out using Surya. All the flags raised were manually reviewed and re-tested.



Our team found:

# of issues	Severity of the risk
2	Critical Risk issue(s
0	High Risk issue(s
2	Medium Risk issue(s
4	Low Risk issue(s
4	Informatory issue(s)



Findings

#	Findings	Risk	Status
1.	Potential reentrancy attack opportunity	Critical	Acknowledge
2.	Permanent roadblock in logical flow	Critical	Acknowledge
3.	No checks for duplicate token entry in ArableFarming contract	Medium	Fixed
4.	No checks for duplicate token entry in ArableFeeCollector contract	Medium	Fixed
5.	Ratio-based return of different stable-tokens on withdrawal	Low	Fixed
6.	Missing zero address checks	Low	Fixed
7.	Function visibility can be changed	Low	Fixed
8.	Missing checks in setters	Low	Fixed
9.	Implemented external functions do not follow style guide	Informatory	Fixed
10.	Function signature style mismatch	Informatory	Fixed
11.	Missing Netspac	Informatory	Fixed
12.	Order of functions is not based on instructions from the style guide	Informatory	Fixed

Critical risk issues

1. **Potential reentrancy attack opportunity**

   Contract: StabilityFund.sol

   Description: There is a potential reentrancy attack possible in the `withdraw()` function. Since the contract is designed to use stable-tokens, any stable-token based on **ERC777 (AMPL)** may respond with a contract hook to execute external code.

```
for (uint256 index = 0; index < stableTokens.length; index++) {
            IERC20 token = stableTokens[index];
            uint256 tAmount = (token.balanceOf(address(this)) * amount) /
totalLP;
            console.log("%s -  - %s", address(token), tAmount);
            if (amount > 0) {
                token.safeTransfer(msg.sender, tAmount);
                emit WithdrawToken(msg.sender, token, tAmount);
            }
        }
        _burn(msg.sender, amount);
```

   Remedy: Withdraw function should have a non-entrant check or it should call burn before transferring the amount to contract.

2. Permanent roadblock in logical flow

Contract: ArableFarming.sol

Description: In the ArableFarming.sol contract, there is a function `stake()`. This function transfers fees to the ArableFeeCollector before transferring the amount to ArableFarming. This execution will fail because the `payFeeFor()` function of ArableCollector uses msg.sender to deduct fees. Considering the contract has no funds of the specific token and the user is staking for the first time, there will be no funds in the farming contract to transfer in the ArableCollector.

```
uint256 fees = payFeesFor(stakingToken, amount, msg.sender,
ArableFees.Model.SETUP_FARM);

      IERC20(stakingTokens[farmId]).transferFrom(msg.sender, address(this),
amount);
```

Remedy: Mentioned below are some solutions that we identified.

1. Admin needs to top-up the token amount into the contract.
2. `FeeCollector()` function can use the address passed in the parameter to deduct the fee directly from the user instead of using msg.sender.

Medium risk issues

3. No checks for duplicate token entry

Contract: ArableFarming.sol

Description: ArableFarming has a function called `registerFarm()`. This function is used to whitelist staking tokens. The function does not check whether the given input address is already whitelisted or not.

```
function registerFarm(address stakingToken) public override onlyOwner returns
(uint256 farmId) {
        require(stakingToken != address(0x0), "stakingToken should be set");
        stakingTokens.push(stakingToken);
        farmId = stakingTokens.length - 1;
        emit RegisterStakingPool(farmId, stakingToken);
        return farmId;
    }
```

Remedy: Check for the already whitelisted staking tokens. It will also limit the `bulkRegistrationFarm()` function to send the same tokens in the array parameter.

4. No checks for duplicate token entry

Contract: ArableFeeCollector.sol

Description: ArableFeeCollector has a function called `setRewardTokens()`. This function receives an array of tokens and does not check for the same tokens in the array and writes them into the reward token mapping.

```
function setRewardTokens(address[] memory _rewardTokens) public override
onlyOwner {
        deleteRewardTokens();
        for (uint256 i = 0; i < _rewardTokens.length; i++) {
            address rewardToken = _rewardTokens[i];
            rewardTokens.push(rewardToken);
            _isRewardToken[rewardToken] = true;
        }
        emit SetRewardTokens(_rewardTokens);
    }
```

Remedy: There should be a check to identify duplicate tokens inside the array.

Low risk issues

5. Ratio-based return of different stable-tokens on withdrawal

Contract: StabilityFund.sol

Description: Assuming contract is minting the same LP tokens for every stable-token whitelisted in the contract. Consider a scenario where the user deposits 100 USDT and the contract has 10 different stable-tokens already deposited. When a user will come to withdraw his 100 USDT, he will receive the 10% of every stable-token that the contract holds.

6. Missing zero address checks

Contract: ArableExchange.sol,
ArableFarming.sol,
ArableFeeCollector.sol,
ArableLiquidation.sol,
ArableManager.sol,
ArableOracle.sol

Description: These contracts use the `initialize()` function to set the initial values of the contract. Since the `initializer()` functionality has been inherited from @Openzeppelin/initializable.sol which means the function is only callable once in a lifetime. There should be a zero address check in all the `initialize()` functions.

7. Function visibility can be changed

Contract: ArableCollateral.sol

Description: ArableCollateral contract contains two main functions `depositCollateral()` and `withdrawCollateral()`. The visibility of both these functions is public, but both the functions are not being called from anywhere else.

Remedy: It is recommended to make these functions external to save gas for the external caller.

8. Missing checks in setters

Contract: ArableFeeCollector.sol,
ArableCollateral.sol,
ArableLiquidation.sol

Description: All the functions used to set or unset values, even if they are called by onlyAdmin, should have a zero value/address check placed to save the admin from redundant calling. Following are the functions that need to be checked before the admin sets the values.

- setAllowedProvider()
- unsetAllowedProvider()
- setAllProviders()
- unsetAllProviders()
- setRewardTokens()
- setAddressRegistry()
- depositCollateral()
- setLiquidationRate()
- setImmediateLiquidationRate()
- setLiquidationDelay()
- setLiquidationPenalty()

Informatory issues and Optimization

9. Implemented external functions do not follow style guide

Contract: ArableCollateral.sol,
ArableExchange.sol,
ArableFarming.sol,
ArableFeeCollector.sol,
ArableLiquidation.sol,
ArableManager.sol,
ArableOracle.sol,
ArableSynth.sol

Description: All these contracts contain the `pause()` and `unpause()` functionality inherited from @Openzeppelin/PausableUpgradable.sol. Both these functions are external and should be implemented right below the constructor.

10. Function signature style mismatch

Contract: ArableCollateral.sol

Description: There is a function in the contract named `_userCollateralBalance()`. In this function, the visibility is public. Only internal functions start with the underscore. Follow the Solidity style guide or make the function internal.

```
function _userCollateralBalance(address user, address token) public view
returns (uint256) {
        return _individualCollateral[user][token];
    }
```

11. Missing Netspac

Contract: ArableExchange.sol

Description: Contract has two main functions `swapSynths()` and `convertFeesToUsd()`. Both these functions are missing the netspac.

12. Order of functions is not based on instructions from the style guide

Description: As stated in the Solidity style guide, the functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- Private

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.
