Three Sigma

Code Audit

BuzzFun Token Launcher

Disclaimer

Code Audit

BuzzFun Token Launcher

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

BuzzFun Token Launcher

Table of Contents

Summary

Code Audit

BuzzFun Token Launcher

Summary

Three Sigma audited BuzzFun in a 2 person week engagement. The audit was conducted from 20-01-2025 to 24-01-2025.

Protocol Description

Buzz is a decentralized platform revolutionizing token creation and trading on Abstract chain and Ethereum mainnet. With a custom contract deployer system, Buzz enables seamless token launches, tailored features, and dynamic growth incentives. Designed to drive sustainable growth and long-term value, Buzz rewards creators and traders through innovative tokenomics, optimized bonding curves, and a robust XP rewards system that incentivizes late-stage purchases and engagement.

Buzz includes a powerful live chat system with advanced community management features, real-time engagement, and moderation tools. It bridges the gap between token creation and active community building, helping users discover vibrant communities and track trending narratives.

Scope

Code Audit

BuzzFun Token Launcher

Scope

Filepath	nSLOC
BondingCurve.sol	202
BeastTokenContract.sol	847
RetardedTokenContract.sol	724

Assumptions

OpenZeppelin is considered secure.

Methodology

Code Audit

BuzzFun Token Launcher

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

BuzzFun Token Launcher

Project Dashboard

Application Summary

Name	BuzzFun
Commit	77bce53
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	20-01-2025 to 24-01-2025
№ of Auditors	2
Review Time	2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	3	3	0
Medium	3	3	0
Low	4	3	1

None	10	8	2

Category Breakdown

Suggestion	3
Documentation	0
Bug	11
Optimization	1
Good Code Practices	6

Code Maturity Evaluation

Code Audit

BuzzFun Token Launcher

Code Maturity Evaluation

Code Maturity Evaluation Guidelines

Category	Evaluation
Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system.
Arithmetic	The proper use of mathematical operations and semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Code Stability	The extent to which the code was altered during the audit.
Upgradeability	The presence of parameterizations of the system that allow modifications after deployment.
Function Composition	The functions are generally small and have clear purposes.
Front-Running	The system's resistance to front-running attacks.
Monitoring	All operations that change the state of the system emit events, making it simple to monitor the state of the system. These events need to be correctly emitted.
Specification	The presence of comprehensive and readable codebase documentation.
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage.

Code Maturity Evaluation Results

Category	Evaluation
Access Controls	Satisfactory. The codebase has a strong access control mechanism.
Arithmetic	Satisfactory. No arithmetic errors were found.
Centralization	Satisfactory. The owner has control over a marginal functionality, but most of the design is decentralized.
Code Stability	Satisfactory. The code was stable during the audit.
Upgradeability	Satisfactory. The contracts are not upgradeable.
Function Composition	Satisfactory. Functionality is well split into helper functions.
Front-Running	Moderate. A low likelihood front-running issue was found.
Monitoring	Satisfactory. Events were correctly emitted.
Specification	Moderate. High-level specification was missing, codebase documentation was limited.
Testing and Verification	Weak. The development environment was not configured.

Findings

Code Audit

BuzzFun Token Launcher

Findings

3S-BuzzFun-C01

The granularity of 1M tokens for buy and sell operations in the **`BondingCurve`** contract leads to user fund losses

Id	3S-BuzzFun-C01
Classification	Critical
Severity	Critical
Likelihood	High
Category	Bug
Status	Addressed in #cec834c.

Description

The current implementation contains significant flaws that disrupt the system and cause substantial losses for users. These issues arise because the calculations for buy and sell amounts are truncated to full millions of tokens.

In the **`buy()`** function, when a user transfers ETH to buy tokens, if the resulting amount of tokens for a given trade is, for example, 1.9M tokens, the user will still pay as if he is receiving 1.9M tokens but will only receive 1M. This results in a loss of approximately 47%.

```
uint256 actualTokens = (tokensToTransfer / 1e18) * 1_000_000 * 1e18;
```

In the **`sell()`** function, if a user transfers 1.9M tokens to sell, he will only be refunded ETH equivalent to 1M tokens, but the entire 1.9M tokens will be transferred to the contract. This results in a similar 47% loss. Furthermore, incorrect deduction of the internal **`n`** virtual accounting value, used for bonding curve calculations, leads to inaccurate stored values, which inflates the token price.

```
uint256 newN = n - (tokenAmount / (1_000_000)); // <-- truncation
uint256 currentETH = ethForN(n);
uint256 newETH = ethForN(newN);
uint256 refundETH = currentETH - newETH;
```

```
        ...
        require(
            tokenContract.transferFrom(msg.sender, address(this),
tokenAmount), // <-- full amount transfer
            "Token transfer failed"
        );
```

```
    function ethForN(uint256 n_) public pure returns (uint256) {
        uint256 n_value = n_ / 1e18; // <-- truncation
```

This issue stems from the **ethForN()** function, where the bonding curve is implemented as a step function that calculates values per million tokens. The decision to use cubic calculations in the bonding curve introduced complications, and to avoid overflows, the calculations were restricted to per-million token steps, which impacts both the linear and cubic parts of the equation.

Recommendation

While there is no simple solution for cubic calculations, the bonding curve can be updated to better align with expected calculations and business requirements. The curve can be smoothed by implementing a linear function with a granularity of 1 wei and a cubic part with a granularity of 1e6, instead of the current 1e24 granularity for both parts.

Update the **ethForN()** function as follows:

```
    function ethForN(uint256 n_) public pure returns (uint256) {
        uint256 linear = (P0 * n_) / 1e18;
        uint256 cubic = (A * ((n_ / 1e6) ** 3)) / 1e54;
        return linear + cubic;
    }
```

Where **P0** is **5e9** and **A** is **60e9**.

In addition to updating **ethForN()**, make the following adjustments to ensure consistency with the updated bonding curve:

- In **solveForN()**, use **uint256 high = MAX_MILLION_TOKENS * 1e24**.

- Remove adjustments to millions in the **`buy()`** and **`sell()`** functions.

- Update the old require:

```
        require(newN <= MAX_MILLION_TOKENS * 1e18, "Max token supply
reached");
```

and adjust it to correctly check for 500M tokens.

3S-BuzzFun-H01

Last buyer in the **`BondingCurve`** contract can experience significant fund losses if he overpays for remaining tokens

Id	3S-BuzzFun-H01
Classification	High
Severity	High
Likelihood	Medium
Category	Bug
Status	Addressed in #e579e74.

Description

The **`buy()`** function does not refund overpayments to the last buyer in a situation where he transfers more ETH than needed to complete the sale.

This issue arises from the incorrect interaction between the **`solveForN()`** function and the following require check:

```
require(newN <= MAX_MILLION_TOKENS * 1e18, "Max token supply reached");
```

The value returned by **`solveForN()`** will always be less than or equal to **`MAX_MILLION_TOKENS * 1e18`** due to internal rounding down and the fact that the result is constrained to the maximum **`MAX_MILLION_TOKENS * 1e18`**. Therefore, the require statement using the **`<=`** comparison is always satisfied.

In cases where the last buyer transfers more ETH to the **`buy()`** function than calculated by the **`ethForN()`** function, the excess funds will be lost as there is no refund mechanism to handle overpayments.

Recommendation

Firstly, update **`solveForN()`** to return the exact **`MAX_MILLION_TOKENS * 1e18`** value for ETH values of 10 or more. This can be achieved by adjusting the rounding in the **`uint256 mid`** calculation within the **`solveForN()`** function as follows:

```
uint256 mid = (low + high + 1) / 2;
```

Secondly, implement a refund mechanism in the **buy()** function for cases where **solveForN()** returns exactly **MAX_MILLION_TOKENS * 1e18**. For example:

```
...
uint256 newN = solveForN(newETH);
uint256 refund;
if (newN == MAX_MILLION_TOKENS * 1e18) {
    refund = (netETH - (ethForN(newN) - ethForN(n)));
    uint256 feeAdjustment =  refund / (100 - fees_PERCENT);
    fees -=  feeAdjustment;
    refund += feeAdjustment;
}
...
if (refund > 0) {
    (bool refundSent, ) = address(msg.sender).call{value: refund}("");
    require(refundSent, "Refund transfer failed");
}
```

3S-BuzzFun-H02

Uninitialized **`maxTransaction`** and **`maxWallet`** variables prevent trading for non-excluded users

Id	3S-BuzzFun-H02
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #cec834c.

Description

In the **`RetardedTokenContract`**, the variables **`maxTransaction`** and **`maxWallet`** are intended to enforce trading limits during the initial 10 minutes after trading is activated. This is conditional upon the **`maxBuyStat1`** variable being set to **`true`** in the constructor. The purpose of these variables is to restrict the maximum transaction size and wallet balance for users who are not excluded from these limits, as defined by the **`_isExcludedFromMaxTransaction`** mapping.

However, these variables are never initialized within the contract, resulting in their default value being zero. Consequently, if **`maxBuyStat1`** is true, any transaction amount or wallet balance for non-excluded users will be forced to be zero, effectively preventing them from trading during this initial 10 minutes period.

Recommendation

To resolve this issue, initialize the **`maxTransaction`** and **`maxWallet`** variables with appropriate values in the constructor.

3S-BuzzFun-H03

Potential Denial of Service in liquidity migration due to unprotected pair creation

Id	3S-BuzzFun-H03
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #cec834c.

Description

The **`addLiquidity()`** function in both RetardedTokenContract and BeastTokenContract is designed to migrate accumulated funds from the bonding curve to a UniswapV2 pool once the bonding curve phase is complete. The function attempts to create a new trading pair between the **`token`** and **`WETH`** using **`createPair()`**.

However, since the pair creation is performed within the migration function itself, a malicious actor could preemptively create the pair, causing the **`createPair()`** call inside **`addLiquidity()`** to revert. This would effectively block the liquidity migration process, trapping funds in the contract until administrative intervention.

Recommendation

Move the pair creation logic to the token contract constructor. This ensures the pair is created at deployment time when no external actors can interfere.

Since the **`addLiquidity()`** function can only be called before **`tradingActive`** is set to **`true`**, there is no risk of unauthorized liquidity addition before the official migration, a situation that would allow users to steal the liquidity being added by the token contract.

3S-BuzzFun-M01

Incorrect key usage in transfer delay feature causes global sell swap restriction

Id	3S-BuzzFun-M01
Classification	Medium
Severity	Medium
Likelihood	High
Category	Bug
Status	Addressed in #cec834c.

Description

The **`transferDelayEnabled`** function is designed to enforce a delay between consecutive token transfers to prevent rapid trading and potential market manipulation. This feature is particularly relevant during the initial trading period, where it enforces a 10-block delay between sell transactions to automated market maker (AMM) pairs.

However, there is an issue in the implementation of this feature within the **`_transfer`** function. The key used to track the last transfer timestamp in the **`_holderLastTransferTimestamp`** mapping is **`msg.sender`**, which is typically the address of the Uniswap router during swaps. This results in the transfer delay being applied globally to all users, rather than individually. Consequently, during the first 20 minutes of trading, only one sell swap can be performed every ten blocks across the entire user base, severely restricting trading activity.

Recommendation

To resolve this issue, the key used in the **`_holderLastTransferTimestamp`** mapping should be changed from **`msg.sender`** to **`from`**. This adjustment ensures that the transfer delay is applied on a per-user basis, rather than globally. By using the **`from`** address as the key, the contract will correctly track the last transfer timestamp for each individual user, allowing the intended 10-block delay to be enforced for each user's sell transactions to AMM pairs.

If the 10-blocks delay is intended to be enforced only between sell swaps, the update of **`_holderLastTransferTimestamp`** should be moved inside the **`if (_automatedMarketMakerPairs[to])`** branch.

3S-BuzzFun-M02

Automate liquidity migration actions to mitigate human error

Id	3S-BuzzFun-M02
Classification	Medium
Severity	High
Likelihood	Low
Category	Bug
Status	Addressed in #0f228c4.

Description

The BondingCurve contract's **`buy`** function is designed to facilitate the purchase of tokens along a bonding curve, adjusting the token supply and price dynamically based on demand. Currently, when the bonding curve is completed, the function emits an event and relies on manual intervention by an administrator to proceed with subsequent actions.

This manual process introduces potential for human error and delays. For example, if **`enableTrading`** is triggered before the bonding curve is fully funded and its ethers sent to the token contract, it will prevent any subsequent attempts to call **`addLiquidity`** once the funds from the bonding curve are received. To avoid locking funds, an admin would need to intervene by using the **`withdrawStuckTokens`** function.

To automate the proecss, the **`transferAll`** function, which is responsible for transferring all Ether and burning residual tokens, should be automatically invoked upon the completion of the bonding curve. Furthermore, the TokenContract's **`enableTrading`** function, which activates trading, should be called automatically at the end of **`transferAll`** 's execution. This function should be restricted to be callable only by the BondingCurve contract, rather than the owner, to ensure a seamless transition and reduce the risk of administrative errors.

Recommendation

To enhance the reliability and security of the protocol, it is recommended to automate the liquidity migration actions currently requiring manual intervention. Modify the BondingCurve contract to automatically call the **`transferAll`** function internally when the bonding curve is completed, thereby eliminating the need for an event-based manual

trigger. Additionally, adjust the **`transferAll`** function to automatically invoke the **`TokenContract.enableTrading`** function upon completion. Ensure that **`BondingCurve.transferAll`** is set as internal, and that **`TokenContract.enableTrading`** is callable only by the BondingCurve contract to prevent unauthorized access and maintain the integrity of the protocol's operational flow.

This automation will streamline the process, reduce the potential for human error, and ensure a more secure and efficient transition from bonding curve completion to active trading.

3S-BuzzFun-M03

Lack of slippage protection in **`buy`** and **`sell`** functions

Id	3S-BuzzFun-M03
Classification	Medium
Severity	Medium
Likelihood	High
Category	Bug
Status	Addressed in #cec834c.

Description

The **`buy`** and **`sell`** functions in the BondingCurve contract currently lack slippage protection mechanisms. These functions are integral to the contract's operation as they facilitate the purchase and sale of tokens based on a bonding curve model, which acts similarly to an Automated Market Maker (AMM).

The absence of slippage protection can lead to users receiving fewer tokens or less Ether than expected due to price fluctuations between the time a transaction is initiated and when it is executed. This can result in a poor user experience and potential financial loss.

Recommendation

To mitigate the risk of slippage, it is recommended to introduce parameters for minimum acceptable amounts in both the **`buy`** and **`sell`** functions. Specifically, the **`buy`** function should include a **`minTokens`** parameter, which specifies the minimum number of tokens the user expects to receive. Similarly, the **`sell`** function should incorporate a **`minEthers`** parameter, indicating the minimum amount of Ether the user expects to receive. Additionally, view functions should be implemented to allow users and frontends to compute these minimum amounts before executing a transaction. This will ensure that transactions are only executed if the expected outcomes are met, thereby protecting users from adverse price movements.

3S-BuzzFun-L01

Non-compliance with Checks-Effects-Interactions pattern in **buy** and **sell** functions

Id	3S-BuzzFun-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Addressed in #cec834c.

Description

The **buy** and **sell** functions in the **BondingCurve** contract do not fully adhere to the Checks-Effects-Interactions (CEI) pattern. The **buy** function allows users to purchase tokens by sending Ether, while the **sell** function enables users to sell tokens back to the contract in exchange for Ether.

Although both functions are protected by the **nonReentrant** modifier, which mitigates reentrancy attacks, they still perform external calls before updating the contract's state. This is particularly concerning in the **sell** function, where Ether is sent to the **msg.sender** before the state variable **n** is updated. This deviation from the CEI pattern could expose the contract to edge-case scenarios, such as read-only reentrancy attacks, where an attacker could potentially exploit an external protocol relying on the state of the **BondingCurve** contract.

Recommendation

It is recommended to strictly follow the Checks-Effects-Interactions pattern. This involves first performing all necessary checks, then updating the contract's state, and finally interacting with external contracts or sending Ether.

3S-BuzzFun-L02

Potential vulnerability to sandwich attacks in **`_swapBack`** function

Id	3S-BuzzFun-L02
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Acknowledged

Description

The **`_swapBack`** function in the **`TokenContract`** is designed to convert accrued token fees into Ether (ETH) and transfer the ETH to the treasury address. This function is automatically triggered during sell transactions when the accrued token fees reach or exceed the **`swapTokensAtAmount`** threshold. The function relies on **`_swapTokensForETH`** to perform the token-to-ETH conversion.

However, **`_swapTokensForETH`** does not enforce a minimum amount of ETH to be received (**`minAmountOut`**), which exposes the function to potential sandwich attacks. In such attacks, an adversary could manipulate the price of the token before and after the swap, effectively stealing the fees intended for the treasury.

Recommendation

To mitigate the risk of sandwich attacks, it is recommended to modify the **`_swapBack`** function to be callable manually by the contract owner or a trusted bot service, rather than being automatically triggered during swaps. Additionally, introduce a **`minAmountOut`** parameter in the **`_swapTokensForETH`** function to ensure that a minimum amount of ETH is received from the swap, thereby protecting against price manipulation and ensuring the integrity of the fee conversion process.

3S-BuzzFun-L03

Unvalidated token swap amounts may lead to Denial of Service in **_swapBack** function

Id	3S-BuzzFun-L03
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Addressed in #cec834c.

Description

The **setTokenLiqSwapAmount** and **setSwapTokensAtAmount** functions in the BeastTokenContract are designed to allow the contract owner to set thresholds for token swaps. Specifically, **setTokenLiqSwapAmount** sets the **_tokenSwapAmount**, which determines the amount of tokens to be swapped for ETH during liquidity operations. Meanwhile, **setSwapTokensAtAmount** sets the **swapTokensAtAmount**, which is the minimum token balance required in the contract before a swap can be triggered.

However, these functions lack validation checks to ensure that **_tokenSwapAmount** is always less than or equal to **swapTokensAtAmount**. If these conditions are not met, the **_swapBack** function could attempt to swap more tokens than are available, leading to a denial of service. This would prevent sell operations during the initial 20 minutes of trading or until sufficient fees are accrued, as the **_swapTokensForETH** function would be unable to execute due to insufficient token balance.

Recommendation

Implement validation checks within the **setTokenLiqSwapAmount** and **setSwapTokensAtAmount** functions to ensure that **_tokenSwapAmount** is always less than or equal to **swapTokensAtAmount**. By doing so, you can prevent the **_swapBack** function from attempting to swap more tokens than are available, thereby avoiding potential denial of service scenarios.

3S-BuzzFun-L04

Inconsistent hardcoded UniswapV2Router address in token contracts

Id	3S-BuzzFun-L04
Classification	Low
Severity	Low
Likelihood	Medium
Category	Bug
Status	Addressed in #cec834c.

Description

The **`uniswapV2Router`** storage variable is hardcoded to different addresses within the constructors of the RetardedTokenContract and BeastTokenContract. This inconsistency can lead to operational issues and potential redeployment costs if the incorrect address is used. The uniswapV2Router is a critical component in the token contracts, facilitating interactions with the Uniswap protocol for liquidity provision and token swaps.

Recommendation

To ensure consistency and reduce the risk of errors, the **`uniswapV2Router`** address should be stored in a single, centralized location and passed as a constructor parameter to both token contracts by either a deploy script or a factory contract. This approach allows for easy updates and ensures that both contracts use the same router address, minimizing the risk of discrepancies and the need for redeployments.

3S-BuzzFun-N01

Redundant bonding curve fill check in sell function

Id	3S-BuzzFun-N01
Classification	None
Category	Suggestion
Status	Addressed in #cec834c.

Description

In the **`sell`** function of the **`BondingCurve`** contract, there is a check on the contract's Ether balance to determine if the bonding curve has been filled ans should thus be deactivated. The **`sell`** function's primary purpose is to allow users to sell their tokens back to the contract in exchange for Ether, applying a spread and fees to the transaction.

The check on the contract's Ether balance is redundant because the balance is expected to decrease with each sell transaction. This makes the condition **`if (contractTokenBalance >= 9.95 ether)`** ineffective for determining the end of the bonding curve, as the balance will not naturally increase to meet this threshold through the **`sell`** function.

Recommendation

Remove the redundant check on the contract's Ether balance within the **`sell`** function. The mechanism for deactivating the bonding curve is already implemented in the **`buy`** function, where the contract's Ether balance is checked to determine if it has reached a sufficient level to end the bonding curve.

3S-BuzzFun-N02

Division before multiplication in fee calculation

Id	3S-BuzzFun-N02
Classification	None
Category	Good Code Practices
Status	Addressed in #cec834c.

Description

In the `TokenContract`'s `_transfer` function, the calculation of transaction fees involves dividing the `amount` by `10000` before multiplying by the `tax()` value. This approach can lead to a loss of precision due to the inherent nature of integer division in Solidity, where any fractional component is truncated. Although the impact is minimal given the 18 decimal places used by the token contract, this pattern is considered a bad practice.

Recommendation

The calculation should be adjusted to perform the multiplication before the division. This can be achieved by rearranging the operations to `fees = (amount * tax()) / 10000;`.

3S-BuzzFun-N03

Use **`safeTransfer`** for ERC20 Transfers

Id	3S-BuzzFun-N03
Classification	None
Category	Good Code Practices
Status	Acknowledged

Description

When interacting with certain ERC20 tokens, those tokens may not be compatible with the **`IERC20::transfer()`** or **`IERC20::transferFrom()`** interfaces, causing the transaction to revert.

The **`withdrawStuckTokens()`** function in the **`BeastTokenContract`** and **`RetardedTokenContract`** contracts currently uses **`transfer()`**:

```
IERC20(tkn).transfer(msg.sender, amount);
```

Recommendation

Consider using **`safeTransfer`** in this cases.

3S-BuzzFun-N04

Unvalidated tax parameters in constructor

Id	3S-BuzzFun-N04
Classification	None
Category	Suggestion
Status	Addressed in #cec834c.

Description

The constructor of the TokenContract accepts two parameters, **`_tax1`** and **`_tax2`**, which are intended to set the initial tax rates for swap transactions. These parameters are used by the contract's **`tax`** function, which is designed to apply decreasing fees over time: **`tax1`** in the first 5 minutes of trading, **`tax2`** in the next 15 minutes, 0 fees after that.

However, the constructor does not validate these parameters to ensure that they adhere to the intended logic of the tax function. Without proper validation, there is a risk that these parameters could be set inappropriately, leading to inconsistent behavior in the tax function, such as applying a higher tax rate after a lower one.

Recommendation

Implement validation logic within the constructor to ensure that the **`_tax1`** and **`_tax2`** parameters are set correctly. The validation should enforce that either both parameters are set to zero, indicating no tax, or that **`_tax1`** is >= than **`_tax2`**, ensuring that fee rates won't increase over time.

3S-BuzzFun-N05

Redundant **`withdrawStuckETH`** function in token contracts

Id	3S-BuzzFun-N05
Classification	None
Category	Good Code Practices
Status	Addressed in #cec834c.

Description

The **`withdrawStuckETH`** function in the BeastTokenContract and RetardedTokenContract is redundant as its functionality is already covered by the **`withdrawStuckTokens`** function. The **`withdrawStuckETH`** function is designed to allow the contract owner to withdraw any Ether that may be stuck in the contract. However, the **`withdrawStuckTokens`** function can achieve the same result by passing the zero address as the token address, which effectively withdraws Ether.

Recommendation

Remove the **`withdrawStuckETH`** function from both the BeastTokenContract and RetardedTokenContract to eliminate redundancy.

3S-BuzzFun-N06

Liquidity addition sends LP tokens to treasury instead of burning

Id	3S-BuzzFun-N06
Classification	None
Category	Suggestion
Status	Acknowledged with note: “This is temporary for testing, it will be updated at the final stage”

Description

The **`addLiquidity`** function in the token contracts is responsible for adding liquidity to the Uniswap pool by pairing the contract's tokens with ETH. This function is called when the bonding curve is filled, and the ETH gathered are sent to the token contract to be added as pool liquidity. However, the current implementation sends the resulting LP tokens to the treasury wallet instead of burning them. This design choice requires users to trust the protocol administrators not to misuse the LP tokens for personal gain, as they have the ability to withdraw liquidity and potentially profit from it.

Recommendation

To enhance trust and security within the protocol, it is recommended to burn the LP tokens instead of transferring them to the treasury wallet. This can be achieved by sending the LP tokens to the zero address, effectively removing them from circulation and ensuring that the liquidity remains locked in the pool. This approach eliminates the risk of administrators misusing the LP tokens.

3S-BuzzFun-N07

Not used internally **`public`** functions can be declared **`external`**

Id	3S-BuzzFun-N07
Classification	None
Category	Good Code Practices
Status	Addressed in #cec834c.

Description

Some functions within contracts are declared **`public`** but are never used within the code. These functions can be declared **`external`**.

RetardedTokenContract:

- **`burn()`**
- **`enableTrading()`**
- **`setTreasuryWallet()`**
- **`withdrawStuckTokens()`**
- **`withdrawStuckETH()`**
- **`isExcludedFromFees()`**
- **`isExcludedFromMaxTransaction()`**
- **`bondingCurveContract()`**

BeastTokenContract:

- **`burn()`**
- **`enableTrading()`**
- **`setTokenLiqSwapAmount()`**
- **`setSwapTokensAtAmount()`**
- **`setTreasuryWallet()`**
- **`withdrawStuckTokens()`**

- `withdrawStuckETH()`
- `isExcludedFromFees()`
- `isExcludedFromMaxTransaction()`

BondingCurve:

- `transferAll()`
- `buy()`
- `sell()`
- `withdrawStuckETH()`
- `transferOperatorAccess()`

Recommendation

Consider declaring these functions as `external`.

3S-BuzzFun-N08

Unreachable code in **`RetardedTokenContract`** and **`BeastTokenContract`** contracts

Id	3S-BuzzFun-N08
Classification	None
Category	Good Code Practices
Status	Addressed in #cec834c.

Description

In the **`RetardedTokenContract`** contract, the **`_swapTokensForETH()`** and **`_addLiquidity()`** internal functions are never used. Additionally, the **`_holderLastTransferTimestamp`** mapping is declared but never utilized.

In the **`BeastTokenContract`** contract, the **`_addLiquidity()`** internal function is never used.

Recommendation

Remove the unused code to improve readability and deployment costs.

3S-BuzzFun-N09

Redundant **`_swapping`** private storage variable in **`RetardedTokenContract`** contract

Id	3S-BuzzFun-N09
Classification	None
Category	Optimization
Status	Addressed in #cec834c.

Description

The swapping functionality is not utilized within the **`RetardedTokenContract`** contract. Consequently, the **`_swapping`** private storage declaration and its usage within **`_transfer()`** are redundant, causing unnecessary storage reads during transfers.

Recommendation

Remove the **`_swapping`** private storage variable and its usage from the **`RetardedTokenContract`** contract.

3S-BuzzFun-N10

Redundant **`_bcContract`** private storage variable

Id	3S-BuzzFun-N10
Classification	None
Category	Good Code Practices
Status	Addressed in #cec834c.

Description

In the **`BeastTokenContract`** contract, the **`_bcContract`** variable is assigned within the **`constructor()`** but is never used.

In the **`RetardedTokenContract`** contract, the **`_bcContract`** variable is assigned within the **`constructor()`** and exposed in the **`bondingCurveContract()`** view function. However, the same data is already stored in the **`TradingContract`** public storage variable.

Recommendation

Remove the **`_bcContract`** private storage variable from both contracts.

If needed, the **`bondingCurveContract()`** view function should return the value from the **`TradingContract`** public storage variable, or the function should be removed entirely.