CD SECURITY

AUDIT REPORT

Arbero

July 2025

Introduction

A time-boxed security review of the **Arbero** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Arbero**

The Bonding Converter developed by Arbera Labs facilitates the conversion of `oBERO` to `arBERO` through a 90-day vesting mechanism or an instant claim option. Users can bond `OHM` to receive `arBERO` with a guaranteed profit, where the fast-claim route yields approximately 10% more in dollar value than bonded `OHM` and vested `oBERO`.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 0df754b85722a4d33f30874b595cca751732d66e**

Scope

The following smart contracts were in scope of the audit:

- `contracts/arbero/BondingConverter.sol`

- `contracts/arbero/ArberoOracle.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 2 issues
- Medium: 2 issues
- Low: 6 issues
- Informational: 2 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	`dollarValue` is not scaled correctly	Critical	Fixed
[C-02]	Scaling issues when calculating the `tvl` of the brarBERO-brOHM LP token	Critical	Fixed
[M-01]	Bond can be done before any vesting	Medium	Acknowledged
[M-02]	Existing native tokens can be drained by creating 1 wei bonds	Medium	Acknowledged
[L-01]	Contract addresses are not yet updated	Low	Fixed
[L-02]	The `fastClaim` and `bond` functions lack deadline input parameter	Low	Fixed
[L-03]	The `fastClaim` and `bond` functions lack slippage	Low	Fixed
[L-04]	Single step ownership transfer in use	Low	Fixed
[L-05]	The `getBrLbgtPriceandLpTVL` and `getBrOhmPriceAndLpTVL` functions lack validation for PYTH oracle output	Low	Fixed
[L-06]	The `fastClaim` function does not update the `arBERO` premium price	Low	Acknowledged
[I-01]	Withdraw of residual `arBERO` and native token is not possible	Informational	Fixed
[I-02]	Incorrect `BondCreated` and `BondUnlocked` event emissions	Informational	Fixed

Detailed Findings

[C-01] `dollarValue` is not scaled correctly

Severity

Impact: High

Likelihood: High

Description

Let's take a look at the following code:

```
    uint dollarValue = _amount * uint256(uint64(ohmBasePrice.price));
```

Here, the `dollarValue` variable is obtained by multiplying `amount` of OHM tokens provided by the users on the price of the OHM (in 8 decimals). The problem is that the amount is not scaled to `1e8` which is ultimately used to represent the actual dollar value, according to this comment from the `ArberoOracle` contract:

```
return (arberoPrice - oBeroPrice) / 1e10; // Return the premium in USD, 8
decimal points
```

So, if the price of the OHM token is 20$ (20e8) and the we provide 100 of these tokens (OHM has 9 decimals so it'll be 100e9), we'll eventually get 2000e17 which is not a correct dollar value. This is also used later to derive the `arberoToClaim`:

```
uint arberoToClaim = (_bonus * 1e18) / arberoPremium;
```

The ARBERO token is just an usual 18 decimal ERC20 so `bonus` has to be of 8 decimals for this expression to be scaled correctly as `arberoPremium` has 8 decimals as well. Otherwise, this would be a very huge amount of ARBERO tokens to claim.

Recommendations

Scale the dollar value to be of 8 decimals.

[C-02] Scaling issues when calculating the `tvl` of the brarBERO-brOHM LP token

Severity

Impact: High

Likelihood: High

Description

The following comment states that the `tvl` returned by the function `_getBrOhmPriceAndLpTVL()` in the `ArberoOracle` smart contract has to have 8 decimal places:

```
@return tvl TVL in brarBERO-brOHM LP, in USD, 8 decimal points.
```

The variable itself is calculated this way:

```
    uint tvl = (reserve * 2 * price) / 1e18;
```

It takes the reserve of one of the tokens (BARBERO or BROHM) and then multiplies it by price (that has `1e8` scaling) and divides by `1e18`. The problem here is that the `reserve` has to be of `1e18` scaling so that `1e18` in the numerator and denominator is omitted and the end result would be of `1e8` scaling because of the price. However, one of the tokens can BROHM that could potentially have 9 decimal places (similar to the OHM token that has 9 decimals) and, in this case, the result would be completely incorrect. If the BROHM token does not have 9 decimal places and has 18 decimals instead, then this expression would be also incorrect in terms of scaling because `ohmInBrOhm` would have 9 decimals:

```
        uint cbr = (ohmInBrOhm * 1e18) / brOhmTotalSupply;
```

Recommendations

Change the scaling in one of the expressions depending on the number of decimals the BROHM token has (the address is currently incorrect so there is no exact info about the token). It can be seen that OHM has 9 decimals here.

[M-01] Bond can be done before any vesting

Severity

Impact: Low

Likelihood: High

Description

Based on the project's documentation the protocol assumes that user firstly performs vesting, then bond action. However, there is no security control implemented prohibiting user performing bond action in prior of vesting. Thus, an user can make any bond in advance. Then, after any period, in particular after 7 days, the user can unlock bonus, vest and immediately fast claim to obtain `arBERO` tokens instantly. As a result, users may circumvent protocol assumption with advance planning and bond actions.

```
```

```
    function bond(uint _amount, address _to, bytes[] calldata
_priceUpdate) external {
        require(_amount > 0, AmountIsZero());
        OHM.safeTransferFrom(msg.sender, ohmReceiver, _amount);

        // Update Pyth price
        uint fee = PYTH.getUpdateFee(_priceUpdate);
        PYTH.updatePriceFeeds{value: fee}(_priceUpdate);

        // get price
        PythStructs.Price memory ohmBasePrice =
PYTH.getEmaPriceNoOlderThan(OHM_PRICE_ID, 60);
        require(ohmBasePrice.expo == -8, InvalidPriceExponent());
        require(ohmBasePrice.price > 0, InvalidPriceVolume());
        uint dollarValue = _amount * uint256(uint64(ohmBasePrice.price));
        dollarValue = (dollarValue * (1e4 + fastClaimProfit)) / (10 **
22); // divide by (1e18 * 1e4)
        require(dollarValue > 0, NotEnoughBonus());

        // Store bonus
        Bond memory newBond = Bond({bonusValueUsd: dollarValue,
unlockTime: block.timestamp + 7 days});

        userBonds[_to].push(newBond);
        emit BondCreated(_to, _to, dollarValue);
    }
```

Recommendations

It is recommended to review defined business rules of the protocol and decide, whether additional validation must be implemented that allows users to bond only while having valid vesting record.

[M-02] Existing native tokens can be drained by creating 1 wei bonds

Severity

Impact: Medium

Likelihood: Medium

Description

The BondingConverter.sol contract receives native tokens using the receive functionality.

```
receive() external payable {}
```

This is required to update the prices on the PYTH feeds. However, if the contract is pre-funded through the receive function, anyone can use up the native tokens by creating minimal 1 wei bonds.

```
PYTH.updatePriceFeeds{value: fee}(_priceUpdate);
```

Recommendations

It is recommended to:

1. Make the bond and fastClaim functionality payable, check if msg.value is enough to pay the fee, refund anything extra at the end of the function. Remove the receive function in this case.

[L-01] Contract addresses are not yet updated

Description

There are currently several contract addresses that are needed to be updated but are not (even those without a "TODO" comment):

```
IDecentralizedIndex public constant BROHM =
IDecentralizedIndex(0x883899D0111d69f85Fdfd19e4B89E613F231B781); // TODO
correct address
IDecentralizedIndex public constant BRLBGT =
IDecentralizedIndex(0x883899D0111d69f85Fdfd19e4B89E613F231B781);
```

```
    bytes32 constant OHM_PRICE_ID =
0x3a8c0214e4fb7f1dd7792ed4d5b2971372e52f088fcd9cc02309253cbdc4a70e;
    bytes32 constant LBGT_PRICE_ID =
0x3a8c0214e4fb7f1dd7792ed4d5b2971372e52f088fcd9cc02309253cbdc4a70e;
```

```
    IExampleSlidingWindowOracle public constant UNISWAP_POOL_ORACLE =

IExampleSlidingWindowOracle(0x883899D0111d69f85Fdfd19e4B89E613F231B781);
// TODO correct address


    IUniswapV2Pair public constant BRARBERO_BRLBGT_PAIR =
IUniswapV2Pair(0x883899D0111d69f85Fdfd19e4B89E613F231B781);
    IUniswapV2Pair public constant BRARBERO_BROHM_PAIR =
IUniswapV2Pair(0x883899D0111d69f85Fdfd19e4B89E613F231B781);
```

Recommendations

Update the addresses.

[L-02] The `fastClaim` and `bond` functions lack deadline input parameter

Description

The `fastClaim` and `bond` functions lack any kind of deadline security check, where within such period the transaction should be finalised without revert. Thus, due to transactions execution delay, the expect output amount can vary as the price can change rapidly.

```
    function fastClaim(uint _bonus, uint _vestId, address _to, bytes[]
calldata _priceUpdate) external {
```

Recommendations

It is recommended to implement the deadline for the aforementioned functions.

[L-03] The `fastClaim` and `bond` functions lack slippage

Description

The `fastClaim` and `bond` functions lack any kind of slippage mechanism or minimum expected amount check for the arBERO tokens that are supposed to be transferred in the former function and how much dollar equivalent value should be saved in the state variable for the latter. Thus, due to the rapid price changes and transactions execution delay, the expect amount can significantly vary.

```
    function fastClaim(uint _bonus, uint _vestId, address _to, bytes[]
calldata _priceUpdate) external {
```

Recommendations

It is recommended to implement slippage or minimum expected amount check.

[L-04] Single step ownership transfer in use

Description

The `BondingConverter` implements `Ownable` library which implements single step ownership transfer. In the event of transferring the ownership to the invalid address, all functions protected by the access control will become permanently unavailable.

```
contract BondingConverter is Ownable, IBondingConverter {
```

Recommendations

It is recommended to use `Ownable2Step` instead.

[L-05] The `getBrLbgtPriceandLpTVL` and `getBrOhmPriceAndLpTVL` functions lack validation for PYTH oracle output

Description

The `getBrLbgtPriceandLpTVL` and `getBrOhmPriceAndLpTVL` functions do not validate the correctness of data returned by the `PYTH` oracle. On the contrary, the `bond` function has proper input validation implemented for both `expo` and `price`. In the rare instances, it may result in usage of incorrect price of temporary disabled or disturbed oracle.

```
    function getBrOhmPriceAndLpTVL() public view returns (uint, uint) {
        uint ohmInBrOhm = OHM.balanceOf(address(BROHM));
        uint brOhmTotalSupply = BROHM.totalSupply();
        if (brOhmTotalSupply == 0 || ohmInBrOhm == 0) return (0, 0);
        uint cbr = (ohmInBrOhm * 1e18) / brOhmTotalSupply;
        PythStructs.Price memory ohmBasePrice =
PYTH.getEmaPriceNoOlderThan(OHM_PRICE_ID, 60);

        uint price = (uint256(uint64(ohmBasePrice.price)) * cbr) / 1e18;
        require(price > 0, "Invalid OHM price");
        (uint r0, uint r1, ) = BRARBERO_BROHM_PAIR.getReserves();
        uint reserve = BRARBERO_BROHM_PAIR.token0() == address(BROHM) ? r0
: r1;
        uint tvl = (reserve * 2 * price) / 1e18;
        return (price, tvl);
    }
```

```
    function getBrLbgtPriceandLpTVL() public view returns (uint price,
uint tvl) {
        uint stlbgtInBrLbgt = STLBGT.balanceOf(address(BRLBGT));
        uint brLbgtTotalSupply = BRLBGT.totalSupply();
        if (brLbgtTotalSupply == 0 || stlbgtInBrLbgt == 0) return (1e18,
0);
```

```
        uint cbr = (stlbgtInBrLbgt * 1e18) / brLbgtTotalSupply;
        uint lbgtInStLbgt = STLBGT.convertToAssets(1e18);
        PythStructs.Price memory lbgtBasePrice =
PYTH.getEmaPriceNoOlderThan(LBGT_PRICE_ID, 60);

        price = (uint256(uint64(lbgtBasePrice.price)) * lbgtInStLbgt *
cbr) / 1e36;
        (uint r0, uint r1, ) = BRARBERO_BRLBGT_PAIR.getReserves();
        uint reserve = BRARBERO_BRLBGT_PAIR.token0() == address(BRLBGT) ?
r0 : r1;
        tvl = (reserve * 2 * price) / 1e18;
    }
```

Recommendations

It is recommended to implement missing validation.

[L-06] The `fastClaim` function does not update the `arBERO` premium price

Description

The `fastClaim` function makes use of the `getArberoPremium` function. However, it does not enforce a call to the `update` function, which updates the `oberoPricePoints` internal collection. Thus, the `fastClaim` function can make use of stale price of arBERO premium.

```
    function update() external {
        UNISWAP_POOL_ORACLE.update(address(BRARBERO), address(BROHM));
        UNISWAP_POOL_ORACLE.update(address(BRARBERO), address(BRLBGT));
        uint oberoPrice = BERO.getOTokenPrice();
        uint index =
UNISWAP_POOL_ORACLE.observationIndexOf(block.timestamp);
        oberoPricePoints[index] = OberoPricePoint({timestamp:
block.timestamp, price: oberoPrice});
    }
```

Recommendations

It is recommended enforce the `update` call within the `fastClaim` function instead of relying of manual update.

[I-01] Withdraw of residual `arBERO` and native token is not possible

Description

The protocol is designed to obtain `arBERO` ERC20 tokens directly, so this token can be transferred as reward in the `claimReleased` and `fastClaim` functions.

The protocol is also designed to obtain native token directly, so `PYTH` fee can be paid automatically. However, there is no mechanism allowing to withdraw `arBERO` token or native token sent to the contract, whenever the contract becomes outdated or not used anymore.

```
    receive() external payable {}
```

Recommendations

It is recommended to implement `arBERO` token and native token withdrawal with proper access control.

[I-02] Incorrect `BondCreated` and `BondUnlocked` event emissions

Description

When a bond is created, it uses the `_to` address in the first parameter. This is incorrect since it should be using the address of the user that created the bond.

```
emit BondCreated(_to, _to, dollarValue);
```

The value of `userBond.bonusValueUsd` will be incorrect in the emission below. This is because:

1. `userBond` is a storage reference
2. When the `userBond` is deleted before the event emission, the storage reference is also updated
3. Since we removed the bond from the user's list, it will either emit the value of another bond that was replaced in the function `_removeFromUserBondsArray` or 0 if no more bonds exist. This is because `userBond` is a storage reference

```
function unlockBonus(address _user, uint _bondId) external {
        uint noUserBonds = userBonds[_user].length;
        require(_bondId < noUserBonds, InvalidBondId(_bondId));
        Bond storage userBond = userBonds[_user][_bondId];
        require(userBond.unlockTime <= block.timestamp,
BondIsNotUnlocked(_bondId));
        userBonuses[_user] += userBond.bonusValueUsd;
        _removeFromUserBondsArray(_user, _bondId);
        emit BondUnlocked(_user, _bondId, userBond.bonusValueUsd); << <<
    }
```

Recommendations

- Use `msg.sender` in the first field of the `BondCreated` event emission.
- Either remove the bond after the `BondUnlocked` event emission or save a memory copy of the value.