Alethea AI Protocol: Bonding Curves Smart Contract Audit

February 29, 2024

Table of Contents

Overview

This document describes the result of auditing for the several smart contracts of Alethea AI protocol. The auditing progress is performed against the contracts in `bonding_curves` folder and `OpAliERC20v2`, `WhitelabelNFT` contracts including deployed contracts, test files.

Project	**ai-protocol-contracts**

Smart Contracts	`bonding_curves`
	`OpAliERC20v2.sol`
	`WhitelabelNFT.sol`

Commit SHA	`a89c33a`
Used Tools	**Manual Review**

Scope

Smart Contracts of Repository

Smart Contracts	Review Status
`contracts/bonding_curves/`	
`AbstractShares.sol`	✓
`BondingCurve.sol`	✓
`FriendTechBondingCurve.sol`	✓
`SharesFactory.sol`	✓
`SharesFactoryV1.sol`	✓
`SharesSubjectLib.sol`	✓

TradeableShares.sol	✓
ETHShares.sol	✓
ERC20Shares.sol	✓
RewardSystem.sol (newly update)	✓
ProtocolFeeDistributorV1.sol	✓
HoldersRewardsDistributor.sol	✓
HoldersRewardsDistributorV1.sol	✓
HiveRegistryV1.sol	✓
contracts/token/	
AliERC20v2.sol	✓
OpAliERC20v2.sol	✓
WhitelabelNFT.sol	✓

Deployed Contracts

Address (Base Mainnet)	Review Status
Proxies	
Shares Factory 0x63aCBC42e466d29F271c102Bf97A18B52203b308	✓
Protocol Fee Distributor 0x6E1B9c06410c234A427ca10935C2438548c988bd	✓
Leaderboard (newly update) 0x608a7511d63b31255Cf035C82E6AA62396882f34	✓
Implementation	

Shares Factory 0x41AeEd8861e3BAa0AF5E5f8A60Ca4fABA3Abb58E	✓
Protocol Fee Distributor 0x9c22495aEf3104014C15274C80Ea5AFdE8280523	✓
Leaderboard (newly update) 0xde04862Ba04E8641e32cD9Ab57F6E816928301E0	✓
ETH Shares 0x8f543a0DB0CF54395c9A1633d109d166C40EAF13	✓
Holders Reward Distributor 0x03F6c12a80F6738F13dD552e286549Eecb9477b5	✓
Hive Registry 0x9D30e52C1290Ee3348be1487127156a1136A57C7	✓
Non-Upgradeable	
DPT NFT 0x303D1e1F43fEF1fB8EAb940D9c11A203281c5211	✓
ALI Token 0x97c806e7665d3AFd84A8Fe1837921403D59F3Dcc	✓

Risk Classification of Findings

- **High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).**
- **Medium: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.**
- **Low: Assets are not at risk: state handling, function incorrect as to spec, issues with comments.**
- **Info: Code style, clarity, syntax, versioning, off-chain monitoring events, etc.**
- **Gas: Re-writing Solidity code to accomplish the same business logic while consuming fewer gas.**

Summary of Findings

Severity	Count
High	0
Medium	1
Low	3
Info	2
Gas	5
Total	11

Finding Details

Medium

[M-01] Users may pay funds more than their expected reasonable amount since there is no slippage checking in `__buySharesTo` function.

Links to Affected Code

`bonding_curves/ETHShares.sol#156-185`

`bonding_curves/ERC20Shares.sol#L172-L201`

Vulnerability Details

`ERC20Shares` and `ETHShares` contract has `__buySharesTo` function which is responsible for buying shares for account. But there is no slippage checking in these functions.
I think when an account tries to buy shares, amount of funds (ETH/ERC20) to be paid can be affected by the total supply.

This means that paying funds amount can exceed the user's expected reasonable amount if there are front-running transactions for buying share. Of course, about `ETHShares`, since user would pay ETH according to pre-calculated by interacting contract, there is nothing special loss of fund. But paid ETH is less than calculated amount during execution due to front-running, user's buying action would be reverted. I think this issue should be considered based on expected frequency of buy/sell shares actions and scalability for ERC20. About `sellSharesTo` function, similar case is possible.

Recommended Mitigation

Consider adding slippage checking to `ERC20Shares` and logic that protecting unexpected revert in `ETHShares` when buy/sell shares.

Low

[L-01] Missing check for zero address

Links to Affected Code

`bonding_curves/ETHShares.sol#L156-L185`

`bonding_curves/ETHShares.sol#L190-L219`

Vulnerability Details

When buy/sell actions are executed, it should be checked that `beneficiary` is not zero address.

```
function __buySharesTo(uint256 amount, address beneficiary) private {
    // cache the supply value
    uint256 supply = getSharesSupply();

    // update the balances (note: security checks are below)
    sharesBalances[beneficiary] += amount;

… …

function sellSharesTo(uint256 amount, address payable beneficiary) public {
    // verify the amount vs total supply
    uint256 supply = getSharesSupply();
    require(supply > amount, "cannot sell the last share");
```

```
        … …

        // price cannot be zero since the last share cannot be sold
        // if the price transfer fails, we do fail
        // note: if any of the fees failed to transfer, they are sent to the
seller
        beneficiary.transfer1(price - protocolFee - holdersFee - subjectFee);
```

Recommended Mitigation

Check if that `beneficiary` is not zero address, else revert.

[L-02] Missing reentrancy guard for buy/sell shares actions in `ETHShares` contract.

Links to Affected Code

`bonding_curves/ETHShares.sol#L142-L148`

`bonding_curves/ETHShares.sol#L190-L219`

Vulnerability Details

When buy/sell actions are executed, it can be reentered by the external contract. Currently, there is no special vulnerability. However, it would be better to consider adding reentrancy guard to these actions for unexpected execution.

Recommended Mitigation

Consider adding reentrancy guard to actions of buy/sell shares in `ETHShares` contract.

[L-03] Incorrect validation for hive id in `linkAsset` function.

Links to Affected Code

`bonding_curves/HiveRegistryV1.sol#L305`

Vulnerability Details

In `linkAsset` function of `HiveRegistryV1.sol`, there is a validation statement that check passed `hiveId` is valid. However, it checks only if the passed `hiveId` is greater than 0, doesn't check that `hiveId` is less than length of hives.

```
require(_hiveId > 0, "invalid hiveId");
```

Recommended Mitigation

Consider updating the above statement as follow like line 396 in the same contract.

```
require(_hiveId > 0 && _hiveId < hives.length, "invalid hiveId");
```

Info

[I-01] Some mutable states of `AbstractShares` contract is described as immutable in their comments.

Links to Affected Code

`bonding_curves/AbstractShares.sol#L25-L32`

Details

Some states of `AbstractShares` contract are mutable but they are described as immutable in their comments.

```
/// @dev Protocol fee percent, immutable; maximum value: 10^18 (< 2^60)
uint64 private /*immutable*/ protocolFeePercent;
```

Recommended Mitigation

Consider updating comments.

[I-02] Calculated `hashType` is not compatible with EIP712.

Links to Affected Code

`bonding_curves/HiveRegistryV1.sol#L730-733`

Details

In `HiveRegistryV1.sol`, `hashStruct` of `RegisterDPTRequest` is calculated but it is not calculated compatibly with EIP712.

According to the EIP721, the `hashStruct` function is defined as:

`hashStruct(s : 𝕊) = keccak256(typeHash ‖ encodeData(s)) where typeHash = keccak256(encodeType(typeOf(s)))`

Here, the type of a `struct` is encoded as `name ‖ "(" ‖ member₁ ‖ "," ‖ member₂ ‖ "," ‖ … ‖ memberₙ ")"` where each member is written as `type ‖ " " ‖ name`.

However, in `HiveRegistryV1.sol`, some `typehash` is not calculated based on EIP712.

For example, `hashType` of `RegisterAsDPTRequest` struct should be calculated as follow:

```
keccak256("RegisterAsDPTRequest(SharesSubject dpt,address
dptHolder,uint256 validFromTimestamp,uint256
expiresAtTimestamp,uint256 nonce)SharesSubject(address
tokenAddress,uint256 tokenId)")
```

However, the calculated value is different from the above, and even from comments. Maybe there is no risk if the `hashtype` is matched well with dApp. But it would be better to make sure that compatible with EIP712 completely.

You can find the similar issue in `TypedStructLib.sol`.

Gas

[G-01] Use `>> 1` instead of `/ 2`.

Links to Affected Code

`bonding_curves/ETHShares.sol#L118-L121`

Details

Solidity's division operation includes a division-by-0 prevention which is bypassed using shifting. Eventually, overflow checks are never performed for shift operations as they are done for arithmetic operations. Instead, the result is always truncated.

```
        // shift the curve by -2
        return super.getPrice(supply, amount) / 2; // @audit gas-op use >> 1
```

Recommended Mitigation

Use `>> 1` instead of `/ 2`.

[G-02] OR in `if-`condition can be rewritten to two single if conditions

Links to Affected Code

`bonding_curves/ HoldersRewardsDistributorV1.sol#L159-L163`

Details

```
    function __accept(uint256 _feeAmount) private {
        // check the state can accept the changes
        if(_feeAmount == 0 || totalShares == 0) {
            return;
        }
```

To save gas, above codebase can be rewritten as:

```
    function __accept(uint256 _feeAmount) private {
        // check the state can accept the changes
        if(_feeAmount == 0) return;
        if( totalShares == 0) return;
```

Recommended Mitigation

Consider rewriting OR in `if-`condition to two single if conditions.

[G-03] Use `!= 0` instead of `> 0` for unsigned integer comparison

Links to Affected Code

bonding_curves/ETHShares.sol#L144

bonding_curves/HoldersRewardsDistributorV1.sol#L120

bonding_curves/HoldersRewardsDistributorV1.sol#L123

bonding_curves/HoldersRewardsDistributorV1.sol#L144

bonding_curves/ProtocolFeeDistributorV1.sol#L125

bonding_curves/ProtocolFeeDistributorV1.sol#L132

bonding_curves/ProtocolFeeDistributorV1.sol#L155

bonding_curves/ProtocolFeeDistributorV1.sol#L162

bonding_curves/ProtocolFeeDistributorV1.sol#L185

bonding_curves/SharesSubjectLib.sol#L163

Details

When dealing with unsigned integer types, comparisons with `!= 0` are cheaper than with `> 0` when solidity version is under `0.8.6`.

```
    function __sharesBought(address _buyer, uint256 _amountBought) private {
        UserInfo storage userDetail = userInfo[_buyer];
@>      if(userDetail.shares > 0) {
            // calculated pending reward if any
            uint256 pending = ((userDetail.shares * accRewardPerShare) / 1e18) -
userDetail.rewardDebt;
@>          if(pending > 0) {
                // update unclaimed amount
                userDetail.unclaimedAmount += pending;
            }
        }
```

Above code can be rewritten as follow:

```
    function __sharesBought(address _buyer, uint256 _amountBought) private {
        UserInfo storage userDetail = userInfo[_buyer];
@>      if(userDetail.shares != 0) {
            // calculated pending reward if any
            uint256 pending = ((userDetail.shares * accRewardPerShare) / 1e18) -
userDetail.rewardDebt;
@>          if(pending != 0) {
                // update unclaimed amount
                userDetail.unclaimedAmount += pending;
```

```
        }
    }
```

Recommended Mitigation

Use `!= 0` instead of `> 0` for unsigned integer comparison. Or update solidity version to at least `0.8.6` (Current version is `0.8.4`)

[G-04] Use inline assembly to increase the index of `for` loop. And use `++i` instead of `i++`.

Links to Affected Code

`bonding_curves/ProtocolFeeDistributorV1.sol#L129-L138`

`bonding_curves/ProtocolFeeDistributorV1.sol#L159-L167`

`bonding_curves/ProtocolFeeDistributorV1.sol#L192-L200`

Details

To increase index of for loop, use inline assembly and `++i` instead of `i++`.

```
        for(uint8 i = 0; i < recipients.length; i++) {
            allocatedAmount = amount * recipients[i].allocationPercent / 1e6;

            if(allocatedAmount > 0) {
                // transfer the ETH to the recipient
                Transfers.transfer(recipients[i].recipient, allocatedAmount);
                // emit an event
                emit ETHSent(recipients[i].recipient, allocatedAmount);
            }
        }
```

For example, above `for` loop can be rewritten as follow:

```
@>      for(uint8 i = 0; i < recipients.length;) {
            allocatedAmount = amount * recipients[i].allocationPercent / 1e6;

            if(allocatedAmount > 0) {
                // transfer the ETH to the recipient
                Transfers.transfer(recipients[i].recipient, allocatedAmount);
                // emit an event
```

```
                emit ETHSent(recipients[i].recipient, allocatedAmount);
            }

            assembly {
@>              ++i;
            }
        }
```

Recommended Mitigation

Use inline assembly to increase the index of for loop. And use `++i` instead of `i++`.

[G-05] Cache the length in `for` loop

Links to Affected Code

`bonding_curves/ProtocolFeeDistributorV1.sol#L129`

`bonding_curves/ProtocolFeeDistributorV1.sol#L159`

`bonding_curves/ProtocolFeeDistributorV1.sol#L192`

Details

To avoiding extra gas fee consumption by reading the length of the array during each iteration, consider caching the length of array.

```
        for(uint8 i = 0; i < recipients.length; i++) {
```

For example, above statement can be rewritten as follow:

```
        uint length = recipients.length;
        for(uint8 i = 0; i < length; i++) {
```

Recommended Mitigation

Cache the length of array, then use it in `for` loop.

[G-06] Don’t calculate inside the statement.

Links to Affected Code

bonding_curves/ETHShares.sol#L136

Details

Don’t execute unnecessary calculation inside statement unless it’s crucial for readability.

```
return (10**5 / 2) * super.getPrice(supply, amount);
```

Recommended Mitigation

Consider updating above statement as follow:

```
return 25000 * super.getPrice(supply, amount);
```

Conclusion

All contracts in source are well constructed and written.
I also checked about the solidity version issues and EIP712 compatibility against overall contracts in scope, but there is no founded vulnerability.
Especially, updated leaderboard contract is reviewed and discussed well about suggested issue with dev team.