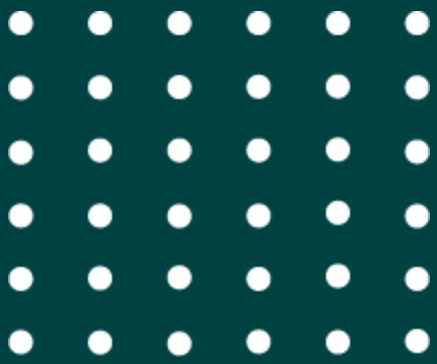
Alethea

Bonding Curve

SMART CONTRACT AUDIT REPORT



IMMUNEBYTES

08 January 2024

Introduction

About Alethea AI

Alethea AI is building a decentralized protocol that will enable the creation of interactive and intelligent NFTs (iNFTs). As originators of the iNFT standard, Alethea AI is on the cutting edge of embedding AI animation, interaction, and generative AI capabilities into NFTs. Anyone can use the iNFT protocol to Create, Train, and Earn from their iNFTs in the world's first Intelligent Metaverse known as Noah's Ark.

Visit https://alethea.ai/ to know more about it.

About ImmuneBytes

ImmuneBytes is a security start-up that provides professional services in the blockchain space. The team has hands-on experience conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and understand DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, and dydx.

The team has secured 205+ blockchain projects by providing security services on different frameworks. The ImmuneBytes team helps start-ups with detailed system analysis, ensuring security and managing the overall project.

Visit http://immunebytes.com/ to learn more about the services.

Project Overview

Project Name:
AletheaAI - AIProtocol - Bonding Curve

Project Scope:
1. ai-protocol-contracts/tree/poc/f_tech_curve/contracts/bonding_curves
2. ai-protocol-contracts/blob/release/2.8-base/contracts/token/OpAliERC20v2.sol

Blockchain:
Ethereum (EVM Compatible Chains)

Language:
Solidity

Documentation Details

The team has provided the following doc for audit:

1. https://github.com/AletheaAI/ai-protocol-contracts/blob/master/README.md

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

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

Audit Process & Methodology

The ImmuneBytes team has performed thorough testing of the project, starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is a structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract to find potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors, which includes -

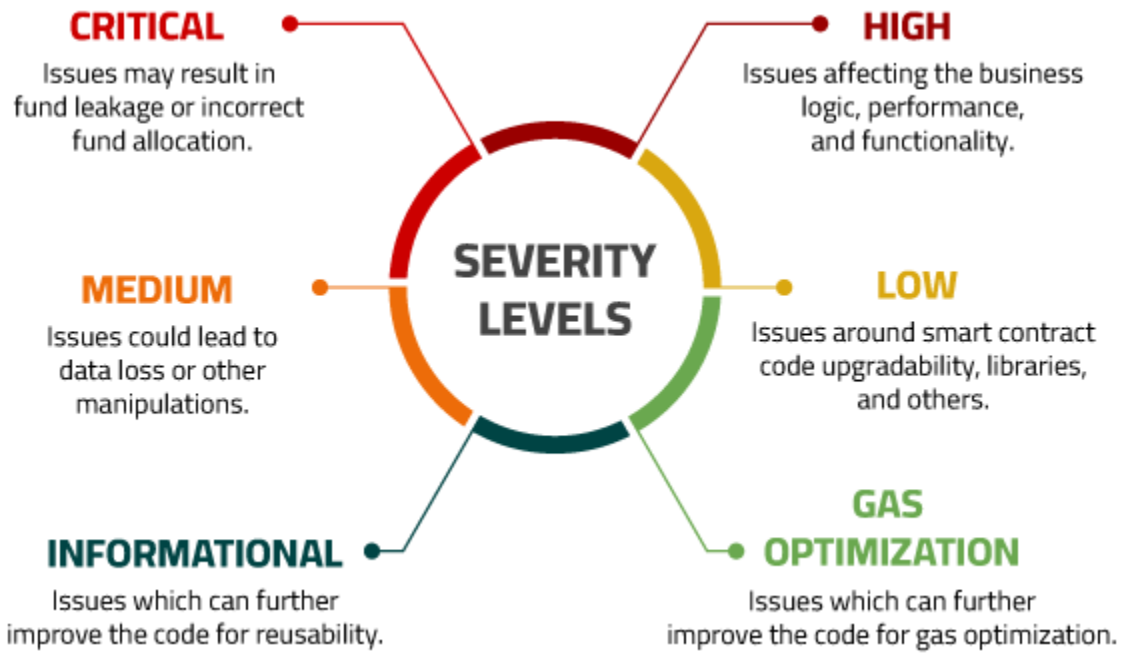
1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

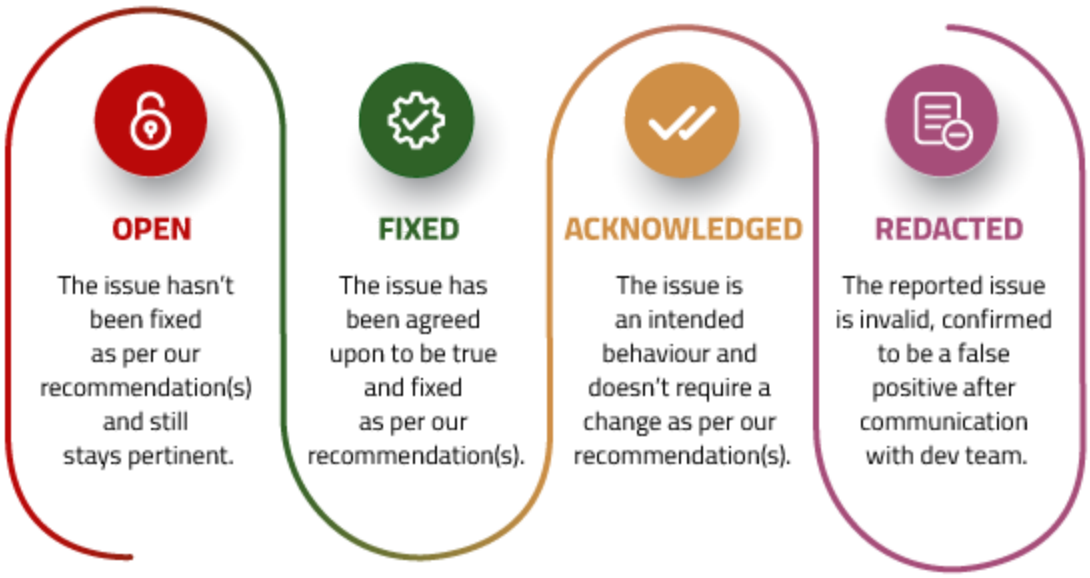
Project Name	Alethea
Platform	EVM
Languages	Solidity
GitHub Link	https://github.com/AletheaAI/ai-protocol-contracts

Security Level References

Every issue in this report was assigned a severity level from the following:



Severity Status References



Severity Status

Issues	Critical	High	Medium	Low	Informational	Gas
Open	-	-	-	1	-	-
Fixed	-	-	1	2	-	-
Redacted	-	-	-	-	-	-
Acknowledged	-	-	-	-	1	-

Findings

#	Findings	Severity	Status
1	Insufficient Upper Bounds Check on Fee Percentage	**Medium**	**Fixed**
2	Inconsistent Interdependency Between Variables	**Low**	**Fixed**
3	Deprecated Usage of Native Transfers	**Low**	**Partially Fixed**
4	Unsafe Method For ERC20 Token Transfer	**Low**	**Acknowledged**
5	Unsafe External Calls Can Allow Reentrancy	**Informational**	**Closed**

Critical Severity Issues

No issues were found.

High Severity Issues

No issues were found.

Medium severity issues

1. **Issue:**
   Insufficient Upper Bounds Check on Fee Percentage

   Description:

   The setProtocolFeePercent and setHoldersFeePercent functions lack a validation check for an upper limit on the fee percentage. This oversight allows setting the fee percentage to any value, potentially exceeding 100%.

   Impact:

   Setting excessively high fees, such as 100% or more, could lead to operational issues and user dissatisfaction. It may deter users from interacting with the contract and result in transaction failures if the fee exceeds the transaction amount.

   Code-affected:

```
/**
 * @inheritdoc SharesFactory
 */
function setProtocolFeePercent(uint64 _feePercent) public {
        // verify the access permission
        require(isSenderInRole(ROLE_PROTOCOL_FEE_MANAGER), "access denied");
        // verify state change doesn't result into the discrepancy
        require(_feePercent == 0 || protocolFeeDestination != address(0), "protocolFeeDestination must
be set first");

        // update contract's state
        protocolFeePercent = _feePercent;

        // emit an event
```

```
        emit ProtocolFeeUpdated(protocolFeeDestination, _feePercent, holdersFeePercent,
subjectFeePercent);
}

/**
 * @inheritdoc SharesFactory
 */
function setHoldersFeePercent(uint64 _feePercent) public {
        // verify the access permission
        require(isSenderInRole(ROLE_HOLDERS_FEE_MANAGER), "access denied");

        // update contract's state
        holdersFeePercent = _feePercent;

        // emit an event
        emit ProtocolFeeUpdated(protocolFeeDestination, protocolFeePercent, _feePercent,
subjectFeePercent);
}
```

Recommendation:

Implement an upper limit validation in the setProtocolFeePercent function. A typical practice is to ensure the fee does not exceed 100%, with a common representation being in basis points (where 10000 basis points = 100%). For example:
require(_feePercent <= 10000, "Fee percent too high");

Status: Fixed

Low severity issues

1. **Issue**: Inconsistent Interdependency Between Variables

   Description:

   The setProtocolFeeDestination function requires that if a non-zero fee destination is set, the protocol fee percent must not be zero. Conversely, the setProtocolFeePercent function requires a non-zero fee destination address to be set before allowing a non-zero fee percentage. This creates a dependency between setting the fee destination and the fee percentage.

   Impact:

   This interdependency restricts the order of operations, potentially limiting flexibility in managing fee settings. It could prevent the independent update of either parameter without first adjusting the other, which may not always be practical or desirable.

   Code-affected:

```
/**
 * @inheritdoc SharesFactory
 */
function setProtocolFeeDestination(address _feeDestination) public {
        // verify the access permission
        require(isSenderInRole(ROLE_PROTOCOL_FEE_MANAGER), "access denied");
        // verify state change doesn't result into the discrepancy
        require(_feeDestination != address(0) || protocolFeePercent == 0, "protocolFeePercent must be set
to zero first");

        // update contract's state
        protocolFeeDestination = _feeDestination;

        // emit an event
        emit ProtocolFeeUpdated(_feeDestination, protocolFeePercent, holdersFeePercent,
subjectFeePercent);
}

/**
 * @inheritdoc SharesFactory
 */
function setProtocolFeePercent(uint64 _feePercent) public {
```

```
        // verify the access permission
        require(isSenderInRole(ROLE_PROTOCOL_FEE_MANAGER), "access denied");
        // verify state change doesn't result into the discrepancy
        require(_feePercent == 0 || protocolFeeDestination != address(0), "protocolFeeDestination must be
set first");

        // update contract's state
        protocolFeePercent = _feePercent;

        // emit an event
        emit ProtocolFeeUpdated(protocolFeeDestination, _feePercent, holdersFeePercent,
subjectFeePercent);
}
```

Recommendation:

Consider revising the validation logic in these functions to allow more flexibility. This could involve permitting the independent setting of either the fee percentage or the fee destination, while still ensuring that both parameters are eventually set to valid values before fees are collected.

Status: **Fixed**

2. **Issue**: Deprecated Usage of Native Transfers

Description:

The smart contract complex for bonding curves in the alethea ai-protocol uses the transfer method to transfer ethers to and from the contracts. This pattern of funds transfer has been deprecated in favor of the call method due to the gas limitation potentially resulting in transaction failures.

```
/**
 * @inheritdoc HoldersRewardsDistributor
 */
function claimTheReward() public {
    uint256 claimableAmount = pendingReward(msg.sender);
    require(claimableAmount > 0, "Nothing to claim");

    UserInfo storage userDetail = userInfo[msg.sender];
    // update state variable
    userDetail.unclaimedAmount = 0;
    userDetail.claimedAmount += claimableAmount;
```

```
    userDetail.rewardDebt = (userDetail.shares * accRewardPerShare) / 1e18;

    // transfer reward
    if(paymentToken == address(0)) {
        Transfers.transfer(payable(msg.sender), claimableAmount);
    }
    else {
        require(ERC20(paymentToken).transfer(msg.sender, claimableAmount));
    }

    // emit an event
    emit RewardClaimed(msg.sender, claimableAmount);
}
```

Recommendation:

It is highly recommended (refer to Open Zeppelin case study) to utilize the call method for transfer all the while ensuring that reentrancy is restricted using the CEI pattern and mutexes.

Status: Partially Fixed

Developer Response:
The finding is correct. However, we take the risk of reentrancies here more seriously than the risk of failed transfers. We have also mitigated this issue by increasing the gas limit from 2300 to 4900.

3. **Issue**: Unsafe Method For ERC20 Token Transfer

Description:

The smart contract complex for bonding curves in the alethea ai-protocol uses the transfer and transferFrom methods to transfer the token currency. Using transfer has been proven to multiple vulnerabilities that are exploited due to unexpected return values e.g. since the tokens' return values may or may not be handled similarly e.g. the USDT token and few more tokens like that don't returning any boolean for successful transaction or tokens like ERC777 that offer hooks become gateways to vulnerable control flows.

Code Affected:

```
        // do the required ERC20 payment token price transfer
        require(
                // do not try to transfer zero price
                price == 0 || paymentToken.transferFrom(msg.sender, address(this),
```

```
price),
                "payment failed"
        );
```

Recommendation:

It is highly recommended to utilize the safeERC20 library from openzeppelin for ERC20 token transfers to ensure robustness and compliance with industry best practices.

Status: Acknowledged

Developer Response:
This can be indeed an issue when copy pasting the code into other projects which use other ERC20 implementations. We do distinguish, however, from the very beginning, our implementation (ALI ERC20 token) and all other implementations and use the SafeERC20 library only for "foreign" implementations.

Informational

1. **Issue**: Unsafe External Calls Can Allow Reentrancy

Description:

Several functions throughout the codebase implement the .call method to interact with external and internal smart contracts in the AIProtocol ecosystem. When manipulating the states of variables within a smart contract, these calls should be safe during the execution for any reentrancy, which can be ensured by following three renowned best practices in the industry. Namely, non-reentrant libraries that utilize mutexes, injecting checks-effects-interactions patterns and pull-over-push design patterns.

Code-affected:

```
/**
 * @inheritdoc TradeableShares
 */
function buyShares(uint256 amount) public payable {
        // delegate to `buySharesTo`
        buySharesTo(amount, msg.sender);
}

/**
```

```
 * @inheritdoc TradeableShares
 */
function sellShares(uint256 amount) public {
        // delegate to `sellSharesTo`
        sellSharesTo(amount, payable(msg.sender));
}
```

Recommendation:

It's highly recommended that the functions with state manipulation containing external calls be executed under mutexes offered by the Open Zeppelin's implementation of the nonReentrant library.

Status: Closed

Developer Response:
According to OZ documentation, the ReentrancyGuard is a mitigation instrument. Therefore our approach is to try avoiding using it not to fall into a false impression of reentrancy safety. We try to stick to CEI pattern wherever is possible, or, if it is not possible, limit the gas supplied so that no storage modification can be made in the external code.

Gas Optimization

No issues were found.

Concluding Remarks

During a comprehensive security code review of the Bonding Curve smart contracts, the expert team at ImmuneBytes found a few security vulnerabilities. Our auditors suggest that the developers should resolve issues before moving up with production. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes' audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process; therefore, running a bug bounty program complementing this audit is strongly recommended.

Our team does not endorse the Alethea platform or its product, nor is this audit investment advice.

Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for code refactoring by the team on critical issues.

IMMUNEBYTES

AUDITS