Inferno.fun Launchpad Audit Report

Version 1.1

Conducted by:
Alin Barbatei (ABA)
Silverologist

January 27, 2025

Table of Contents

1 Introduction **4**

1.1 About ABA . . . 4

1.2 About Inferno.fun Launchpad . . . 4

1.3 Issues Risk Classification . . . 5

Impact . . . 5

Likelihood . . . 5

Actions required by severity level . . . 5

Informational findings . . . 5

2 Executive Summary **6**

2.1 Overview . . . 6

2.2 Audit Scope . . . 6

2.3 Summary of Findings . . . 6

2.4 Findings & Resolutions . . . 7

3 Findings **9**

3.1 High Severity Findings . . . 9

[H-01] Hardcoded slippage may block liquidity to AMM migration . . . 9

3.2 Medium Severity Findings . . . 11

[M-01] Multiple creators allowed per signature due to hash collisions . . . 11

[M-02] Migrating liquidity to AMM fails if excess dust ETH is returned . . . 13

[M-03] No incentives for creators to use the launchpad . . . 14

[M-04] AMM liquidity migration is blocked if no tokens are available in BondingCurve . 14

[M-05] Lynex liquidity pool can be pre-created and imbalance migration . . . 16

3.3 Low Severity Findings . . . 18

[L-01] BondingCurve contract implementation is uninitialized . . . 18

[L-02] Pool deployment is blocked if initialVirtualToken is less than TOTAL_SUPPLY . . 18

[L-03] Incorrect deadline when migrating liquidity to AMM . . . 19

[L-04] Add contingency in case pool never matures . . . 20

[L-05] Slight rounding error issues . . . 21

[L-06] Missing output amount preview functionality may lead to incorrect user slippage 22

[L-07] Launchpad swaps without deadlines . . . 22

[L-08] Token deployment signatures do not expire . . . 23

3.4 Informational Findings . . . 24

[I-01] Improve variable naming . . . 24

[I-02] Improve public functions return type . . . 24

[I-03] Redundant getters over public storage variables . . . 24

[I-04] Reentrancy guard is not initialized . . . 25

[I-05] Add security contact to contracts . . . 25

[I-06] Use a two-step ownership transfer routine . . . 26

[I-07] Cleanup inheritance . . . 26

[I-08] Style improvements and typographical error . . . 27

[I-09] _safeTransferETH can be simplified . . . 27

[I-10] deployTokenAndCreatePool can be simplified 28
[I-11] General codebase improvements 28
[I-12] Outdated and floating Solidity compiler version 29
[I-13] Tokens mistakenly sent to the contract are blocked 29

4 Inferno.fun Launchpad Deployed Tokens Audit **31**

5 Disclaimer **33**

1 Introduction

1.1 About ABA

ABA, or Alin Mihai Barbatei, is an established independent security researcher with deep expertise in blockchain security. With a background in traditional information security and competitive hacking, ABA has a proven track record of discovering hidden vulnerabilities. He has extensive experience in securing both EVM (Ethereum Virtual Machine) compatible blockchain projects and Bitcoin L2, Stacks projects.

Having conducted several solo and collaborative smart contract security reviews, ABA consistently strives to provide top-quality security auditing services. His dedication to the field is evident in the top-notch, high-quality, comprehensive smart contract auditing services he offers.

To learn more about his services, visit ABA's website abarbatei.xyz. You can also view his audit portfolio here. For audit bookings and security review inquiries, you can reach out to ABA on Telegram, Twitter (X) or WarpCast:

`https://t.me/abarbatei`

`https://x.com/abarbatei`

`https://warpcast.com/abarbatei.eth`

For this audit, Silverologist (@silverologist), a promising junior auditor, contributed his expertise to help secure the protocol.

1.2 About Inferno.fun Launchpad

The inferno.fun launchpad is a bounding curve type token launchpad that allows team-approved creators to deploy coins that trade on a bonding curve (`xy=k` with virtual ETH liquidity).

When liquidity reaches a certain threshold, the pool is migrated to the Lynex AMM (Automated Market Maker) as a `Token/WETH` liquidity pool. The LPs from adding the tokens to the pools are kept by the inferno.fun team.

The Lynex `Token/WETH` pool cannot be created until bounding curve has been deployed.

1.3 Issues Risk Classification

The current report contains issues, or findings, that impact the protocol. Depending on the likelihood of the issue appearing and its impact (damage), an issue is in one of four risk categories or severities: *Critical*, *High*, *Medium*, *Low* or *Informational*.

The following table show an overview of how likelihood and impact determines the severity of an issue.

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

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behavior that's not so critical.

Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Informational findings

Informational findings encompass recommendations to enhance code style, operations alignment with industry best practices, gas optimizations, adherence to documentation, standards, and overall contract design.

Informational findings typically have minimal impact on code functionality or security risk.

Evaluating all vulnerability types, including informational ones, is crucial for a comprehensive security audit to ensure robustness and reliability.

2 Executive Summary

2.1 Overview

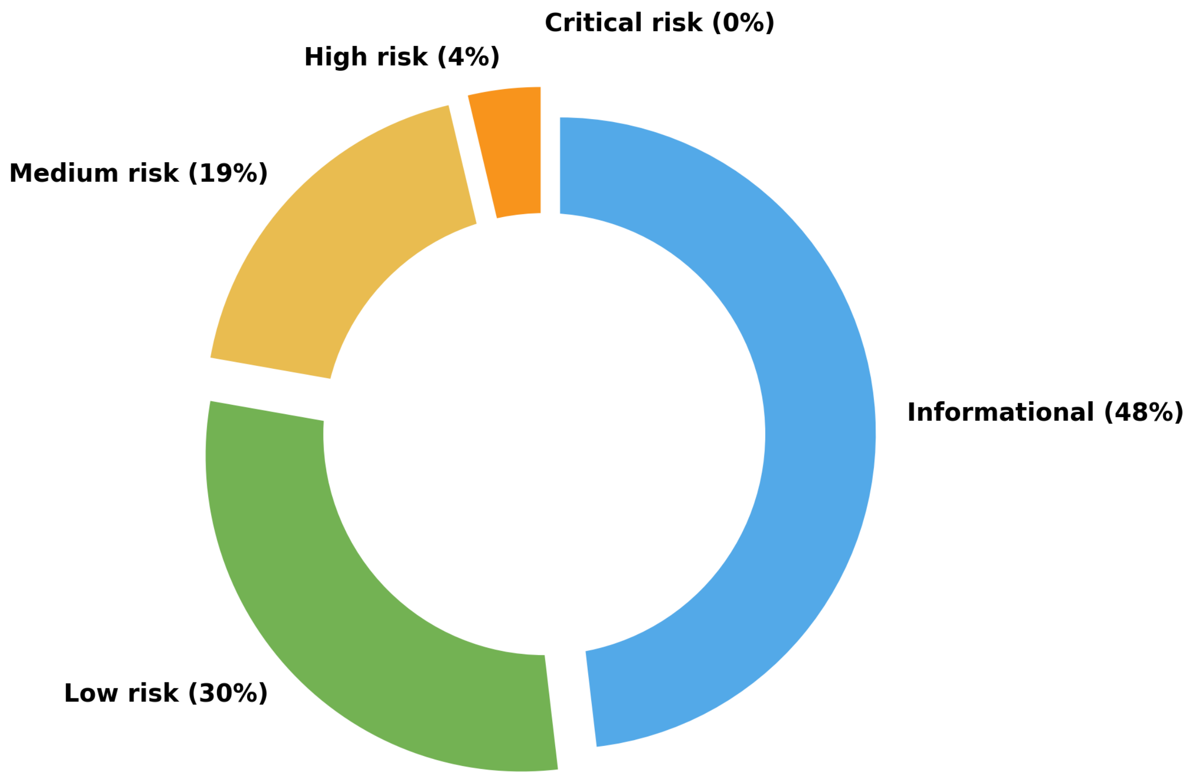
Project Name	Inferno.fun Launchpad
Codebase	https://github.com/LineaLaunchpad/smart-contract
Operating platform	Linea
Programming language	Solidity
Initial commit	`248449d6da58d56ffcd99566cb791b39dc230f7b`
Remediation commit	`09afb10bbe8eef877f7762bb1f94c5b69ed10b8c`
Timeline	From 02.01.2025 to 06.01.2025 (5 days)
Audit methodology	Static analysis and manual review

2.2 Audit Scope

Files and folders in scope
- `src/BondingCurve.sol` - `src/Structures.sol` - `token/Token.sol` - `interfaces/IBondingCurve.sol` - `interfaces/ILynexRouter.sol` - `interfaces/ITokenFactory.sol` - `interfaces/IWETH.sol`

2.3 Summary of Findings

Severity	**Total Found**	**Resolved**	**Partially Resolved**	**Acknowledged**
Critical risk	0	0	0	0
High risk	1	1	0	0
Medium risk	5	4	0	1
Low risk	8	6	0	2
Informational	13	13	0	0



2.4 Findings & Resolutions

ID	Title	Severity	Status
H-01	Hardcoded slippage may block liquidity to AMM migration	High	Resolved
M-01	Multiple creators allowed per signature due to hash collisions	Medium	Resolved
M-02	Migrating liquidity to AMM fails if excess dust ETH is returned	Medium	Resolved
M-03	No incentives for creators to use the launchpad	Medium	Acknowledged
M-04	AMM liquidity migration is blocked if no tokens are available in BondingCurve	Medium	Resolved
M-05	Lynex liquidity pool can be pre-created and imbalance migration	Medium	Resolved
L-01	BondingCurve contract implementation is uninitialized	Low	Resolved

ID	Title	Severity	Status
L-02	Pool deployment is blocked if initialVirtualToken is less than TOTAL_SUPPLY	Low	Acknowledged
L-03	Incorrect deadline when migrating liquidity to AMM	Low	Resolved
L-04	Add contingency in case pool never matures	Low	Acknowledged
L-05	Slight rounding error issues	Low	Resolved
L-06	Missing output amount preview functionality may lead to incorrect user slippage	Low	Resolved
L-07	Launchpad swaps without deadlines	Low	Resolved
L-08	Token deployment signatures do not expire	Low	Resolved
I-01	Improve variable naming	Informational	Resolved
I-02	Improve public functions return type	Informational	Resolved
I-03	Redundant getters over public storage variables	Informational	Resolved
I-04	Reentrancy guard is not initialized	Informational	Resolved
I-05	Add security contact to contracts	Informational	Resolved
I-06	Use a two-step ownership transfer routine	Informational	Resolved
I-07	Cleanup inheritance	Informational	Resolved
I-08	Style improvements and typographical errors	Informational	Resolved
I-09	_safeTransferETH can be simplified	Informational	Resolved
I-10	deployTokenAndCreatePool can be simplified	Informational	Resolved
I-11	General codebase improvements	Informational	Resolved
I-12	Outdated and floating Solidity compiler version	Informational	Resolved
I-13	Tokens mistakenly sent to the contract are blocked	Informational	Resolved

3 Findings

3.1 High Severity Findings

[H-01] Hardcoded slippage may block liquidity to AMM migration

Severity: *High risk* *(Resolved)* *[PoC]*

Context: ***BondingCurve.sol****:211-213*

Description

When the liquidity threshold is reached and the bonding curve pool is migrated to the AMM (Automated Market Maker), the `ILynexRouter::addLiquidityETH` function is called with fixed arguments alongside the last buy that triggered the threshold passing.

```
ILynexRouter(lynexRouter).addLiquidityETH{value: ethAmount}(
    tokenA, false, tokenAmount, tokenAmount, ethAmount, feeReceiver, deadline
);
```

If we consider the `ILynexRouter` interface and interpret the values:

```
interface ILynexRouter {
    function addLiquidityETH(
        address token,
        bool stable,
        uint256 amountTokenDesired,
        uint256 amountTokenMin,
        uint256 amountETHMin,
        address to,
        uint256 deadline
    ) external payable returns (uint256 amountToken, uint256 amountETH, uint256
        liquidity);
```

then adding liquidity to the AMM must fully consume all tokens and all passed Ether otherwise it will revert.

By hardcoding the slippage to 0% (not allowing any), there is a high chance that it will always fail as normal users are able to add liquidity to the original Lynex pool, thus unbalancing the reserves and causing slight slippage to appear, enough to block the liquidity migration.

At an extreme case, the deployment can be attacked to some degree to malicious authors, simply by donating tokens to the reserve (even 1 WEI), each block, can in theory postpone the launch indefinitely (Linea's sequencer logic may make this harder to achieve).

Recommendation

Move the liquidity deployment logic into another function which is callable only by the owner. Have the minimums and (and deadline, but mentioned in another issue) passed to that function.

Resolution: Resolved. A fix was implemented in **PR#36**.

ABA: adding liquidity to the Lynex pools is now done directly, bypassing the Router, as such, this issue does not manifest.

3.2 Medium Severity Findings

[M-01] Multiple creators allowed per signature due to hash collisions

Severity: *Medium risk* *(Resolved)* *[PoC]*

Context: ***BondingCurve.sol****:194*

Description

Deploying a pool requires that a specific creator (function caller) be authorized by the system. This is done through a signature on the following data:

- creator address
- id string, used for uniqueness enforcement by the `BondingCurve` contract
- token name
- token ticker

While the id, token name and token ticker are passed as input alongside the signature itself, the creator is considered the function caller.

These inputs are then verified to have been approved by the signer:

```
require(sigVerify(id, creator, name, ticker, sig), "BondingCurve: Invalid
    signature");
```

The `sigVerify` is, however, vulnerable to hash collisions as it uses `encodePacked` for dynamic types.

```
function sigVerify(
    string memory _id,
    address _creator,
    string memory _name,
    string memory _ticker,
    bytes memory _sig
) internal view returns (bool) {
    bytes32 hash = keccak256(abi.encodePacked(_id, _creator, _name, _ticker));
    return verifier.isValidSignatureNow(hash, _sig);
}
```

By using `encodePacked`, the input data is treated as a compact blob and no type information is embedded in between the variable, such as in the case of `encode`. This creates a risk for hash collisions to appears.

To exemplify the issue, consider the input data from the project test (ID, a random address as creator, name and ticker):

```
"test-launchpad-token", 0xe5D7C2a44FfDDf6b295A15c148167daaAf5Cf34f (20 bytes),
    "Test Launchpad Token", "TLT"
```

The input `keccak256` receives in`sigVerify` from the `encodePacked` is actually the direct byte array (the | indicates where, semantically, there is a different input):

```
74 65 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E | e5 D7 C2 a4 4F fD Df
    6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f | 54 65 73 74 20 4C 61 75 6E 63 68 70 61
    64 20 54 6F 6B 65 6E | 54 4C 54
```

As the input is malleable, several other variations corresponding to the same hash are valid.

Example, by splitting the input as:

```
"t" | 0x6573742D6c61756e63687061642d746f6b656Ee5 |
    hex"D7C2A44FfDDf6b295A15c148167daaAf5Cf34f" "Test Launchpad TokenTLT" | ""

74 | 65 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E e5 | D7 C2 a4 4F fD
    Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f 54 65 73 74 20 4C 61 75 6E 63 68 70
    61 64 20 54 6F 6B 65 6E 54 4C 54 | <empty>
```

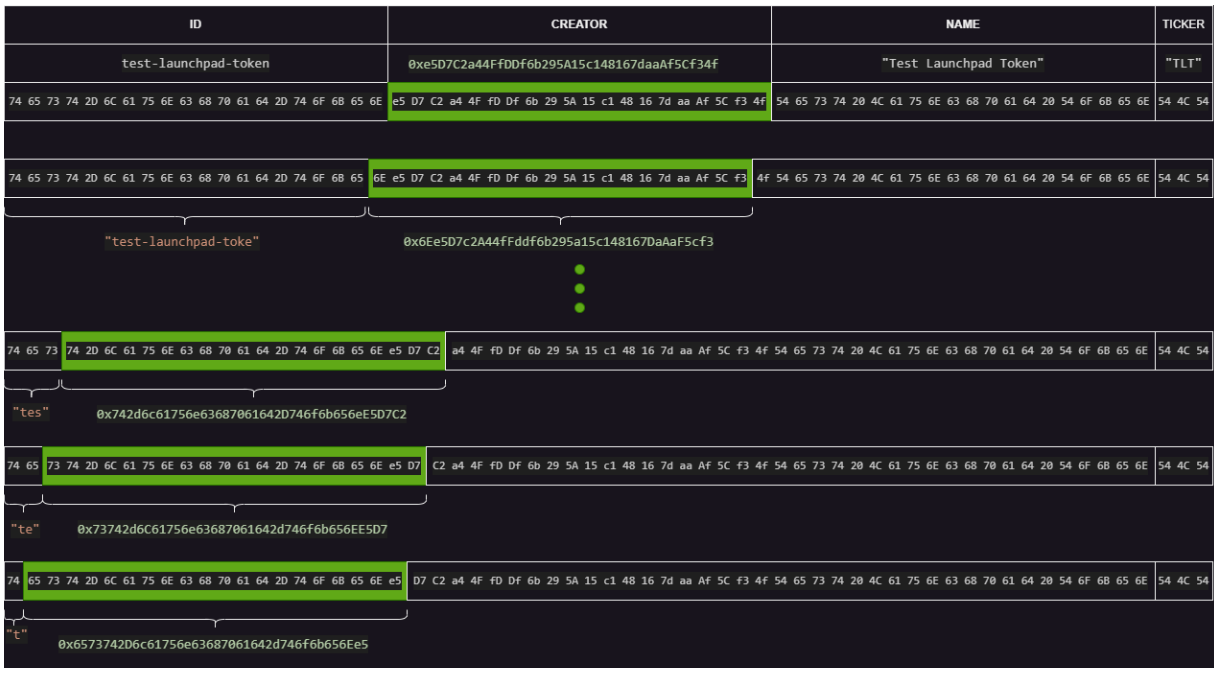
A different caller, can now reuse the same signature, as the resulting input hash is the same.

To be exact, multiple addresses can reuse the same signature. Consider the following observations:

For a given input byte array, the only constraint relies on the caller address, which is at a fixed 20 bytes length. As IDs must be unique, each variation must not overlap the ID with another.

By factoring this, we reach that the total number of valid address that can reuse a given signature is the length of the ID, Name and Ticker combined: `len(ID |Name |Ticker)`. For the example input that translates to 63−20 → 43 different addresses.

The following image shows exact which addresses can be obtained (reused) by sliding the 20 bytes requirement for the creator to the left (sliding to the right is also possible).


ID
CREATOR
NAME
TICKER
test-launchpad-token
0xe5D7C2a44FfDDf6b295A15c148167daaAf5Cf34f
"Test Launchpad Token"
"TLT"
74 65 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E | e5 D7 C2 a4 4F fD Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f | 54 65 73 74 20 4C 61 75 6E 63 68 70 61 64 20 54 6F 6B 65 6E | 54 4C 54
74 65 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 | 6E e5 D7 C2 a4 4F fD Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 | 4f 54 65 73 74 20 4C 61 75 6E 63 68 70 61 64 20 54 6F 6B 65 6E | 54 4C 54
"test-launchpad-toke"
0x6Ee5D7c2A44fFddf6b295a15c148167DaAaF5cf3
74 65 73 | 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E e5 D7 C2 | a4 4F fD Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f 54 65 73 74 20 4C 61 75 6E 63 68 70 61 64 20 54 6F 6B 65 6E | 54 4C 54
"tes"
0x742d6c61756e63687061642D746f6b656eE5D7C2
74 65 | 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E e5 D7 | C2 a4 4F fD Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f 54 65 73 74 20 4C 61 75 6E 63 68 70 61 64 20 54 6F 6B 65 6E | 54 4C 54
"te"
0x73742d6C61756e63687061642d746f6b656EE5D7
74 | 65 73 74 2D 6C 61 75 6E 63 68 70 61 64 2D 74 6F 6B 65 6E e5 | D7 C2 a4 4F fD Df 6b 29 5A 15 c1 48 16 7d aa Af 5C f3 4f 54 65 73 74 20 4C 61 75 6E 63 68 70 61 64 20 54 6F 6B 65 6E | 54 4C 54
"t"
0x6573742D6c61756e63687061642d746f6b656Ee5

The major impact is that, for one signature there can exist multiple callers, which can lead to unwanted deployments.

Another issue, although less severe, is that the protocol verification mechanism may have allowed the creation of a token with a specific Name and ticker, but the creator chooses to split pass it differently.

Example:

	ID	Caller	Name	Ticker
approved by verifier	`"ID#1"`	`<caller>`	`Honey MoneyET`	`"H"`
used by caller	`"ID#1"`	`<caller>`	`Honey Money`	`"ETH"`

This second issue may be abused in generating phishing tokens.

Recommendation

Use `abi.encode` instead of `abi.encodePacked` when generating and validation the signature.

Resolution: Resolved. A fix was implemented in **PR#14**.

Inferno.fun: Added encode change. Also added Ethereum signed message prefix.

[M-02] Migrating liquidity to AMM fails if excess dust ETH is returned

Severity: *Medium risk* *(Resolved)*

Context: ***BondingCurve.sol****:389-391*

Description

When the liquidity threshold is reached and the bonding curve pool is migrated to the AMM (Automated Market Maker), the `ILynexRouter::addLiquidityETH` function is called.

The add liquidity call, in some cases, will return any leftover native ETH to the caller, in this case the `BondingCurve` contract

```
// refund dust ETH, if any
if (msg.value > amountETH) _safeTransferETH(msg.sender, msg.value - amountETH);
```

expecting the transfer to be successfully received.

```
function _safeTransferETH(address to, uint value) internal {
    (bool success,) = to.call{value:value}(new bytes(0));
    require(success, 'TransferHelper: ETH_TRANSFER_FAILED');
}
```

However, by returning native ETH, the operation will revert as the `BondingCurve` contract allows only the WETH implementation to supply native ETH

```
receive() external payable onlyWETH {
    emit Received(msg.sender, msg.value);
}
```

This results in failed deployments, requiring different amounts to be passed and retried until adding liquidity does not return any dust.

Recommendation

Ensure that the AMM router is also allowed to send native ETH back to the contract.

If there is ETH that was sent back, it will be lost in the `BondingCurve` contract. Compare the `ILynexRouter::addLiquidityETH` returned ETH amount that was used (the `amountETH` value) with the `ethAmount` that was sent to it. If there is a difference, send it to the fee receiver.

Resolution: Resolved. A fix was implemented in **PR#36**.

ABA: adding liquidity to the Lynex pools is now done directly, bypassing the Router, as such, this issue does not manifest.

[M-03] No incentives for creators to use the launchpad

> **Severity:** ***Medium risk*** *(Acknowledged)*
>
> **Context:** ***BondingCurve.sol****:212*

Description

When the liquidity threshold is reached and the bonding curve pool is migrated to the AMM (Automated Market Maker), the LPs from adding the tokens is kept by the protocol fee receiver.

This design severely disincentives any creator from using the launchpad as they have no advantage from a regular user.

Recommendation

Instead of sending the entire LP tokens to the team `feeReceiver`, split it into two parts, one that goes to the protocol `creator` and one that goes to the team `feeReceiver`.

More options can be included, such as burning the LPs if the creator decides when creating the pool. This gives more flexibility and enhances attractiveness, as market participants would know that upon deployment to the AMM, a part of LPs are forever burned/locked.

Resolution: Acknowledged by the team.

[M-04] AMM liquidity migration is blocked if no tokens are available in BondingCurve

> **Severity:** ***Medium risk*** *(Resolved) [PoC]*
>
> **Context:** ***BondingCurve.sol****:216-288*

Description Launching a token pair on the AMM using the `BondingCurve::launchOnDex` function may revert if the provided liquidity does not satisfy a minimum threshold.

Currently, BondingCurve will attempt to launch the token pair on the AMM as soon as the targetEthThreshold is reached:

```
if ((virtualTokenB - pool.additionalTokenB) >= pool.targetEthThreshold) {
    launchOnDex(token, virtualTokenA - pool.additionalTokenA, virtualTokenB -
        pool.additionalTokenB);
```

In the lynexRouter::addLiquidityETH function, liquidity provider tokens are minted by calling the Pair::mint function. For the minting call to succeed, the calculated liquidity must be greater than 0:

```
function mint(address to) external lock returns (uint liquidity) {
    (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
    uint _balance0 = IERC20(token0).balanceOf(address(this));
    uint _balance1 = IERC20(token1).balanceOf(address(this));
    uint _amount0 = _balance0 - _reserve0;
    uint _amount1 = _balance1 - _reserve1;

    uint _totalSupply = totalSupply; // gas savings, must be defined here since
        totalSupply can update in _mintFee
    if (_totalSupply == 0) {
>>      liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
            MINIMUM_LIQUIDITY tokens
    } else {
>>      liquidity = Math.min(_amount0 * _totalSupply / _reserve0, _amount1 *
    _totalSupply / _reserve1);
    }
>>   require(liquidity > 0, 'ILM'); // Pair: INSUFFICIENT_LIQUIDITY_MINTED
    _mint(to, liquidity);

    _update(_balance0, _balance1, _reserve0, _reserve1);
    emit Mint(msg.sender, _amount0, _amount1);
}
```

As the token will not be launched on the AMM by a third party (since transfers will be disabled), the relation that must hold true for Pair::mint to be successful is Math.sqrt(_amount0 * _amount1) > MINIMUM_LIQUIDITY. This is equivalent to tokenAmount * ethAmount > 10e6 in BondingCurve::launchOnDex.

However, the BondingCurve protocol does not ensure that the tokenAmount and ethAmount provided for BondingCurve::launchOnDex are sufficient to generate the required liquidity. This may lead to the liquidity being unable to be added to the AMM.

Another consideration here is that, users that wish to buy more tokens that exist will have the execution reverted with an "ERC20: transfer amount exceeds balance" error, but simply capping the amount to be bought to the maximum available Token balance will then result in the other issues, of not being able to move liquidity due to the Pair::mint underflow.

Recommendation

Ensure the BondingCurve contract keeps hold of enough liquidity to successfully launch the token pair to the AMM. This can be done as:

- ensure that `targetEthThreshold` is always greater then `1e6`, for good measure have it at `1e7`
- ensure that in the `BondingCurve` contract there will always exist at least 1 wei of `Token` accounted for.

If buys that go over the `TOTAL_SUPPLY` are to be capped, the cap should also leave out 1 wei of Tokens.

Resolution: Resolved, the recommended fix was implemented in **PR#31**.

[M-05] Lynex liquidity pool can be pre-created and imbalance migration

Severity: ***Medium risk*** *(Resolved) [PoC]*

Context: ***Token.sol BondingCurve.sol***

Description

The `BondingCurve` contract's role is to ensure users create tokens that trade according to a specific bonding curve while being paired with virtual ETH liquidity. When real liquidity reaches a certain threshold, the pool is migrated to an AMM (`Lynex`).

The current design allows users to buy the tokens from the launchpad and directly make liquidity pools on any DEX, including the Lynex pool. Allowing users to make other pools is a design decision with advantages and disadvantages and the team has decided to allow the creation of other pools.

However, an issue can appear regarding the Lynex pools, where liquidity will be migrated at maturity. The Lynex liquidity pool may already exist and be severely imbalanced so that, when the migration happens, the paired token price post-bonding is not respected, users may experience severe price volatility during the process.

Recommendation

Do not allow the creation of a Lynex liquidity pool with the `Token/WETH` pair.

This can be achieved by modifying the `Token` contract so that no transfers are allowed to the `Token/WETH` DEX Pair address. The pair address can be determined deterministically via the `IPairFactory::getPair` function, as Lynex is an Uniswap-v2 fork swap.

```
    address pairFactory = ILynexRouter(lynexRouter).factory();
    address pair =  IPairFactory(pairFactory).getPair(tokenA, address(weth), false);
```

Have the pair address blocked for transfers from the `Token` contract itself (from and to) up until the bounding matures, at which point the internal launchpad token pair prices relationship will be kept after migration.

Adding the token transfer constraint, however, introduces an issue.

If the `ILynexRouter::addLiquidityETH` is used, then an attacker can directly send WETH to the underlying pool pair and call `Pair::sync`. By doing this, adding liquidity through the router will fail in `quoteLiquidity` with the error `BaseV1Router: INSUFFICIENT_LIQUIDITY`.

```
// given some amount of an asset and pair reserves, returns an equivalent amount of
    the other asset
function quoteLiquidity(uint amountA, uint reserveA, uint reserveB) internal pure
    returns (uint amountB) {
    require(amountA > 0, 'BaseV1Router: INSUFFICIENT_AMOUNT');
    require(reserveA > 0 && reserveB > 0, 'BaseV1Router: INSUFFICIENT_LIQUIDITY');
    amountB = amountA * reserveB / reserveA;
}
```

Note: The contract links provided are for illustrative purposes and refer to a random Lynex pool pair and the Lynex router.

This attack is inherently doable at any tine for any give Uniswap V2 type pair and also possible when the transfers were allowed, but mitigating it at that point would have been for anyone to also transfer the missing pair to call `sync`. By blocking token transfers to the pair pre-bonding, this option is no longer available and would lead to tokens being blocked in the `BondingCurve` contract.

Thus, after modifying the `Token` contract, also change from using `ILynexRouter`'s `addLiquidityETH` function to directly interacting with the underlying pair. Example implementation:

```
function _customAddLiquidity(address tokenA, uint256 tokenAmount, uint256
    ethAmount) internal {
    address pairFactory = ILynexRouter(lynexRouter).factory();
    address pair =  IPairFactory(pairFactory).getPair(tokenA, address(weth), false);
    if (pair == address(0)) {
        pair = IPairFactory(pairFactory).createPair(tokenA, address(weth), false);
    }

    weth.deposit{value: ethAmount }();
    weth.safeTransfer(pair, ethAmount);
    IERC20(tokenA).safeTransfer(pair, tokenAmount);
    IPair(pair).mint(feeReceiver);
}
```

There are other advantages of implementing the fix as such:

- it is simple and straightforward
- no need for deadline or minimum WETH/Token outputs, since an attacker can only increase the WETH pair reserve, meaning that, when migrating liquidity, the valuation of the launchpad token will be slightly higher, which is good. This implicitly also resolves the *Hardcoded slippage may block liquidity to AMM migration* and *Incorrect deadline when migrating liquidity to AMM* issues.

Resolution: Resolved, the recommended fix was implemented in **PR#36**.

3.3 Low Severity Findings

[L-01] BondingCurve contract implementation is uninitialized

Severity: ***Low risk*** *(Resolved)*

Context: ***BondingCurve.sol***

Description

As described by OpenZeppelin documentation

> An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy.

The `BondingCurve` contract is missing the call to `_disableInitializers`. Anyone can call the `initialize` function on the implementation itself and work with it.

No damage will be done to the actual official contract, which is through the proxy, but this allows attackers to potentially phish users by leveraging the compromised, official implementation.

Recommendation

To prevent the implementation contract from being used, you should invoke the `_disableInitializers` function in the constructor to automatically lock it when it is deployed:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Resolution: Resolved, the recommended fix was implemented in **PR#27**.

[L-02] Pool deployment is blocked if initialVirtualToken is less than TOTAL_SUPPLY

Severity: ***Low risk*** *(Acknowledged)*

Context: ***BondingCurve.sol****:358*

Description

When a token and pool are deployed via `BondingCurve::deployTokenAndCreatePool`:

- the contract first deploys the token, which in turn mints the `TOTAL_SUPPLY` to it

```
Token token = new Token(name, ticker, TOTAL_SUPPLY);
```

- the token balance is then saved in tokenBalance. This is actually the same as TOTAL_SUPPLY, since the current Token implementation mints all tokens to the bonding curve contract

```
uint256 tokenBalance = token.balanceOf(address(this));
```

- now, when saving the pool information, the additionalTokenA field is saved as the difference between the virtualTokenA (which is initialVirtualToken) and the token balance

```
poolDetails[tokenAddress] = Pool({
//  ...
    additionalTokenA: virtualTokenA - tokenBalance,
//  ...
});
```

This subtraction can underflow and revert it the token balance, meaning TOTAL_SUPPLY, is ever larger than the initial virtual token amount initialVirtualToken, blocking all pool deployments until the value is changed via BondingCurve::configure.

Recommendation

When initializing the contract or when subsequently calling BondingCurve::configure, ensure that the initialVirtualToken is greater than TOTAL_SUPPLY.

Resolution: Acknowledged by the team.

Inferno.fun: There are other restrictions too for the virtual token and virtual eth which is kept out of scope for this contract. The numbers will be calculated off chain and provided to the contract.

[L-03] Incorrect deadline when migrating liquidity to AMM

> **Severity:** *Low risk* *(Resolved)*
>
> **Context:** *BondingCurve.sol:210*

Description

When a pool has reached maturity and the liquidity is moved to the Lynex liquidity pools, the ILynexRouter::addLiquidityETH function is called with appropriate parameters.

The deadline that is used in the function is wrongly dependent on the block.timestamp.

```
uint256 deadline = block.timestamp + 5 minutes;
```

By setting it as so, it behaves as if no deadline was passed, because block.timestamp represents the exact time this transaction is being executed. This includes the exact time adding liquidity is also checking for freshness. It is equivalent to comparing:

```
require((block.timestamp + 5 minutes) >= block.timestamp, 'BaseV1Router: EXPIRED');
```

which is always true.

Lack of a deadline can result in the transaction being executed at a later date when ETH denomination is not at an intended value.

Consider the following scenario:

- `alice` wants to buy $3000 worth of tokens, which would be the price of 1 ETH at that time
- `alice`'s buy is enough to mature the pool and have it migrated to the AMM (Automated Market Maker)
- `alice` initiates a buy of 1 ETH worth of tokens, but the transaction is not taken by any miner and remains in the mempool
- at a later date, the transaction is picked up and executed but when the price of ETH has spiked to say $3300
- now, `alice` has ended up paying more in actual value than she intended, causing indirectly a loss for her

Recommendation

Separate the AMM migration logic (the `launchOnDex` call) from the token buy logic to its own function, permissioned so that only the owner can call it. Buying and selling of the specific token would be deactivated. Have the deadline be transmitted as a parameter in this function.

Resolution: Resolved. A fix was implemented in **PR#36**.

ABA: adding liquidity to the Lynex pools is now done directly, bypassing the Router, as such, this issue does not manifest.

[L-04] Add contingency in case pool never matures

> **Severity:** ***Low risk*** *(Acknowledged)*
>
> **Context:** ***BondingCurve.sol***

Description

A pool will be migrated to an AMM only if a certain liquidity threshold has been reached.

However, if the threshold will not be reached in the foreseeable future, due to extreme market conditions, then the only option for the protocol to release the funds to an AMM is to lower the liquidity threshold. But this cannot be easily done, since it is global and not per launched token.

Recommendation

On each token creation add expiration date, after which, the owner of the `BondingCurve` may launch the token even if the minimum threshold has not been reached.

Resolution: Acknowledged by the team.

[L-05] Slight rounding error issues

Severity: *Low risk* *(Resolved)*

Context: *BondingCurve.sol:237,270,240,267*

Description

There are instances in the `BondingCurve` contract where rounding errors appear:

1. When calculating the fees for buys and sells:

```
if (isBuy) {
//  ...
    uint256 fee = (swapFee * inputAmount) / POINTS;
//  ...
} else {
//  ...
    uint256 fee = (swapFee * actualOutputAmount) / POINTS;
//  ...
}
```

This rounding is against the protocol, meaning the protocol loses tokens.

In this case, users will not pay fees if they are trading dust amounts. Example for with 99 wei of ETH tokens, at a 10% fee rate, they save on paying $0.000000000000036009072, which is insignificant.

1. When calculating the output amounts

```
if (isBuy) {
//  ...
    actualOutputAmount = (pool.virtualTokenA * swapAmount) / (pool.virtualTokenB +
        swapAmount);
//  ...
} else {
//  ...
    actualOutputAmount = (pool.virtualTokenB * inputAmount) / (pool.virtualTokenA +
        inputAmount);
//  ...
}
```

This rounding is against the user, which is an accepted type of rounding error, that appears in popular UniswapV2-like protocols such as the Lynux pools and Uniswap V2.

But the slight loss due to rounding affects the K constant. Again, this is something that happens in normal `k=x*y` AMM.

Recommendation

The fee rounding issue can be resolved by always rounding up the amount. After each swap, the k constant can be recalculated to give a better external overview of the state of the pool. Example implementation:

```diff
    // update pool details
    pool.virtualTokenA = virtualTokenA;
    pool.virtualTokenB = virtualTokenB;
+   pool.k = virtualTokenA * virtualTokenB;
```

Resolution: Resolved, the recommended fix was implemented in **PR#28**.

[L-06] Missing output amount preview functionality may lead to incorrect user slippage

> **Severity:** ***Low risk*** *(Resolved)*
>
> **Context:** ***BondingCurve.sol***

Description*

The functions from theBondingCurve contract:buy, `buyWithWETH` and `sell` require users to provide an `outputAmount` parameter for slippage protection.

However, the protocol does not provide a built-in mechanism to estimate the expected `outputAmount`.

Users must analyze the token pool's state, including reserves and protocol fees, and reverse-engineer the bonding curve formula to calculate the expected ETH or token amount. This process is both complex and time-intensive, as pool conditions can rapidly change.

Without accurate estimation, users may set inappropriate slippage limits, resulting in failed transactions or in receiving less value than intended from a swap.

Recommendation

Implement functionality to estimate the output amount for a given input. Fees must also be taken into consideration in these "preview" type functions.

Resolution: Resolved, the recommended fix was implemented in **PR#30**.

[L-07] Launchpad swaps without deadlines

> **Severity:** ***Low risk*** *(Resolved)*
>
> **Context:** ***BondingCurve.sol****:377-387*

Description

When users buy or sell launchpad tokens from the `BondingCurve` contract, they call either one of the following functions buy, `buyWithWETH` or `sell`.

These functions have a minimum output token amount (slippage), which is good, but are lacking a deadline option.

Deadlines allow users to specify the time when the swap must be executed otherwise it would be invalid.

Lack of a deadline can result in the transaction being executed at a later date when Token/WETH denomination is not at an intended value.

Recommendation

Add a deadline parameter to the `buy`, `buyWithWETH` and `sell` functions, which is verified in `handleSwap`.

Resolution: Resolved, the recommended fix was implemented in **PR#34**.

[L-08] Token deployment signatures do not expire

Severity: *Low risk* *(Resolved)*

Context: ***BondingCurve.sol****:377-387*

Description

Deploying a pool requires that a specific creator be authorized by the system. This is done through a signature on specific collection data, such as token name and symbol. These inputs are then verified to have been approved by the signer:

```
require(sigVerify(id, creator, name, ticker, sig), "BondingCurve: Invalid
    signature");
```

As it is, there is no deadline for submitting the deploy token themselves. A creator may be authorized to deploy a token now, but nothing is stopping him in submitting the deploy token signature at a later time, when it would be inconvenient for the protocol.

Recommendation

Add a deadline in the encoded data signature and verify it before deploying the token, in the `deployTokenAndCreatePool` function.

Resolution: Resolved, the recommended fix was implemented in **PR#37**.

3.4 Informational Findings

[I-01] Improve variable naming

Severity: *Informational* *(Resolved)*

Context: ***BondingCurve.sol***:*75,216,377,381,385*

Description

In the `BondingCurve` contract, there are several variables that can have better names, that would be closer to how they are used:

- `BondingCurve.sol#L75`: the `POINTS` variable refers to an equivalent of 100% in basis points, as such can be changed to `BASIS_POINTS`.
- the `outputAmount` in the `handleSwap`, `buy`, `buyWithWETH` and `sell` functions is actually the minimum out amount to be allowed, a slippage equivalent. A more suggestive name would be `minimumOutputAmount`.

Recommendation

Change the name of the mentioned variables as indicated.

Resolution: Resolved, the recommended fix was implemented in **PR#15**.

[I-02] Improve public functions return type

Severity: *Informational* *(Resolved)*

Context: ***BondingCurve.sol***:*334-375,377-387*

Description

There are several public functions in the `BondingCurve` contract that do not return any type, while there would be relevant information to be returned for any third party integrator.

Recommendation

The `deployTokenAndCreatePool` function can return the newly-deployed token address, and output tokens, in case any integrator wishes to use that information.

The `buy`, `buyWithWETH` and `sell` should return the output token amount from the `handleSwap` call, so that any integrator knows the exact amount they should have received

Resolution: Resolved, the recommended fix was implemented in **PR#16**.

[I-03] Redundant getters over public storage variables

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***BondingCurve.sol****:167-173*

Description

In the `BondingCurve` contract, there are two functions `getVirtualToken` and `getVirtualReserve`, which are wrapper getters over two public storage variables.

The Solidity compiler already automatically creates getters for any `public` storage variables. making the `getVirtualToken` and `getVirtualReserve` functions redundant in terms of usability.

Recommendation

Make the underlying `initialVirtualToken` and `initialVirtualReserve` variables private, to reduce the code size of the contract and ensure only the intended getters are used.

Resolution: Resolved, the recommended fix was implemented in **PR#17**.

[I-04] Reentrancy guard is not initialized

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***BondingCurve.sol****:105-158*

Description

As the `BondingCurve` contract is an upgradable contract, the `initialize` function is the first to be called. In it, each inherited contract initializer must be called.

The `BondingCurve` contract inherits from the `ReentrancyGuardUpgradeable` abstract contract but its initializer, `__ReentrancyGuard_init`, is not called.

The `ReentrancyGuardUpgradeable` implementation, is not affected by this, as the `__ReentrancyGuard_init` call sets the contract to `NOT_ENTERED` (value `1`), where the default value is `0`, which does not impact functionality since `ENTERED` is `2`.

Recommendation

For uniformity, consider adding the `__ReentrancyGuard_init` call in the `initialize` function.

Resolution: Resolved, the recommended fix was implemented in **PR#18**.

[I-05] Add security contact to contracts

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***BondingCurve.sol Token.sol***

Description

In case a whitehat identifies an issue with the on-chain contracts of the protocol, to more easily be able to contact the team, have a security contact added to the contracts.

Recommendation

Add the email address for the team security contact either as a plain comment or as a

`custom:security-contact` natspec tag to the contracts.

Resolution: Resolved, the recommended fix was implemented in **PR#29**.

[I-06] Use a two-step ownership transfer routine

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***BondingCurve.sol***

Description

When transferring ownership of a contract, mistakes that transfer the ownership to an unwarned address can be avoided by using a two-step transfer routine.

In the first step a new owner is proposed and in the second step the new owner must accept the ownership.

Recommendation

Use OpenZeppelin's 2-step ownership transfer contract in the `BondingCurve` contract

Resolution: Resolved, the recommended fix was implemented in **PR#19**.

[I-07] Cleanup inheritance

> **Severity:** ***Informational*** *(Resolved)*
>
> **Context:** ***BondingCurve.sol****:5,13,19*

Description

`BondingCurve` currently inherits `Structures`, `Initializable`, `OwnableUpgradeable`, `ReentrancyGuardUpgradeable`.

This inheritance chain, and subsequent imports, can be simplified:

- in the `BondingCurve.sol` file itself, there is the `IBondingCurve` interface, which inherits from the `Structures` interface, it is unused
  - remove the `Structures` inheritance and import, and the use the `IBondingCurve` as
- `BondingCurve` inherits from `Initializable` but so does `OwnableUpgradeable` and `ReentrancyGuardUpgradeable`, thus it is redundant to also inherit it again

  - remove the `Initializable` inheritance and import altogether

Recommendation

Remove the `Structures` and `Initializable` inheritance and import and replace them with the mentioned counterparts.

Resolution: Resolved, the recommended fix was implemented in **PR#20**.

[I-08] Style improvements and typographical error

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***BondingCurve.sol***:*23-29,51-55,60-64*

Description

In the `BondingCurve` contract, events do not respect standard Solidity style writing and there is one instance of a typo.

- the `onInitialMarketCapUdpated` event name should be `onInitialMarketCapUpdated`
- the arguments for the `onSwapFeeUpdated` event `oldswapFee` and `newswapFee` should be mixed-case, as: `oldSwapFee` and `newSwapFee`
- all events names should be `CapWords` style, not start with a lowercase letter. This applies to all events starting with `on*` (e.g. `onVerifierUpdated`, `onTokenFactoryUpdated`, etc.). Either make the first letter uppercase or remove the `"on"` initial part, as semantically, events are always trigger **on-an-event**.
- all internal or private functions should start with an underline _

Recommendation

Implemented the mentioned slight style change and fix the minor typo.

Resolution: Resolved, the recommended fix was implemented in **PR#21**.

[I-09] _safeTransferETH can be simplified

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***BondingCurve.sol***:*198-201*

Description

The `BondingCurve::_safeTransferETH` function is responsible for sending native ETH to an indicated token address. There are two improvements that can be made to the function, both for uniformity:

- the `new bytes(0)` declaration can be rewritten as `""`

- the `require` error message `"TransferHelper: ETH_TRANSFER_FAILED"` is incorrect as it mentions a `TransferHelper` component, while all other logs mention `BondingCurve`.

Recommendation

Replace `new bytes(0)` with `""` and `TransferHelper` with `BondingCurve`.

Resolution: Resolved, the recommended fix was implemented in **PR#22**.

[I-10] deployTokenAndCreatePool can be simplified

Severity: ***Informational*** *(Resolved)*

Context: ***BondingCurve.sol****:334-375*

Description

The `BondingCurve::deployTokenAndCreatePool` function has several slight improvements that can be done to reduce execution cost:

1. move all circuit breakers as close to each other as possible to ensure a timely exist and reduce revert gas costs
    1. move `address creator = msg.sender;` logic exactly before the `sigVerify` requirement
    2. move `!tokenInfo[id].isCreated` check to be first in the function
2. use direct inner attribution for storage variables, instead of going through getters
    1. instead of using `getVirtualToken` to set `virtualTokenA`, directly set `virtualTokenA` to `initialVirtualToken`
    2. instead of using `getVirtualReserve` to set `virtualTokenB`, directly set `virtualTokenB` to `initialVirtualReserve`
3. the variable k is used only in one place
    1. remove the `uint256 k = virtualTokenA * virtualTokenB;` declaration and directly use `virtualTokenA * virtualTokenB` when creating the pool

Recommendation

Implement the mentioned changes to slightly reduce execution costs and increase code readability.

Resolution: Resolved, the recommended fix was implemented in **PR#23**.

[I-11] General codebase improvements

Severity: ***Informational*** *(Resolved)*

> **Context:** *BondingCurve.sol:381* *ITokenFactory.sol*

Description

There are some miscellaneous light improvements that can be done to the codebase as it is:

1. more Natspec can be added overall, to better document functionality for integrators
2. the `src\Structures.sol` file has the `Structure` interface. Rename the file to `IStructures` and move it in the `src\interfaces` folder
3. in the `BondingCurve` contract
    1. remove the outdate `onTokenFactoryUpdated` event
    2. remove the `initialMarketCap` variable and linking events as it is not used
    3. change `buyWithWETH`'s visibility to `external`, as no internal call is needed, plus a slight gas improvement is received
4. remove unused `interfaces\ITokenFactory.sol` file
5. add `TOTAL_SUPPLY` public getter function to the `IBondingCUrve` interface

Recommendation

Implement the mentioned codebase improvements.

Resolution: Resolved, the recommended fix was implemented in **PR#26**.

[I-12] Outdated and floating Solidity compiler version

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** Global

Description

Contracts should be deployed with the same compiler version used during development and testing. Locking the pragma version ensures this, preventing bugs from outdated or untested compiler versions.

The current codebase has a floating pragma for all contracts of `pragma solidity ^0.8.15;`

Recommendation

Use a newer and fixed pragma version in the codebase.

Resolution: Resolved, the recommended fix was implemented in **PR#25**.

[I-13] Tokens mistakenly sent to the contract are blocked

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** *BondingCurve.sol*

Description

If, by mistake, a user directly sends any of the deployed tokens back to the `BondingCurve` contract, they are blocked, as there is no mechanism for retrieving stranded tokens.

Recommendation

Create an `emergencyWithdrawal` function, admin callable only, which given a token, it will withdraw *only the difference between what is already accounted in the `Pool` structure for the respective token and balance of the contract*. As all buys and sells are accounted within the contract, any direct transfer will not register and be unaccounted for.

The functionality may be also added at a later time, with an upgrade, if the need ever arises. If that is the desired behavior, acknowledge this issue.

Resolution: Resolved, the recommended fix was implemented in **PR#38**.

4 Inferno.fun Launchpad Deployed Tokens Audit

The following section covers the tokens that will be deployed through the Inferno.fun Launchpad from a **Token Audit** perspective, focusing on areas where other tools such as De.Fi Scanner or GoPlus Token Security, which are used by GekoTerminal, or DEX Screener's service, might show inconsistencies or false positives.

The verdict section is split into two columns due to how a bonding curve contract system works: the verdict before liquidity migration to the Lynex AMM (Automatic Market Maker) and after.

To elaborate, the Inferno.fun Launchpad utilizes a bounding curve contract of `xy=k`, similar to how UniswapV2-like forks (including Lynex) work. At first, users will only be able to buy and sell tokens off and to the Inferno.fun Launchpad contract (`BondingCurve`), until the minimum liquidity threshold is reached (the threshold can be changed by the team). Transferring tokens between users is allowed but creating the WETH pair on Lynex is not permitted until migration.

When the minimum liquidity threshold is reached, existing liquidity (`WETH`) from the Inferno.fun Launchpad contract is migrated to the Lynex liquidity pools and the token is changed to behave as any other. Buy/sell fees are removed and the launchpad will not be able to influence it any further.

Because of the above mechanism, several alerts and red flags will most likely be raised by automated token audit services, of which the following are detailed:

Note: this report is not, nor should be considered, an "endorsement" or "disapproval" of the project or team, nor should be considered, an indication of the economics or value of any "product" or "asset" created by the team. This report should not be used in any way to make decisions around investment or involvement with the Inferno.fun project. Please red the Disclaimer section for more details.

Issue	**Description**	**Initial Verdict**	**After Migration Verdict**	**Observations**
Buy Tax	Taxes incurred when buying this token	Yes	No	Up until liquidity migration, token sale has a buy tax and the token can be bought only from the Inferno.fun Launchpad contract. After the migration, the tax is removed

Issue	**Description**	**Initial Verdict**	**After Migration Verdict**	**Observations**
Can't be Bought	If the token can be bought	Possibly Yes	No	Up until liquidity migration, the token can initially only be bought from the Inferno.fun Launchpad contract itself, which is expected behavior for a bound token. Users can buy it and create pools on popular exchanges during bounding, except the WETH pair on Lynex. This behavior may trigger the mentioned flag
In main DEX	If the token can be traded on the main blockchain DEX	Possibly No	Yes	Up until liquidity migration, token will not be tradeable on the Lynea DEX as a WETH pair. This may trigger this flag for automation tools. After migration, it will be tradeable on any DEX and pool pair
Sell Tax	Taxes incurred when selling this token	Yes	No	Up until liquidity migration, there is a sell tax and the token can be sold mainly to the Inferno.fun Launchpad contract. After the initial sell period is done and liquidity is move to Lynex liquidity pools, the tax is removed
Buy/Sell Tax Modifiable	If the buy/sell tax can be changed	Yes	No	The buy and sell tax are the same and can be changed by the owner of the Inferno.fun Launchpad contract. This applies only until liquidity migration, afterwards there wil be no buy or sell tax
Recently Deployed Contract	If the contract has been recently deployed	Yes	No	This will, evidently, be true after the contract will be deployed up until the contract is old enough to not trigger this flag. Each automate system defines his own time-frame for what "recently" implies (e.g. 2 weeks old, 1-week-old)
Incorrect Solidity Version	Contracts should be deployed with newer versions of the compiler	Possibly Yes	Possibly Yes	The Linea blockchain supports at most Solidity version 0.8.19 which may not be accounted by the automated systems

5 Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts the consultant to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. The consultant’s position is that each company and individual are responsible for their own due diligence and continuous security. The consultant’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology that is analyzed.

The assessment services provided by the consultant is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. Furthermore, because a single assessment can never be considered comprehensive, multiple independent assessments paired with a bug bounty program are always recommend.

For each finding, the consultant provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved, but they may not be tested or functional code. These recommendations are not exhaustive, and the clients are encouraged to consider them as a starting point for further discussion.

The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties. Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, the consultant does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

The consultant retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. The consultant is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. The consultant is furthermore allowed to claim bug bounties from third-parties while doing so.