HyperLaunch Bonding Curve

Security Audit Report

May 5, 2025

Copyright © 2025 BugBlow. All rights reserved.

No part of this publication, in whole or in part, may be reproduced, copied, transferred or any other right reserved to its copyright owner, including photocopying and all other copying, any transfer or transmission using any network or other means of communication, any broadcast for distant learning, in any form or by any means such as any information storage, transmission or retril system, without prior written permission from BugBlow.

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

BugBlow utilized a widely adopted approach to performing a security audit. Below is a breakout of how our team was able to identify and exploit vulnerabilities.

1.2.1 Code and architecture review:

- Review the source code.
- Read the project's documentation.
- Review the project’s architecture.

Stage goals

- Build a good understanding of how the application works.

1.2.2 Check code against vulnerability checklist:

- Understand the project’s critical assets, resources, and security requirements.
- Manual check for vulnerabilities based on the auditor's checklist.
- Run static analyzers.

Stage goals

- Identify and eliminate typical vulnerabilities (gas limit, replay, flash-loans attack, etc.)

1.2.3 Threat & Attack Simulation:

- Analyze the project's critical assets, resources and security requirements.
- Exploit the found weaknesses in a safe local environment.
- Document the performed work.

Stage goals

- Identify vulnerabilities that are not listed in static analyzers that would likely be exploited by hackers.
- Develop Proof of Concept (PoC).

1.2.4 Report found vulnerabilities:

- Discuss the found issues with developers
- Show remediations.
- Write an initial audit report.

Stage goals

- Verify that all found issues are relevant.

1.2.5 Fix bugs & re-audit code:

- Help developers fix bugs.

- ➢ Re-audit the updated code again.

Stage goals

- ➢ Double-check that the found vulnerabilities or bugs were fixed and were fixed correctly.

1.2.4. Deliver final report:

- ➢ The Client deploys the re-audited version of the code
- ➢ The final report is issued for the Client.

Stage goals

- ➢ Provide the Client with the final report that serves as security proof.

Severity classification

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Vulnerabilities leading to assets theft, fund access locking, or any other loss of funds.
High	Vulnerabilities that can trigger contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Mediu m	Vulnerabilities that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Vulnerabilities that do not have a significant immediate impact and could be easily fixed.

1.3 Project Overview

Bonding Curve smart contracts implement decentralized bonding curve system with pre-bonding capabilities and eventual migration to Uniswap V3, featuring LP NFT locking for security. The following features are highlighted: Pre-bonding phase for initial token distribution; Active trading phase with constant product AMM formula; Automatic migration to Uniswap V3; LP NFT locking.

A detailed technical functions analysis is presented in figures 1-5.

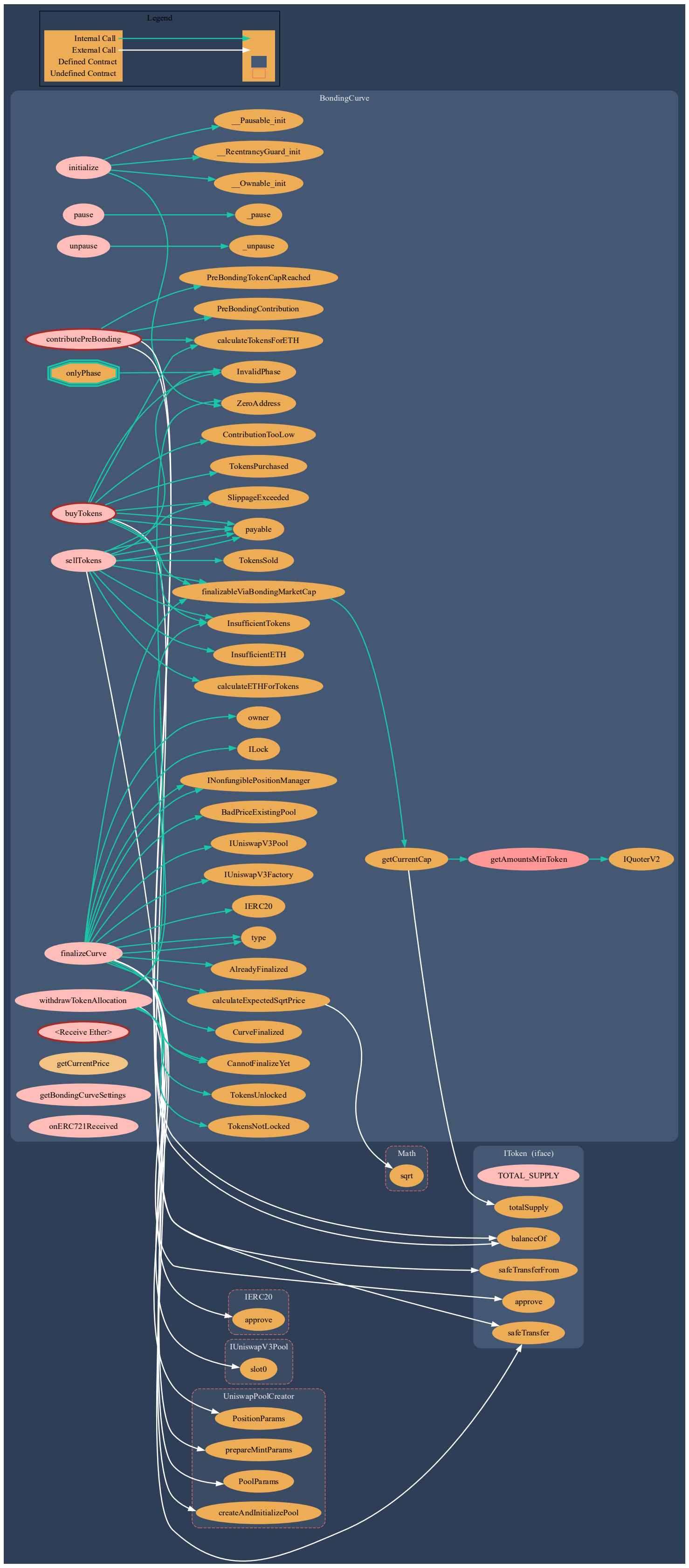

Figure 1. Control flow graph of BondingCurve

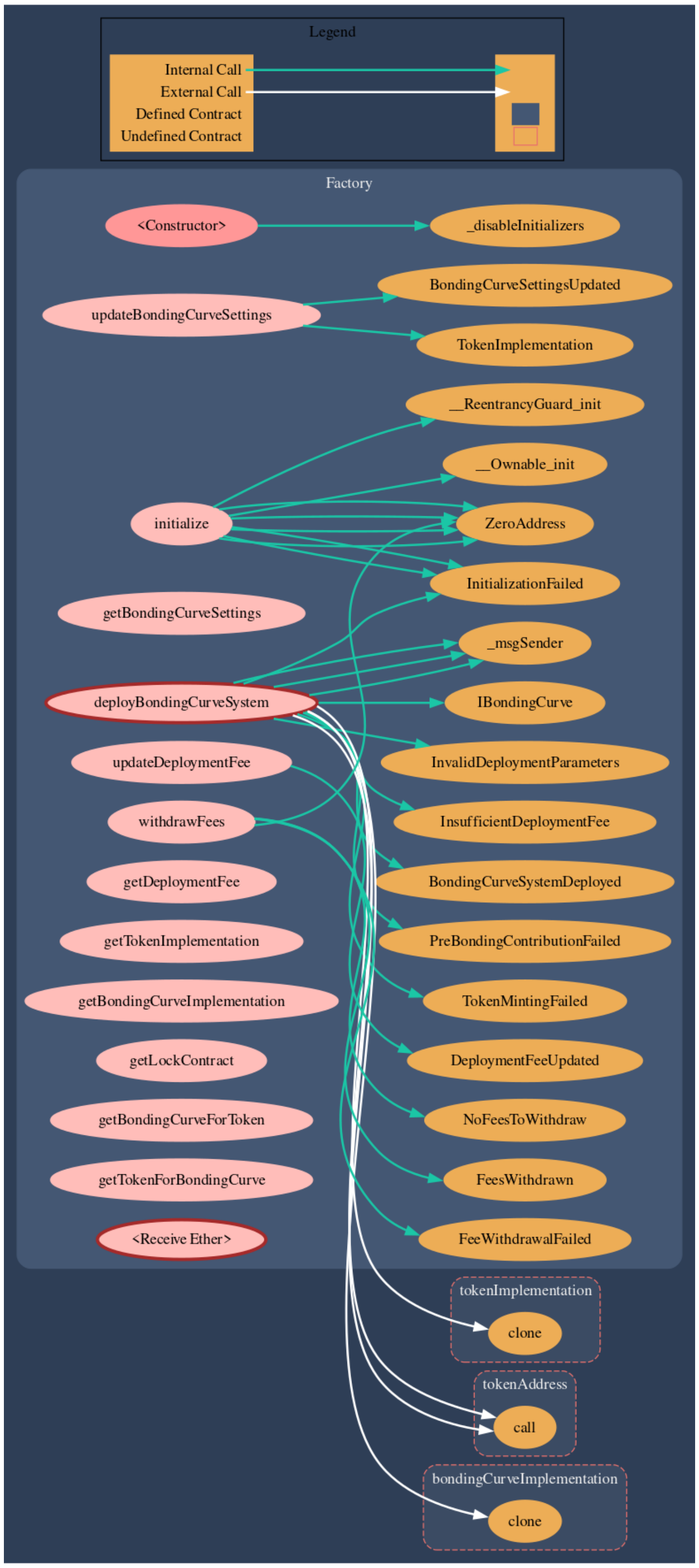

Figure 2. Control flow graph of Factory

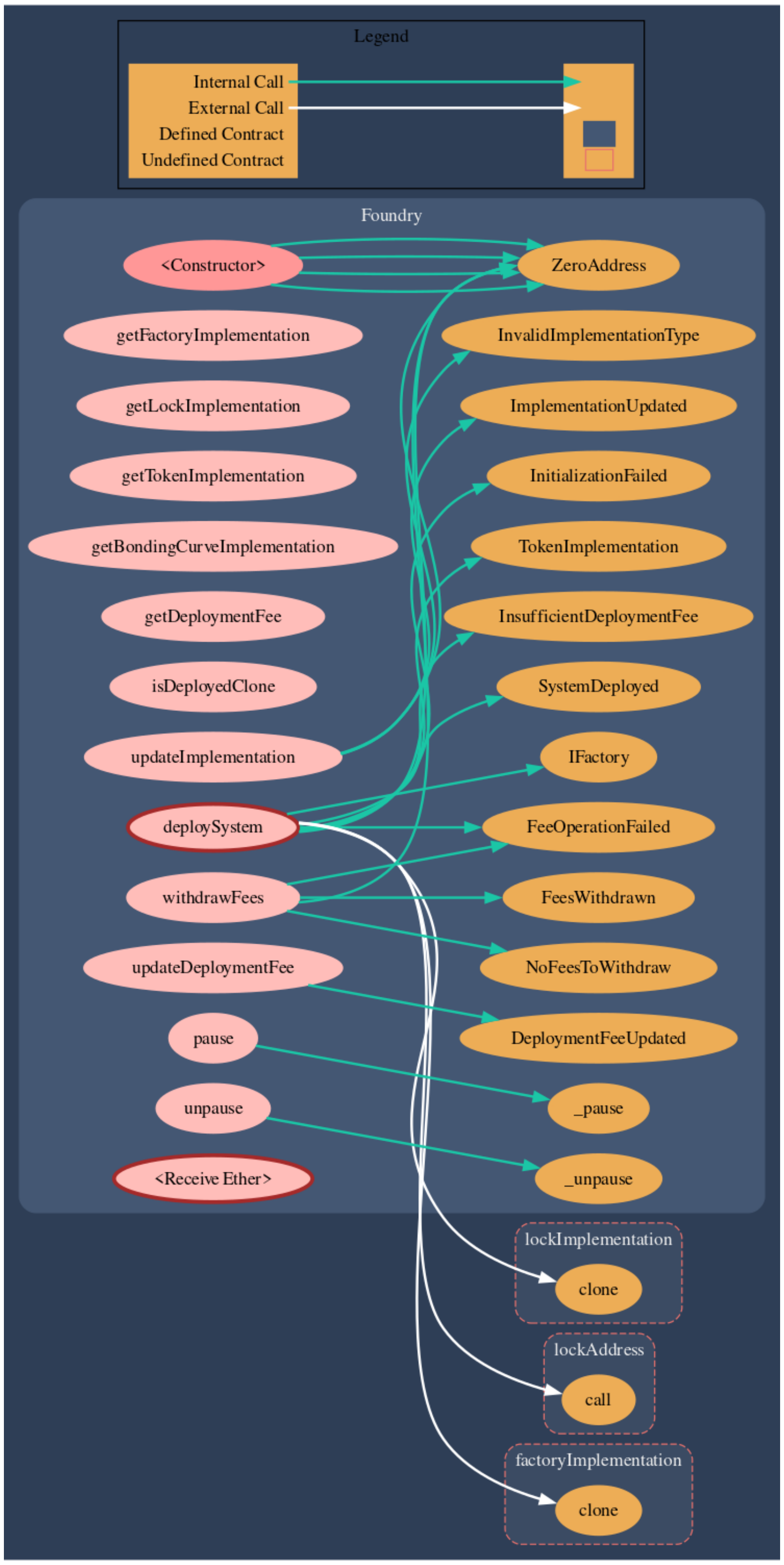

Figure 3. Control flow graph of Foundry

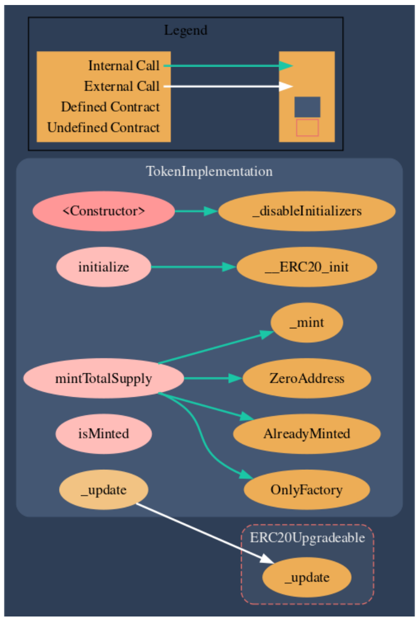

Figure 4. Control flow graph of Token Implementation

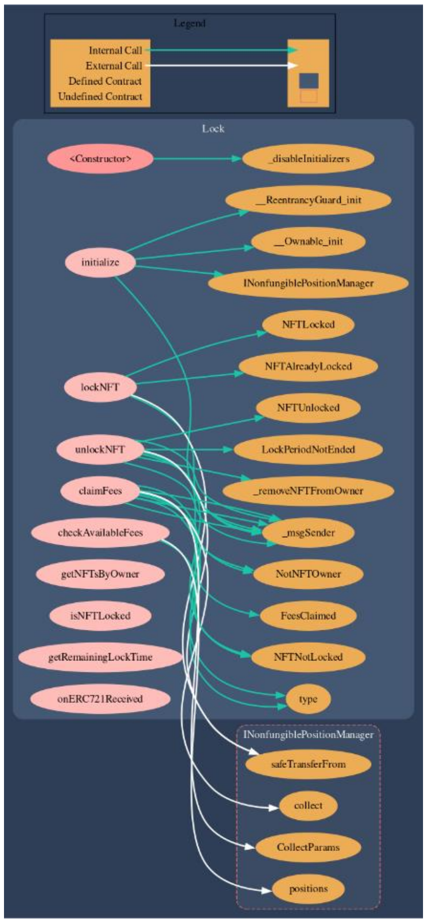

Figure 5. Control flow graph of Locking Mechanism

1.4 Project Dashboard

Project Summary

Title	Bonding Curve
Client name	HyperLaunch
Project name	IFO Contracts
Timeline	28.03.2025 - 02.04.2025
Number of Auditors	3

Project Last Log

Date	Commit Hash	Note
26.03.2025	2a0d1938f9e4cc53b3f57bb049f62778e33afdf2	upd readme

Project Scope

The audit covered the following smart contract files.

File name
contracts/BondingCurve.sol
contracts/Factory.sol
contracts/Foundry.sol
contracts/Lock.sol

contracts/TokenImplementation.sol
libraries/UniswapPoolCreator.sol
hardhat.config.js
Interfaces/*

1.5 Summary of findings

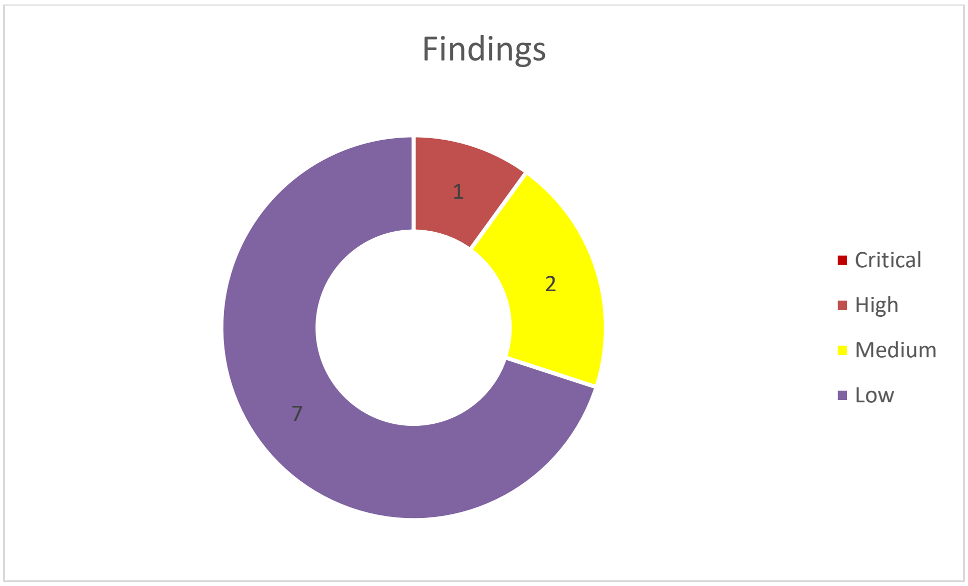

Figure 1. Findings chart

Severity	# of Findings
CRITICAL	0
HIGH	1
MEDIUM	2
LOW	7

2. FINDINGS REPORT

2.1 Critical

Not found

2.2 High

2.2.1 Leftover tokens locked in the contract forever

Status: Fixed

Description

The contract's **actual ETH balance** (address(this).balance) can be **less** than the internal accounting variable *ethReserve* because of the use of *virtualEth* in the pre-bonding stage. Essentially, the contract adds *virtualEth* to *ethReserve* (an internal variable) but that "virtual" portion of ETH is never actually sent to the contract. This inflates *ethReserve* above the real on-chain balance of ETH. As a result, after finalization there are some leftover tokens stuck in the contract (not counting pre-bonding contribution Tokens). Here is a more detailed example.

1. Assume the contract's reserves at finalization are:
    - *ethReserve* = 10 ETH
    - *tokenReserve* = 100 tokens
2. The **actual ETH balance** in the contract might be **8 ETH** (realEthBalance = 8), because some portion of ETH was spent on user redemptions or there was a mismatch.
3. The formula says:

$$tokenLiquidity = \frac{realEthBalance * tokenReserve}{ethReserve} = \frac{8 * 100}{10} = 80$$

4. The contract deposits **8 ETH** and **80 tokens** into Uniswap.
5. That leaves **20 tokens** (100 - 80) still sitting in the contract.

If the contract has **no function** to withdraw or "rescue" tokens after finalization and returning the prebonding tokens, then those **20 tokens** remain in the BondingCurve contract forever, effectively stuck.

Remediation:

Implement withdrawLeftoverTokens functions that can be only called after the finalization phase and the withdrawal of the prebonding contribution tokens.

2.3 Medium

2.3.1 Pre-bonding contribution can be 0 or near 0

Status: Acknowledged

Description

It's possible for the **pre-bonding contribution** (i.e. the amount of tokens minted from *msg.value* in *contributePreBonding*) to come out **equal to zero** if *msg.value* is small enough relative to *settings.virtualEth* and the token supply. Because the result is calculated with integer division, you can end up with (tokensOut == 0) if the ratio is tiny.

Effectively, if *msg.value = newEthIn* is too small, and *settings.virtualEth(_ethReserve*) is large enough, then the division result can be 0.

This will make the prebonding functions useless, jumping to the next phase without fulfilling the requirements of the first phase.

Remediation:

Set a threshold (if condition) what a minimum pre-contribution must be (*tokensOut*) and revert if it's below the minimum.

2.3.2 Early finalization of the bonding curve (sandwich attack)

Status: Fixed

Description

A finalization of the bonding curve may occur early due to a malicious actor's allowed behavior. Here is a breakdown of their actions:

- The attacker's transaction(s) perform a **large trade** on buying tokens, moving the price.

- The attacker calls the finalization function.
- The contract then finalizes prematurely.
- The attacker sells all their tokens within the same transaction or block right after finalization.

Remediation:

Use Delayed Finalization: E.g., “the contract can only finalize 5-10 blocks after hitting the threshold,” allowing the price to settle. As an alternative, you could use a Time-Weighted Oracle .

2.4 Low

2.4.1 Unexpected ETH Balance in BondingCurve Contract

Status: Fixed

Description

The buyTokens() and sellTokens() functions send trading and affiliate fees in a non-reversible manner:

```
function sellTokens(uint256 tokenAmount, uint256 minETH, address affiliate)
    external
    nonReentrant
    whenNotPaused
    onlyPhase(Phase.Bonding)
returns (uint256 ethToReceive, uint256 fee) {
    …
      // changed in favor of non-reverting Ether sending
        payable(msg.sender).send(ethToReceive);

        uint affiliateAmountFee;
        if (affiliate != address(0)) {
            affiliateAmountFee = fee * settings.affiliateFee / 10000;
            if (affiliateAmountFee > 0) {
                payable(affiliate).send(affiliateAmountFee);
            }
        }
        // changed in favor of non-reverting Ether sending
        payable(settings.feeTo).send(fee - affiliateAmountFee);
}
```

If fee transfers fail, ETH remains in the contract balance until the curve is finalized. This can affect the *prepareMintParams*`()` function by up to the tradeFee (%) amount.

Remediation

Move fee transfers to a separate function and call it from *buyTokens*() and *sellTokens*(). If the transfer fails, the function should send ETH to a designated fallback address, such as address(0) so that the contract wouldn't hold any excess balance.

```
function sendFees(
    address _affiliate,
    uint256 _fee
) private {
    uint affiliateAmountFee;
    if (_affiliate != address(0)) {
        affiliateAmountFee = _fee * settings.affiliateFee / 10000;
        if (affiliateAmountFee > 0) {
            bool success = payable(_affiliate).send(affiliateAmountFee);
            if (!success) {
                affiliateAmountFee = 0;
            }
        }
    }

    bool success = payable(settings.feeTo).send(_fee - affiliateAmountFee);
    if (!success) {
        payable(address(0)).send(_fee - affiliateAmountFee); //dust remover
    }
}
```

2.4.2 Unused variable

Status: Fixed

Description

An unused variable is declared in `BondingCurve.sol`:

```
  uint256 public MyMaxPreBondingTokens;
```

Remediation

Simply remove this line:

```
  uint256 public MyMaxPreBondingTokens;
```

2.4.3 Hardcoded NFT lock duration

Status: Fixed

Description

In the `Lock` contract, a commented-out line contains a production value of one year. If left as is, the Liquidity Provider (LP) will be able to create a new token every 30 minutes and will be unable to claim fees.

```
  uint256 public constant LOCK_DURATION = 1800; // 30 minutes for testing
```

Remediation

Make sure the 30 min value satisfies your economic model requirement and is not just for testing. A correct value must be set before deployment to production. Ideally, make it an immutable variable and initialize it during deployment.

2.4.4 Unnecessary local variable

Status: Fixed

Description

In `Factory.sol`, the function `deployBondingCurveSystem()` declares a redundant variable `requiredPayment`, which does not change within the function.

```
 uint256 requiredPayment = deploymentFee;
```

Remediation

Use the `deploymentFee` contract variable directly instead of declaring an unnecessary local variable.

2.4.5 Arbitrary "Owner" in `lockNFT`

Status: Acknowledged

Description

A user could call lockNFT(tokenId, someOtherAddress) where msg.sender actually owns the NFT, but they designate a different address as the "owner" in the Lock contract.

Then only that designated “owner” can unlock or claim fees. This might be intended behavior—perhaps a user wants to lock an NFT for a DAO or another account. If so, please disregard this issue. If not, then sender can use a different address and not be able to use their own to later unlock the NFT.

Remediation**:**

Make the locking mechanism only available to the sender or disregard this issue.

2.4.6 Magic fee value

Status: Fixed

Description

It’s easy to accidentally change 10000 to 1000 (or 100000) if someone is updating or refactoring the code and doesn’t realize the significance. Even a single missing zero breaks your entire fee logic.

Remediation**:**

Create a constant variable uint256 private constant FEE_DENOMINATOR = 10000 and use it instead of the magic number.

2.4.7 EVM Compatibility

Status: Fixed

Description

Your hardhat configuration does not specify the EVM's version. By default, hardhat sets the Paris EVM version. Your Lumia chain supports a different version (Shanghai), the OP codes and precompiles have different behavior in different EVM versions.

Remediation

Specify the EVM version in the hardhat configuration. However, keep in mind, that shanghai supports Solidity 0.8.20, so you make your contracts' versions "^0.8.19" to that the older versions could compile your contracts.

```
module.exports = {
  solidity: {
    version: "0.8.20",
    settings: {
      viaIR: true,
      evmVersion: “shanghai”,
      optimizer: {
        enabled: true,
        runs: 200,
      },
    },
  },
};
```

An example of differences in different EVM versions can be found here [2].

CONCLUSION

This security audit found no critical issues but did identify one high-severity finding and two medium-severity findings. The high-severity issue concerns leftover tokens that can become permanently locked in the contract. Medium-severity issues involve the possibility of a zero or near-zero pre-bonding contribution and the potential for early finalization of the bonding curve. A series of low-severity findings address minor code artifacts, unused variables, and the use of magic numbers..

REFERENCES

[1] https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

[2] https://docs.linea.build/get-started/build/ethereum-differences

[3] https://coinmarketcap.com/academy/article/what-are-sandwich-attacks-in-defi-and-how-can-you-avoid-them