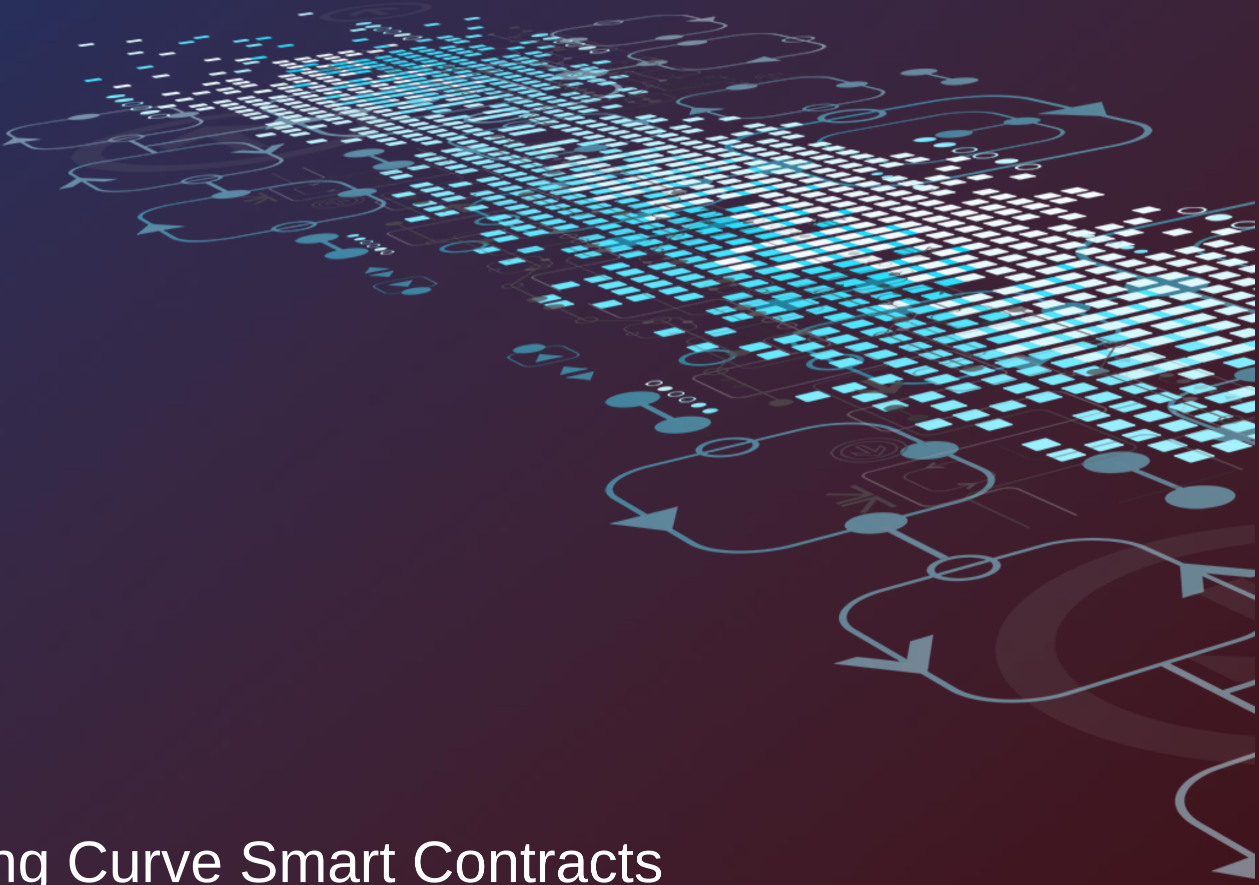
alethea.ai



Bonding Curve Smart Contracts

Audit Report

August 9, 2024

Mykyta Mesentsev

Table of Contents

Overview

This document describes the audit results for smart contract of ERC20 Bonding Curve in Alethea AI protocols.

The smart contract will enable easy and quick creation of tokens, and has a mechanism of “graduating” these tokens to a DEX.

Each token will have a starting market cap, a target market cap, and liquidity needed to reach the target market cap. Once the target market cap is reached - the smart contract is designed to transfer the entire liquidity for that particular token to a Uniswap V3 Pool that it creates.

Token price will be determined by the specified bonding curve formula by protocol team. Here’s the bonding curve price formula of the protocol:

$$p = a \cdot \mathrm{e}^{b \cdot t}$$

p : token price

t : token supply

a, b : constants that was determined by protocol team

Project	Alethea AI
Repository	AI Protocol Contracts
Commit Hash	bd122d45032a69b547453f65aa740ade2dbdde65
Network	Ethereum Mainnet, Base
Type of Project	Bonding Curve
Audit Period	August 1 to August 9

Risk Classification

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

High	0
Medium	2
Low	2
Info	3
Gas	2
Total	8

Finding Details

Medium

[M-01] Reentrancy possiblity in `__buytokens(), sellTokensTo()` function.

Severity

Medium

Affected Lines

ERC20BondingCurveFactoryV1.sol# L1061, L1143

Finding Description

In __buytokens() function, the issue could potentially lead to a reentrancy attack if not addressed. In terms of the logic, if ether amount that buyer sent is greater than expected and buyer is malicious smart contract, they can try to attempt to reentrancy call.
In this case, there may be happened unexpected error because if reentrancy attack is occurred, buyer can use same token amount and ether amount. Vice versa selltokensTo() function.

```
// return the change back to the buyer; here we do fail on error
// note: if any of the fees failed to transfer, they are sent to the buyer
if(_msgValue > value) {
  payable(msg.sender).transfer1(_msgValue - value); // may occur reentrancy attack
}
```

```
    // note: if any of the fees failed to transfer, they are sent to the seller
    _beneficiary.transfer1(price - protocolFee - subjectFee);
```

Recommended Mitigation

We can use openzepplin modifier for the reentrancy.

```
  function __buyTokens(
    address _economyToken,
    uint256 _etherAmount,
    uint256 _minAmountsOut,
    uint256 _msgValue,
    address _beneficiary
  ) private returns (uint256) nonReentrant {
```

```
function sellTokensTo(address _economyToken, uint256 _amount, address payable
_beneficiary) public nonReentrant {
```

[M-01] DoS possiblity in __processLiquidityGraduationFee() function.

Severity

Medium

Affected Lines

ERC20BondingCurveFactoryV1.sol# L1312, L1322, L1325, L933

Finding Description

In __processLiquidityGraduationFee() function, the issue could potentially lead to a Denial of Service (DoS) attack if not addressed.

In terms of the logic, if the eth reach out to the marketcapBenchmark, the economy token will be listed to the UniswapV3 dex. buyTokens function calls __processLiquidityGraduationFee() function internally.

```
function __processLiquidityGraduationFee(address economyToken) private {
  // process liquidity graduation protocol token share
  if(protocolTokenShare != 0) {
    uint256 tokenShare = protocolTokenShare * ERC20(economyToken).totalSupply() / 1
ether;
    IERC20(economyToken).transfer(protocolShareDestination, tokenShare);
  }

  if(liquidityGraduationFee != 0 ) {
    // derive fee amount from accumulated ether in pool
    uint256 feeAmount = (economies[economyToken].ethInPool * liquidityGraduationFee)
/ 1 ether;
    address creator =
ERC721(aiAgentCollection).ownerOf(economies[economyToken].aiAgentTokenId);

    uint256 halfFee = feeAmount / 2;
    // process liquidity graduation protocol fee
    payable(protocolFeeDestination).transfer1(halfFee); // if protocolFeeDestination
is contract address and there isn't receive function

    // process liquidity graduation creator fee
    payable(creator).transfer1(feeAmount - halfFee); // @audit if creator is
contract address and there isn't receive function
  }
}
```

In this function, if creater is contract and it doesn't have receive function, any transaction will be reverted.
So after raising funds, it cannot be listed to dex.

Recommended Mitigation

Need to consider checking contract address or not while create ecnomy token.

```
function checkContract(
    address target
  )
  public
  view
  returns (bool isContract) {
    if (target.code.length == 0) {
      isContract = false;
    } else {
      isContract = true;
    }
  }
}
```

Low

[L-01] Wrong minimum and maximum tick value in `__addLiquidityCreatePool` function.

Severity

Low

Affected Lines

ERC20BondingCurveFactoryV1.sol#L1376, L1378, L1382, L1383

Finding Description

In `__addLiquidityCreatePool` function, there is calculations for lower and upper tick for provided liquidity.

```
        // calculate lower/upper tick
        if (isOneSidedLiquidity) {
            if (isToken0Liquidity){
                lowerTick = (currentTick <= 0) ? (currentTick / tickSpacing) *
tickSpacing : ((currentTick / tickSpacing) + 1) * tickSpacing;
                upperTick = int24((887270 / tickSpacing) * tickSpacing
            } else {
                lowerTick = int24((-887270 / tickSpacing) * tickSpacing
                upperTick = (currentTick < 0) ? ((currentTick / tickSpacing) - 1) *
tickSpacing : (currentTick / tickSpacing) * tickSpacing;
            }
        } else {
            lowerTick = int24((-887270 / tickSpacing) * tickSpacing
            upperTick = int24((887270 / tickSpacing) * tickSpacing
        }
```

But in this calculation, they are using wrong minimum and maximum tick value of Uniswap V3 (887272).

This will cause wrong calculation of lower and upper tick of liquidity, then may leads to unexpected behavior.

Recommended Mitigation

Update current value of `887270` as exact value of `887272`.

Mitigation Review

Fixed

[L-02] Missing check that `msg.sender` is actual NFT owner in eip712 function

Severity

Low

Affected Lines

ERC20BondingCurveFactoryV1.sol#L837-L871

Finding Description

When the `FEATURE_ALLOW_ANY_DEPLOYER` feature flag is disabled, users will only be able to launch economies via eip712 function.

```
    function eip712MintAndLaunchEconomy(
        CreateEconomyRequest calldata _req,
        bytes calldata _signature
    ) external virtual payable whenNotPaused returns (address){
        // ensure EIP-712 minting with authorization is enabled
        require(isFeatureEnabled(FEATURE_LAUNCH_ECONOMY_WITH_AUTH), "launch with
auth is disabled");

        // perform message integrity and security validations
        require(block.timestamp > _req.validAfter, "signature not yet valid");
        require(block.timestamp < _req.validBefore, "signature expired");
        // @audit-issue verify _req.aiAgentOwner is caller
```

However, in that function, there is no checking that caller is same as `aiAgentOwner` which is passed in `_req` data. It may cause damage and infringement of private actions.

Recommended Mitigation

Add `require` statement to verify `msg.sender` is same as `aiAgentOwner` value.

Mitigation Review

Fixed

Info

[I-01] Use cached value for duplicated calculation.

Severity

Informative

Affected Lines

ERC20BondingCurveFactoryV1.sol#L1246, L1249

Finding Description

```
        if(liquidityGraduationFee != 0 ) {
            // derive fee amount from accumulated ether in pool
            uint256 feeAmount = (economies[economyToken].ethInPool *
liquidityGraduationFee) / 1 ether;
            address creator =
ERC721(aiAgentCollection).ownerOf(economies[economyToken].aiAgentTokenId);

            // process liquidity graduation protocol fee
@>          payable(protocolFeeDestination).transfer1(feeAmount / 2);

            // process liquidity graduation creator fee
@>          payable(creator).transfer1(feeAmount / 2);
        }
```

In above codes, there are duplicated calculations of `feeAmount / 2`. They can be simplified as one calculation, then use cached value for each statements for efficiency if it doesn't hurt readability.

Recommended Mitigation

Consider updating code as follow:

```
        if(liquidityGraduationFee != 0 ) {
            // derive fee amount from accumulated ether in pool
            uint256 feeAmount = (economies[economyToken].ethInPool *
liquidityGraduationFee) / 1 ether;
            address creator =
ERC721(aiAgentCollection).ownerOf(economies[economyToken].aiAgentTokenId);

            feeAmount = feeAmount >> 1;
            // process liquidity graduation protocol fee
            payable(protocolFeeDestination).transfer1(feeAmount);
            // process liquidity graduation creator fee
            payable(creator).transfer1(feeAmount);
        }
```

Mitigation Review

Fixed

[I-02] Update the incorrect comments in contract

Severity

Informative

Affected Lines

ERC20BondingCurveFactoryV1.sol#L62

Finding Description

```
    struct EconomyDetails {
        // AI Agent NFT ID
        uint256 aiAgentTokenId;
        // Target market cap for the economy token
        uint256 targetMarketCap;
@>      // Amount of ALI in the pool // @audit-issue wrong comment
        uint256 ethInPool;
        // Economy token address
        address economyToken;
        // economy creator
        address economyCreator;
        // economy creation time
        uint64 economyCreationTime;
        // Flag indicating if liquidity pool is created
        bool isLpPoolCreated;
    }
```

In definition of `EconomyDetails` struct, there is a wrong comment for internal key `ethInPool`. It represents ETH amount in the pool, but the comment says ALI token.

Recommended Mitigation

Change the comment as exact one.

Mitigation Review

Fixed

[I-03] Missing `indexed` keyword in `TokenPrice` event

Severity

Informative

Affected Lines

ERC20BondingCurveFactoryV1.sol#L243

Finding Description

```
    /**
     * @dev Fired in `buyTokensTo()` and in `sellTokensTo()`
     *
     * @param tokenPrice price of economy token at the time event is emitted
     * @param timestamp time of the emitted token price
     */
@>  event TokenPrice(uint256 tokenPrice, uint256 timestamp); // @audit-issue missing
token price indexed
```

There is not indexed parameter in above `TokenPrice` event. Due to lack of indexed parameter, it may cause difficulty of filtering specific logs efficiently.

Recommended Mitigation

Consider updating event definition as follow:

```
    event TokenPrice(uint256 indexed tokenPrice, uint256 timestamp);
```

Mitigation Review

Fixed

Gas Optimization

[G-01] Consider using bitwise shift operator instead of division operator

Affected Lines

ERC20BondingCurveFactoryV1.sol#L1246, L1249

Descriptions

In Solidity, it is often more gas-efficient to use bitwise shift operations (`>>` and `<<`) instead of division (`/`) and multiplication (`*`) when performing arithmetic operations on integers.

```
            // process liquidity graduation protocol fee
@>          payable(protocolFeeDestination).transfer1(feeAmount / 2);

            // process liquidity graduation creator fee
@>          payable(creator).transfer1(feeAmount / 2);
```

Recommended Mitigation

Consider using shift operator instead of division.

Mitigation Review

Acknowledged

Note from Dev Team: The optimization proposed allows to save 53 gas. Leaving the division operator intact for better code readability.

[G-02] Use `!= 0` instead of `> 0` for Unsigned Integer Comparison

Affected Lines

ERC20BondingCurveFactoryV1.sol#L330, L740, L745, L946, L1007, …

Descriptions

In Solidity, when comparing unsigned integer values, it is generally considered safer to use the `!= 0` operator instead of the `> 0` operator. This is because the `> 0` operator might lead to unexpected behavior or vulnerabilities in certain scenarios. In this blog post, we will explore why using `!= 0` is recommended and provide examples to illustrate the potential risks of using `> 0`.

```
        // update if greater then 0 and less then 20 ether hardcap
@>      if (_marketcapBenchmark > 0  && _marketcapBenchmark <= 20 ether) {
            marketcapBenchmark = _marketcapBenchmark;
        }

        // update if non zero value
@>      if (_maxTokenSupply > 0) {
            maxTokenSupply = _maxTokenSupply;
        }
```

Recommended Mitigation

Consider replacing `> 0` with `!= 0`.

Mitigation Review

Not Valid

Note from Dev Team: While looking valid in theory, the finding doesn't confirm itself in tests. Tried both > 0 and != 0 in the updateGeneralConfig function and in both cases function consumes same 37193 gas.

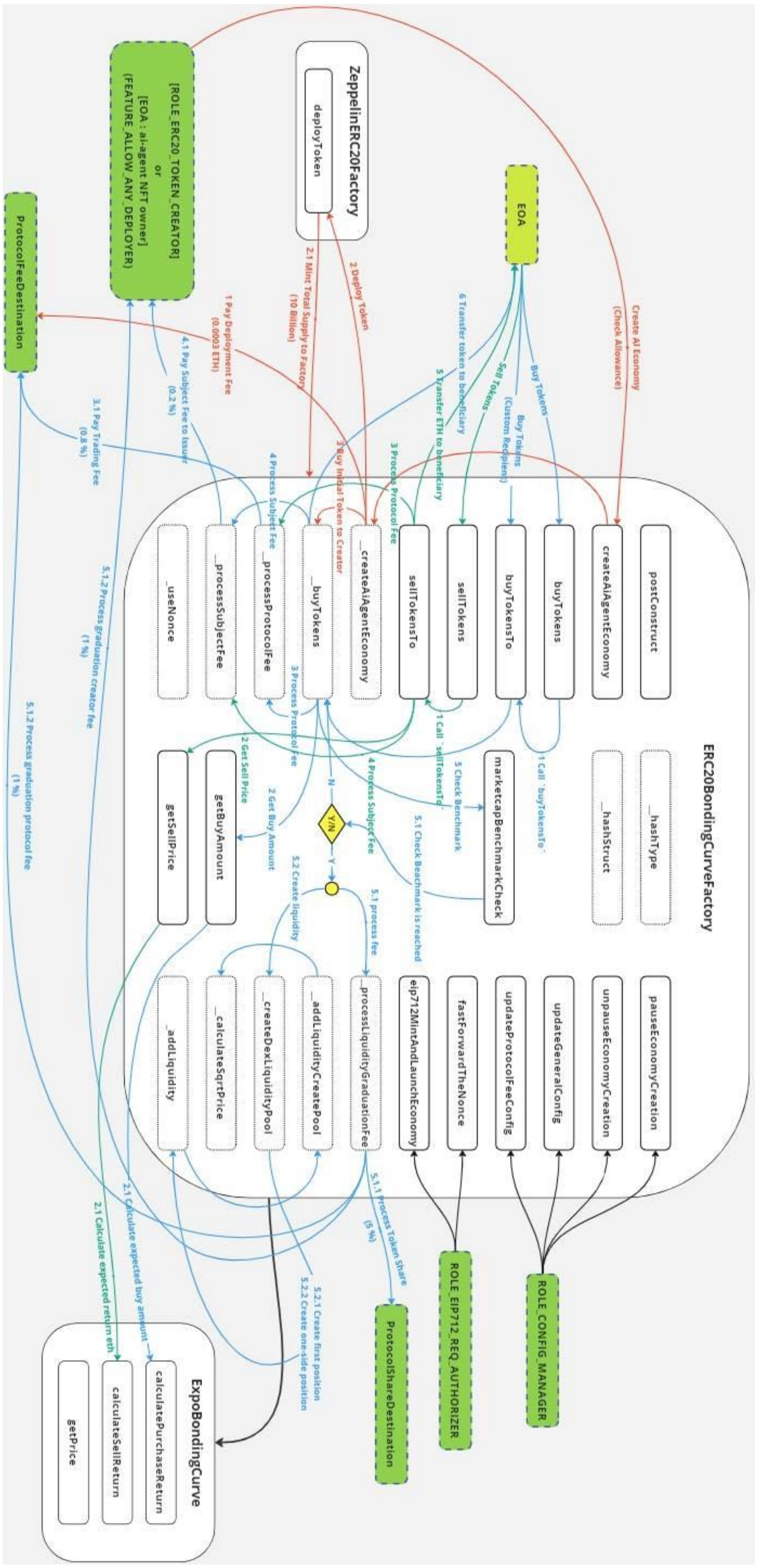
Comment: This is due to Solidity compiler’s versions.

Conclusion

The audit was performed focusing on finding the possibilities for loss of funds or other extra bugs, and optimizing the code. As a result of auditing, the contract is healthy for overall codes and well modified.

Depending on the logics of web3 interface and off-chain usages, they can be needed to perform changing between `public` and `external`, `private` and `internal` for efficiency.

Appendix: Analysis diagram of smart contract



ERC20BondingCurveFactory
postConstruct
createAiAgentEconomy
buyTokens
buyTokensTo
sellTokens
sellTokensTo
_createAiAgentEconomy
__buyTokens
__processProtocolFee
__processSubjectFee
_useNonce
_hashType
_hashStruct
marketcapBenchmarkCheck
getBuyAmount
getSellPrice
pauseEconomyCreation
unpauseEconomyCreation
updateGeneralConfig
updateProtocolFeeConfig
fastForwardTheNonce
eip712MintAndLaunchEconomy
__processLiquidityGraduationFee
__addLiquidityCreatePool
__createDexLiquidityPool
__calculateSqrtPrice
_addLiquidity
ZeppelinERC20Factory
deployToken
EOA
[ROLE_ERC20_TOKEN_CREATOR]
or
[EOA : ai-agent NFT owner]
(FEATURE_ALLOW_ANY_DEPLOYER)
ProtocolFeeDestination
ProtocolShareDestination
ROLE_EIP712_REQ_AUTHORIZER
ROLE_CONFIG_MANAGER
ExpoBondingCurve
calculatePurchaseReturn
calculateSellReturn
getPrice
Create AI Economy (Check Allowance)
Buy Tokens
Buy Tokens (Custom Recipient)
Sell Tokens
6 Transfer token to beneficiary
5 Transfer ETH to beneficiary
2 Deploy Token
2.1 Mint Total Supply to Factory (10 Billion)
1 Pay Deployment Fee (0.0003 ETH)
3 Buy Initial Token to Creator
3 Process Protocol Fee
4 Process Subject Fee
4.1 Pay Subject Fee to Issuer (0.2 %)
3.1 Pay Trading Fee (0.8 %)
5.1.2 Process graduation creator fee (1 %)
5.1.2 Process graduation protocol fee (1 %)
1 Call `sellTokensTo`
1 Call `buyTokensTo`
5 Check Benchmark
5.1 Check Beachmark is reached
3 Process Protocol Fee
4 Process Subject Fee
2 Get Sell Price
2 Get Buy Amount
Y/N
N
Y
5.2 Create liquidity
5.1 process fee
5.1.1 Process Token Share (5 %)
5.2.1 Create first position
5.2.2 Create one-side position
2.1 Calculate expected buy amount
2.1 Calculate expected return eth