coverage

bera.buzz

Smart Contract Security Review

Report prepared by

Coverage Labs

5 February, 2025

coveragelabs.io

Contents

1 Disclaimer **1**

2 About bera.buzz **2**

3 Methodology **3**
3.1 Context & Cleanup . . . 3
3.2 Manual Review . . . 3
3.3 Quality Assurance . . . 3
3.4 Fix Period . . . 3

4 Severity Classification **4**
4.1 Impact . . . 4
4.2 Likelihood . . . 4
4.3 Action Required . . . 4

5 Executive Summary **5**
5.1 Engineers Involved . . . 5
5.2 Project Summary . . . 5
5.3 Code Versioning Control . . . 5
5.4 Scope of Work . . . 5
5.5 Findings Summary . . . 6

6 Code Maturity **7**
6.1 Code Maturity Evaluation Guidelines . . . 7
6.2 Code Maturity Evaluation Results . . . 8

7 Access Control **9**

8 Findings **11**
8.1 Critical Findings . . . 11
8.1.1 Attacker can block the migration of liquidity in `BexLiquidityManager` by frontrunning the deployment of the `BEX` pool . . . 11
8.2 Medium Findings . . . 13
8.2.1 The owner can frontrun the liquidity migration mechanism to steal funds . . . 13
8.2.2 Flawed validation in `HighlightsManager` allows for foreign tokens to be highlighted 14
8.2.3 Wrong calculation of direct and indirect referral fees in `ReferralManager` will result in revenue loss for indirect referrers . . . 15
8.3 Low Findings . . . 17
8.3.1 Lack of deadline checks in `BuzzVault` can cause transactions to be executed under less favorable market conditions . . . 17
8.3.2 Rounding down allows users to avoid trading fees . . . 18
8.3.3 The `amountOut` calculation in `BuzzVaultExponential` favors users, leading to value extraction from the protocol . . . 19
8.3.4 Frontrunning attack on token creation in `BuzzTokenFactory` can cause users to buy the wrong token . . . 21
8.3.5 Unclaimed dust referral rewards in `ReferralProgram` become permanently locked after program expiration . . . 23
8.3.6 `BERA` sent in excess to the `BuzzTokenFactory` becomes permanently locked if there is no initial buy . . . 24
8.4 Informational Findings . . . 26
8.4.1 Useless `mint` function in `BuzzToken` . . . 26
8.4.2 Enforce every token deployed through the `BuzzTokenFactory` to have the `1bee` suffix 27
8.4.3 Protocol is incompatible with non-standard `ERC20` tokens like `USDT` . . . 29

1 Disclaimer

A security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities within the defined scope.

2 About bera.buzz

bera.buzz allows users to seamlessly launch and trade their own ERC20 tokens on the Berachain network. Tokens are initially created and traded directly on the platform until they accumulate enough liquidity to reach a predefined threshold. Once this threshold is reached liquidity is automatically migrated to a newly established pool on BEX.

3 Methodology

Our methodology is divided into four distinct phases, each designed to enhance the security and reliability of the contracts in scope.

3.1 Context & Cleanup

This phase focused on understanding the codebase's intricacies, establishing context, and removing known anti-patterns.

3.2 Manual Review

The codebase was thoroughly reviewed to uncover potential edge cases, design flaws, and attack vectors, ensuring alignment between business logic and technical specifications.

3.3 Quality Assurance

We documented the roles and capabilities of privileged actors within the protocol and assessed the codebase's maturity across various categories, identifying areas for improvement.

3.4 Fix Period

Mitigations for previously identified vulnerabilities were reviewed to verify their correctness and ensure no new issues were introduced.

4 Severity Classification

Impact / Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

4.1 Impact

- **High** - Leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - Leads to a moderate material loss of assets in the protocol or moderately harms a group of users.
- **Low** - Leads to a minor material loss of assets in the protocol or harms a small group of users.

4.2 Likelihood

- **High** - Almost certain to happen, easy to perform, or not easy but highly incentivized.
- **Medium** - Only conditionally possible or incentivized, but still relatively likely.
- **Low** - Requires a highly complex setup, or has little to no incentive in accomplishing the attack.

4.3 Action Required

- **Critical** - Must fix as soon as possible.
- **High** - Must fix.
- **Medium** - Should fix.
- **Low** - Could fix.
- **Informational** - Optional fix.

5 Executive Summary

Coverage engaged with bera.buzz over the course of 12 days to review bera.buzz-contracts.

5.1 Engineers Involved

Name	Role	Contact
José Garção	Lead Security Engineer	garcao.random@gmail.com
Rafael Nicolau	Lead Security Engineer	0xrafaelnicolau@gmail.com

5.2 Project Summary

Name	bera.buzz
Repository	bera.buzz-contracts
Language	Solidity
Platform	Berachain
Review Period	Jan 23rd - Feb 3rd
Fix Period	Feb 3rd - Feb 5th

5.3 Code Versioning Control

- Review period commit hash - `9dfdcda`.
- Fix period commit hash - `cbc59fa`.

5.4 Scope of Work

- `contracts/BexLiquidityManager.sol`
- `contracts/BuzzToken.sol`
- `contracts/BuzzTokenFactory.sol`
- `contracts/BuzzVault.sol`
- `contracts/BuzzVaultExponential.sol`
- `contracts/FeeManager.sol`
- `contracts/HighlightsManager.sol`
- `contracts/ReferralManager.sol`
- `contracts/TokenVesting.sol`

5.5 Findings Summary

Severity	Count	Fixed	Acknowledged
Critical	1	1	0
High	0	0	0
Medium	3	3	0
Low	6	5	1
Informational	3	3	0
TOTAL	**13**	**12**	**1**

6 Code Maturity

6.1 Code Maturity Evaluation Guidelines

Category	Description
Access Control	The use of robust access controls to handle identification and authorization, as well as ensuring safe interactions with the system.
Arithmetic	The proper use of mathematical operations, including addition, subtraction, multiplication, and division, as well as semantics.
Centralization	The proper use of permissionless principles for mitigating insider threats and managing risks posed by contract upgrades and permissioned functions
Code Stability	The extent to which the code was altered during the audit and the frequency of changes made over time.
Upgradability	The presence of parametrizations of the system that allow modifications after deployment, ensuring adaptability to future needs.
Front-Running	The system's resistance to front-running attacks, where transactions are manipulated to exploit market conditions.
Monitoring	The presence of events that are emitted whenever there are operations that change the state of the system.
Specification	The presence of comprehensive and readable codebase documentation outlining the purpose, functionality, and design choices of the system.
Testing and Verification	The presence of robust testing procedures, including unit tests, integration and end-to-end tests, ensuring the reliability and correctness of the system.

6.2 Code Maturity Evaluation Results

Category	Description
Access Control	Satisfactory. All access control functionalities are properly implemented, ensuring secure interactions with the system.
Arithmetic	Moderate. The codebase suffers from some rounding direction issues and a wrong mathematical formula regarding referral fees.
Centralization	Moderate. Some owner functions introduce the risk of changing protocol fees to 100% and adding and/or removing arbitrary addresses.
Code Stability	Satisfactory. No mitigation has required massive changes to the code during the review period.
Upgradability	Satisfactory. The protocol benefits from parameterization functions, allowing the system to adapt to future needs.
Front-Running	Weak. An attacker can front-run and block the migration of liquidity and front-run token creation, potentially causing users to buy the wrong token.
Monitoring	Satisfactory. All contracts in scope have comprehensive event emissions in every state changing operation.
Specification	Satisfactory. The NatSpec and external documentation are up-to-date and correctly describe the protocol's functionality.
Testing and Verification	Satisfactory. The test suite provided possesses strong test coverage with regard to statements, branches, functions and lines of code.

7 Access Control

The protocol implements access control through OpenZeppelin's `Ownable` pattern, granting administrative privileges to a single address. Below are all administrative capabilities available to the owner:

1. `BexTokenFactory`
    - The owner can whitelist different vaults with different swap curves through the `BexTokenFactory::setVault()` function.
    - The owner can pause and unpause the creation of tokens through the `BexTokenFactory::setAllowTokenCreation()` function.
    - The owner can change the address of the fee manager contract through the `BexTokenFactory::setFeeManager()` function.
    - The owner can whitelist different tokens to be used as counter part in the bonding curve through the `BexTokenFactory::setAllowedBaseToken()` function.
2. `BuzzVault`
    - The owner can pause trading through the `BuzzVault::pause()` function.
    - The owner can unpause trading through the `BuzzVault::unpause()` function.
3. `BexLiquidityFactory`
    - The owner can add vaults to the whitelist through the `BexLiquidityFactory::addVaults()` function.
    - The owner can remove vaults from the whitelist through the `BexLiquidityManager::removeVaults()` function.
    - The owner can change the berabator contract address through the `BexLiquidityFactory::setBerabatorAddress()` function.
    - The owner can whitelist tokens to be used in berabator through the `BexLiquidityFactory::addBerabatorWhitelist()` function.
4. `FeeManager`
    - The owner can change the trading fee percentage from 0% to 100% through the `FeeManager::setTradingFeeBps()` function.
    - The owner can change the listing fee cost in `BERA` through `FeeManager::setListingFee()` function.
    - The owner can change the migration fee percentage from 0% to 100% through the `FeeManager::setMigrationFeeBps()` function.
    - The owner can change the fee recipient address through the `FeeManager::setTreasury()` function.
5. `HighlightsManager`
    - The owner can change the fee recipient address through the `HighlightsManager::setTreasury()` function.
    - The owner can change the maximum allowed highlight duration through the `HighlightsManager::setHardCap()` function.
    - The owner can change the base fee per second highlighted in `BERA` through the `HighlightsManager::setBaseFee()` function.

- The owner can change time before a token can be highlighted again through the `HighlightsManager::setCoolDownPeriod()` function.
- The owner can pause token highlighting operations through the `HighlightsManager::pause()` function.
- The owner can unpause token highlighting operations through the `HighlightsManager::unpause()` function.

6. `ReferralManager`
    - The owner can change the direct referral fee percentage from 0% to 100% through the `ReferralManager::setDirectRefFeesBps()` function.
    - The owner can change the indirect referral fee percentage from 0% to 100% through the `ReferralManager::setIndirectRefFeesBps()` function.
    - The owner can change the deadline of the referral program through the `ReferralManager::setValidUntil()` function.
    - The owner can change the minimum reward threshold for users to withdraw their earned rewards through the `ReferralManager::setPayoutThreshold()` function.
    - The owner can whitelist different vaults to work with the referral program through the `ReferralManager::setWhitelistedVault()` function.

8 Findings

8.1 Critical Findings

8.1.1 Attacker can block the migration of liquidity in `BexLiquidityManager` by frontrunning the deployment of the `BEX` pool

Description

When merging liquidity from the bonding curve to `BEX`, the `BexLiquidityManager` is responsible for creating the pool where both tokens will be deposited. However, an attacker can deliberately create a liquidity pool in advance with the same parameters and `salt` as the one expected to be deployed by the `BexLiquidityManager` contract. This prevents the liquidity migration from proceeding, as the deployment of the new `BEX` pool fails due to an address collision revert, resulting in a permanent denial of service and loss of funds.

Context

- BexLiquidityManager.sol#L308-L331

Impact

High. The bonding curve will be impossible to merge, causing financial losses to users who bought the buzz token.

Likelihood

High. The address of the pool can be precomputed in advance, and there is a significant delay between token creation and liquidity merging, giving attackers time to create a conflicting pool and cause a denial of service.

Proof of Concept

```
function test_poc_liquidity_migration_dos() public {
    // Give BERA to both Alice and Bob to buy the buzz token with.
    deal(_ALICE, 50e18);
    deal(_BOB, 450e18);

    // Set the existing fees 0 to avoid any fees being charged.
    vm.startPrank(_OWNER);
    feeManager.setListingFee(0);
    feeManager.setTradingFeeBps(0);
    feeManager.setMigrationFeeBps(0);
    vm.stopPrank();

    // Alice creates a new buzz token via the buzz token factory with
    // an initial buy of 50 BERA.
    vm.prank(_ALICE);
    address token = buzzTokenFactory.createToken{value: 50e18}(
        ["AliceCoin", "AC"],
        [address(wBERA), address(buzzVaultExponential)],
        [uint256(1000e18), uint256(1500e18)],
        50e18,
        keccak256("0x1337")
    );

    // An attacker creates a 50/50 weighted pool with the same parameters as the
    // one that will be created once liquidity is migrated.
    IERC20[] memory tokens = new IERC20[](2);
    tokens[0] = IERC20(address(wBERA));
    tokens[1] = IERC20(token);
    uint256[] memory weights = new uint256[](2);
    weights[0] = 0.5e18;
    weights[1] = 0.5e18;
```

```
        vm.prank(_ATTACKER);
        IWeightedPoolFactory(_BEX_POOL_FACTORY).create(
            string(
                abi.encodePacked(
                    "BEX 50 ",
                    ERC20(address(tokens[0])).symbol(),
                    " 50 ",
                    ERC20(address(tokens[1])).symbol()
                )
            ),
            string(
                abi.encodePacked(
                    "BEX-50",
                    ERC20(address(tokens[0])).symbol(),
                    "-50",
                    ERC20(address(tokens[1])).symbol()
                )
            ),
            tokens,
            weights,
            new IRateProvider[](2),
            0.01e18,
            address(bexLiquidityManager),
            keccak256(abi.encodePacked(address(wBERA), token))
        );

        // Bob buys the remaining amount of Alice Coin with 450 BERA,
        // which triggers the liquidity migration.
        vm.prank(_BOB);
        buzzVaultExponential.buyNative{value: 450e18}(
            token,
            0,
            address(0),
            _BOB
        );
    }
```

Recommendation

To mitigate this issue, consider one of the following solutions:

1. Partial Mitigation: Include the address of the last buyer and `block.timestamp` in the `salt` computation during pool creation. This increases entropy, making it harder for attackers to pre-compute and deploy a conflicting pool. However, this does not fully mitigate the issue, as liquidity migration can still be griefed. The migration can always be retried by submitting the buy transaction again, but it remains susceptible to being frontrun until it eventually succeeds.
2. Full Mitigation: Implement a permissioned pool factory for deploying `WeightedPool` instances. Restrict pool creation to the `BexLiquidityManager`, contract to completely prevent attackers from deploying conflicting pools in advance.

bera.buzz: Fixed in PR 53.

Coverage: Confirmed. The client decided to implement the partial mitigation.

8.2 Medium Findings

8.2.1 The owner can frontrun the liquidity migration mechanism to steal funds

Description

When liquidity is migrated from the bonding curve to `BEX`, the migration fee is charged over the `baseToken` through the `BuzzVault::_lockCurveAndDeposit()` function. Since the owner can set the migration fee to 100%, he could frontrun a large migration by increasing the fee to 99.9%, effectively stealing nearly all of the `baseToken` amount being migrated. A similar issue exists with the trading fee, but it is harder to exploit since users can protect themselves using the `minAmountOut` parameter.

Context

- BuzzVault.sol#L392-L426
- FeeManager.sol#L163-L168
- FeeManager.sol#L141-L146

Impact

High. The owner can steal nearly all of the `baseToken` funds during liquidity migration.

Likelihood

Low. The attack requires owner privileges, limiting the risk to cases of malicious or compromised ownership.

Recommendation

Consider modifying the `FeeManager::setMigrationFeeBps()` and `FeeManager::setTradingFeeBps()` functions to cap fee percentages to reasonable values. These caps should also be enforced in the constructor.

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.2.2 Flawed validation in `HighlightsManager` allows for foreign tokens to be highlighted

Description

The protocol allows tokens to be highlighted on the platform under the condition that they have a certain suffix. However, this condition is not enough to guarantee that only tokens that have been deployed by the protocol are the only ones to be highlighted. If foreign addresses somehow have the same suffix, they can be highlighted on the platform, which is not the intended behavior.

Context

- HighlightsManager.sol#L109-L136

Impact

Medium. This issue breaks a key feature of the protocol, allowing foreign addresses to be highlighted on the platform, possibly including addresses that correspond to tokens that have been deployed by competitors. By taking the space of native users of the protocol for foreign users, it could damage the reputation and trust of the protocol.

Likelihood

Medium. This issue is somewhat likely to occur, especially if there is an incentive. If the protocol captures significant attention and volume, it will incentivize foreign users to precompute addresses with the same suffix to be highlighted on the platform.

Recommendation

Store the `BuzzTokenFactory` contract's instance in the `HighlightsManager` contract and call the `BuzzTokenFactory::isDeployed()` to ensure that the token being highlighted was deployed by the protocol's token factory.

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.2.3 Wrong calculation of direct and indirect referral fees in `ReferralManager` will result in revenue loss for indirect referrers

Description

Whenever a user performs a trade, their direct and indirect referrers receive a referral fee, which is charged on top of the trading fee. The direct and indirect referral commissions are supposed to be 15% and 1% of the trading fee, respectively. However, in the current implementation, the direct referral fee is 15,84% and the indirect referral fee is 0,16% of the trading fee.

Context

- ReferralManager.sol#L135-L136

Impact

Low. Indirect referrers will see their referral fee cut by 6.25x, leading to a loss of rewards. Consequently, it would make it harder for them to reach the payout threshold to claim their rewards.

Likelihood

High. This issue will occur every time users perform a trade and have both a direct and indirect referrer.

Proof of Concept

```
function test_wrong_referral_fees_calculation() public {
    // Get the cost of the listing fee in the native token.
    uint256 listingFee = feeManager.listingFee();

    // Give the amount of the listing fee to the user.
    deal(_USER, listingFee);

    // Create a new token via the factory.
    string[2] memory metadata;
    metadata[0] = "Memecoin";
    metadata[1] = "MEME";

    address[2] memory addr;
    addr[0] = address(NECT);
    addr[1] = address(buzzVaultExponential);

    uint256[2] memory raiseData;
    raiseData[0] = 1e18;
    raiseData[1] = 1000e18;

    vm.prank(_USER);
    address token = buzzTokenFactory.createToken{value: listingFee}(metadata, addr,
        raiseData, 0, keccak256("MEME"));

    // Set direct and indirect referrers.
    vm.startPrank(address(buzzVaultExponential));
    referralManager.setReferral(_BOB, _ALICE);
    referralManager.setReferral(_ALICE, _USER);
    vm.stopPrank();

    // Buy tokens.
    uint256 baseAmount = 10e18;
    deal(address(NECT), _USER, baseAmount);

    vm.startPrank(_USER);
    NECT.approve(address(buzzVaultExponential), baseAmount);
    buzzVaultExponential.buy(token, baseAmount, 0, address(0), _USER);
    vm.stopPrank();

    // Compute fees.
    uint256 tradingFee = baseAmount * feeManager.tradingFeeBps() / 1e4;
```

```
    uint256 directFee = referralManager.getReferralRewardFor(_ALICE, address(NECT))
        ;
    uint256 directFeeBps = directFee * 1e4 / tradingFee;
    uint256 indirectFee = referralManager.getReferralRewardFor(_BOB, address(NECT))
        ;
    uint256 indirectFeeBps = indirectFee * 1e4 / tradingFee;

    // Assert that direct and indirect referral fees do not match.
    assertNotEq(directFeeBps, referralManager.directRefFeeBps());
    assertNotEq(indirectFeeBps, referralManager.indirectRefFeeBps());
}
```

Recommendation

Whenever there is an indirect referrer, compute their share of the total amount instead of applying the `indirectRefFeeBps` directly.

```
function receiveReferral(
    address user,
    address token,
    uint256 amount
) external {
    ...
    if (indirectReferral[user] != address(0)) {
        // If there is an indirect referral
+       uint256 indirectRefFeeShareBps = (indirectRefFeeBps * MAX_FEE_BPS) /
+           (directRefFeeBps + indirectRefFeeBps);
+       uint256 indirectReferralAmount = (amount * indirectRefFeeShareBps) /
+           MAX_FEE_BPS;
-       uint256 indirectReferralAmount = (amount * indirectRefFeeBps) /
-            MAX_FEE_BPS;
        ...
    }
    ...
}
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.3 Low Findings

8.3.1 Lack of deadline checks in `BuzzVault` can cause transactions to be executed under less favorable market conditions

Description

The `BuzzVault::buyNative()`, `BuzzVault::buy()` and `BuzzVault::sell()` functions lack deadline parameters and checks. Without these checks, transactions could be pending in the mempool for extended periods and executed at a latter time, potentially under less favorable market conditions than when the user initiated the transaction. While the contract implements slippage protection via `minTokensOut` and `minAmountOut` parameters, the lack of `deadline` checks still leaves users vulnerable to value extraction.

Context

- BuzzVault.sol#L138-L169
- BuzzVault.sol#L179-L205
- BuzzVault.sol#L216-L257

Impact

Low. Users may have their transactions executed during less favorable market conditions, however this issue is minimized due to the slippage protection already present in the implementation.

Likelihood

Medium. While the presence of slippage protection mitigates most risks, the lack of deadline checks could still cause users to get a worse quote than what they otherwise could've got if their transaction was not held.

Recommendation

Modify the `BuzzVault::buyNative()`, `BuzzVault::buy()` and `BuzzVault::sell()` functions to include a `deadline` parameter, and check its expiration to allow users to specify a maximum time their transaction should be valid for.

bera.buzz: Acknowledged. We did not want to introduce ABI breaking changes.

Coverage: Acknowledged.

8.3.2 Rounding down allows users to avoid trading fees

Description

Whenever users buy or sell tokens, they are charged a 1% trading fee, which is computed in the `FeeManager::quoteTradingFee()` function of the `FeeManager` contract using the following expression:

```
function quoteTradingFee(
    uint256 amount
) external view returns (uint256 fee) {
    fee = (amount * tradingFeeBps) / FEE_DIVISOR;
}
```

The division above is rounded down, allowing users to bypass the fee by trading small amounts.

Context

- FeeManager.sol#L116-L120

Impact

Low. Even when the trading fee rounds down to zero, the precision loss remains minimal.

Likelihood

Medium. Any trading fee computation that involves division precision loss will result in the user paying less than the intended 1%. Consequently, users trading with very small amounts can bypass the fee entirely.

Recommendation

Consider using OpenZeppelin's `Math.sol` library to perform the aforementioned division.

```
+ import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

  contract FeeManager is Ownable, IFeeManager {
+     using Math for uint256;

      function quoteTradingFee(
          uint256 amount
      ) external view returns (uint256 fee) {
-         fee = (amount * tradingFeeBps) / FEE_DIVISOR;
+         fee = amount.mulDiv(tradingFeeBps, FEE_DIVISOR, Math.Rounding.Up);
      }
  }
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.3.3 The `amountOut` calculation in `BuzzVaultExponential` favors users, leading to value extraction from the protocol

Description

In the `BuzzVaultExponential` contract, the internal functions `_calculateBuyPrice` and `_calculateSellPrice` compute the corresponding `amountOut` when buying and selling tokens, respectively. The expression used in both functions to compute the expected `amountOut` favors users, leading to a decrease in the value of the constant `k`. This means that users are receiving more tokens than they should and consequently extracting value from the protocol.

Context

- BuzzVaultExponential.sol#L210
- BuzzVaultExponential.sol#L229

Impact

Low. The value extracted from the protocol in each buy or sell transaction is minimal.

Likelihood

Medium. Any buy or sell transaction that involves a precision loss in the `amountOut` calculation will result in value extraction from the protocol.

Proof of Concept

```
function test_amount_out_favors_users() public {
    // Create token
    vm.startPrank(_USER);

    string[2] memory metadata;
    metadata[0] = "Meme Coin";
    metadata[1] = "MEME";

    address[2] memory addr;
    addr[0] = address(NECT);
    addr[1] = address(buzzVaultExponential);

    uint256[2] memory raiseData;
    raiseData[0] = 1e27;
    raiseData[1] = 100_001e22;

    deal(_USER, feeManager.listingFee());
    address token = buzzTokenFactory.createToken{value: feeManager.listingFee()}(
        metadata, addr, raiseData, 0, keccak256("MEME")
    );

    // Buy token
    deal(address(NECT), _USER, 1e18);
    NECT.approve(address(buzzVaultExponential), 1e18);
    buzzVaultExponential.buy(token, 1e18, 0, address(0), _USER);
    vm.stopPrank();

    // Assert that the new K has become smaller
    (,, uint256 tokenBalance, uint256 baseBalance,,,, uint256 k) =
        buzzVaultExponential.tokenInfo(token);
    uint256 newK = tokenBalance * baseBalance;

    console.log("INITIAL K  :", k);
    console.log("FINAL K    :", newK);

    assertLt(newK, k);
}
```

Recommendation

Consider using the expression used by Uniswap V2 to compute the corresponding `amountOut`, favoring the protocol:

```
function _calculateBuyPrice(
    uint256 baseAmountIn,
    uint256 baseBalance,
    uint256 quoteBalance,
    uint256 quoteThreshold,
-   uint256 k
) internal pure returns (uint256 amountOut, bool exceeded) {
-   uint256 amountAux = quoteBalance - k / (baseBalance + baseAmountIn);
+   uint256 amountAux = (quoteBalance * baseAmountIn) /
+      (baseBalance + baseAmountIn);
    exceeded = amountAux >= quoteBalance - quoteThreshold;
    amountOut = exceeded ? quoteBalance - quoteThreshold : amountAux;
}

function _calculateSellPrice(
    uint256 quoteAmountIn,
    uint256 quoteBalance,
    uint256 baseBalance,
-   uint256 k
) internal pure returns (uint256 amountOut) {
-   amountOut = baseBalance - k / (quoteBalance + quoteAmountIn);
+   amountOut = (baseBalance * quoteAmountIn) / (quoteBalance + quoteAmountIn);
}
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.3.4 Frontrunning attack on token creation in `BuzzTokenFactory` can cause users to buy the wrong token

Description

The `BuzzTokenFactory` uses `CREATE3` to generate token addresses based on a given `address` and `salt`. The `address` is always set to `address(this)` and the `salt` is provided as input to the `BuzzTokenFactory::createToken()`. This setup makes it possible for an attacker to monitor the mempool, observe the `salt` used in an upcoming token creation transaction, and frontrun it, causing the transaction to fail. While this is primarily a griefing attack that would incur the cost of the `listingFee`, it becomes more problematic if the user does not make an initial purchase (`baseAmount == 0`) during token creation and submits a separate transaction to buy the token right after. In this case, an attacker can frontrun the token creation transaction, resulting in the user accidentally buying the attacker's token instead. Here's step by step overview of how the attack could unfold:

1. Alice submits a transaction to create a new token, specifying a `salt`, name `Alice Coin`, and symbol `AC`.
2. Alice then submits a transaction to buy 100,000 `NECT` worth of her newly created token.
3. Bob observes Alice's transaction in the mempool and frontruns her by creating a token with the same `salt`, but with the name `Bob Coin` and symbol `BC`.
4. As a result, once the second transaction submitted by Alice goes through she ends up buying Bob's token instead of her own.

Context

- BuzzTokenFactory.sol#L127-L202

Impact

Medium. The attack can result in users inadvertently purchasing the wrong token, potentially with a significant amount of capital at stake.

Likelihood

Low. For this attack to succeed, the user would need to submit the token creation and purchase transactions separately, which is not the intended behavior when creating a token with the expectation of an initial buy.

Proof of Concept

```
function test_poc_frontrunning_token_creation() public {
    // Give BERA to pay for the listing fee to both Alice and Bob.
    uint256 listingFee = feeManager.listingFee();
    deal(_ALICE, listingFee);
    deal(_BOB, listingFee);

    // Give NECT to Alice for her to perform the first buy of the token she
    // wants to create.
    uint256 buyAmount = 100_000e18;
    deal(address(NECT), _ALICE, buyAmount);

    // Alice approves the buzz vault exponential to spend NECT.
    vm.prank(_ALICE);
    NECT.approve(address(buzzVaultExponential), buyAmount);

    // Alice precomputes the address of the token that she will create.
    address aliceToken =
        ICREATE3Factory(_CREATE3_DEPLOYER).getDeployed(address(buzzTokenFactory),
            keccak256("0x1337"));

    // Bob frontruns Alice's transaction and creates a new buzz token with the
    // same base token and salt as Alice.
```

```
    // Bob also uses the same initial and final reserve amounts as Alice to make
    // sure that her transaction does not revert due to slippage protection.
    string[2] memory metadata;
    metadata[0] = "Bob Memecoin";
    metadata[1] = "BOBMEME";
    address[2] memory addr;
    addr[0] = address(NECT);
    addr[1] = address(buzzVaultExponential);
    uint256[2] memory raise;
    raise[0] = 1e18;
    raise[1] = 1000e18;
    vm.prank(_BOB);
    address bobToken =
        buzzTokenFactory.createToken{value: listingFee}(metadata, addr, raise, 0,
            keccak256("0x1337"));

    // Alice's transaction to create a new token will revert because Bob's token
    // was already deployed at the expected address.
    metadata[0] = "Alice Coin";
    metadata[1] = "AC";
    addr[0] = address(NECT);
    addr[1] = address(buzzVaultExponential);
    raise[0] = 1e18;
    raise[1] = 1000e18;
    vm.prank(_ALICE);
    vm.expectRevert();
    buzzTokenFactory.createToken{value: listingFee}(metadata, addr, raise,
        buyAmount, keccak256("0x1337"));

    // Alice's initial buy will go through.
    uint256 quote = buzzVaultExponential.quote(aliceToken, buyAmount, true);
    vm.prank(_ALICE);
    buzzVaultExponential.buy(aliceToken, buyAmount, quote, address(0), _ALICE);

    // Assert that by frontrunning Alice's transaction, Bob's newly created token
    // has the same address as the one Alice was expecting to create.
    assertEq(bobToken, aliceToken);
    // Assert that the token Alice bought was actually Bob's token.
    assertEq(BuzzToken(aliceToken).name(), "Bob Coin");
    assertEq(BuzzToken(aliceToken).symbol(), "BC");
    // Assert that Alice's buy went through and she bought Bob's token.
    assertEq(BuzzToken(bobToken).balanceOf(_ALICE), quote);
}
```

Recommendation

Consider preventing users from buying or selling buzz tokens within the same block they were created. When registering a buzz token store the `block.number` when calling the `BuzzVault::registerToken()` and use it to enforce this restriction in the `BuzzVault::buyNative()`, `BuzzVault::buy()` and `BuzzVault::sell()` functions. Allow the token creator to perform buy and sell operations within the block it was created to ensure that the auto-buy mechanism works as intended.

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.3.5 Unclaimed dust referral rewards in `ReferralProgram` become permanently locked after program expiration

Description

Users are allowed to claim their referral rewards until the referral program has expired. Once it has expired, all the rewards that weren't claimed will be lost forever since there is no mechanism implemented to recover them.

Context

- ReferralManager.sol#L237-L249

Impact

Low. All the referral rewards that were not claimed, either because of forgetting or not having reached the payout threshold, will be lost forever.

Likelihood

Medium. There is a substantial probability of having pending rewards after the referral program expiration date.

Recommendation

Consider allowing the full reward amount to be claimed once the referral program ends.

```
function claimReferralReward(address token) external {
    if (token == address(0)) revert ReferralManager_AddressZero();
    uint256 reward = _referrerBalances[msg.sender][token];

-   if (reward < payoutThreshold[token])
+   if (reward < payoutThreshold[token] && validUntil >= block.timestamp)
        revert ReferralManager_PayoutBelowThreshold();
    if (reward == 0) revert ReferralManager_ZeroPayout();

    _referrerBalances[msg.sender][token] = 0;
    IERC20(token).safeTransfer(msg.sender, reward);

    emit ReferralPaidOut(msg.sender, token, reward);
}
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.3.6 BERA sent in excess to the BuzzTokenFactory becomes permanently locked if there is no initial buy

Description

If a user creates a token through the `BuzzTokenFactory::createToken()` function with a `msg.value` greater than the required `listingFee`, any amount of BERA sent in excess will be permanently locked in the `BuzzTokenFactory`. This occurs because, without an initial buy for the newly created contract, there is no mechanism in place to refund the overpaid tokens.

Context

- BuzzTokenFactory.sol#L149-L153
- BuzzTokenFactory.sol#L173-L202

Impact

Medium. While the locked amount does not affect the core functionality of the contract, it can lead to a financial loss for users, as any excess funds sent will be permanently locked and irretrievable.

Likelihood

Low. If users mistakenly send more than the required listing fee, the excess funds will remain permanently locked in the contract.

Proof of Concept

```
function test_poc_excess_native_token_not_reembursed() public {
    // Get the cost of the listing fee in the native token.
    uint256 listingFee = feeManager.listingFee();

    // Give twice the amount of the listing fee to the user.
    deal(_USER, listingFee * 2);

    // Create a new token via the factory.
    string[2] memory metadata;
    metadata[0] = "Memecoin";
    metadata[1] = "MEME";
    address[2] memory addr;
    addr[0] = address(wBERA);
    addr[1] = address(buzzVaultExponential);
    uint256[2] memory raiseData;
    raiseData[0] = 1e18;
    raiseData[1] = 1000e18;

    vm.startPrank(_USER);
    buzzTokenFactory.createToken{value: listingFee * 2}(
        metadata,
        addr,
        raiseData,
        0,
        keccak256("MEME")
    );
    vm.stopPrank();

    // Assert that the amount sent in excess is not reembursed.
    assertEq(_USER.balance, 0);

    // Assert that the treasury has only received the listing fee.
    assertEq(address(_TREASURY).balance, listingFee);

    // Assert that the remaning amount is locked in the factory forever.
    assertEq(address(buzzTokenFactory).balance, listingFee);
}
```

Recommendation

Modify the `BuzzTokenFactory::createToken()` function to reimburse the excess `BERA` when there is no initial buy.

```
function createToken(
    string[2] calldata metadata,
    address[2] calldata addr,
    uint256[2] calldata raiseData,
    uint256 baseAmount,
    bytes32 salt
) external payable returns (address token) {
    ...
+   uint256 remainingValue = msg.value - listingFee;
    if (baseAmount > 0) {
-       uint256 remainingValue = msg.value - listingFee;
        if (remainingValue > 0) {
            ...
        }
+   } else {
+       if (remainingValue > 0) {
+           (success,) = msg.sender.call{value: remainingValue}("");
+           if (!success) revert BuzzTokenFactory_EthTransferFailed();
+       }
+   }
}
```

And add the following custom error.

```
+ error BuzzTokenFactory_EthTransferFailed();
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.4 Informational Findings

8.4.1 Useless `mint` function in `BuzzToken`

Description

The `mint()` function in the `BuzzToken` contract that is not being used anywhere in the codebase.

Context

- BuzzToken.sol#L31-L36

Recommendation

Consider removing the `mint` function and the access control mechanism.

```
- import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";

- contract BuzzToken is ERC20, AccessControl, IBuzzToken {
+ contract BuzzToken is ERC20, IBuzzToken {
-     /// @dev access control minter role.
-     bytes32 public immutable MINTER_ROLE;

      constructor(
          string memory name,
          string memory symbol,
          uint256 _initialSupply,
          address mintTo,
-         address _owner
      ) ERC20(name, symbol) {
          _mint(mintTo, _initialSupply);
-         MINTER_ROLE = keccak256("MINTER_ROLE");
-         _grantRole(MINTER_ROLE, _owner);
      }

-     function mint(
-         address account,
-         uint256 amount
-     ) external onlyRole(MINTER_ROLE) {
-         _mint(account, amount);
-     }
  }
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.4.2 Enforce every token deployed through the `BuzzTokenFactory` to have the `1bee` suffix

Description

The `BuzzTokenFactory` contract is designed to ensure that all deployed tokens have the `1bee` suffix in their contract address. However, this constraint is not enforced, allowing tokens to be deployed without the expected suffix.

Context

- BuzzTokenFactory.sol

Impact

Low. Although tokens can be deployed through the `BuzzTokenFactory` without the `1bee` suffix preventing the token from possibly being highlighted.

Likelihood

Low. Most token deployments occur via the frontend, which enforces the use of a `salt` that will result in a token address with `1bee` as suffix.

Recommendation

```
contract BuzzTokenFactory is AccessControl, IBuzzTokenFactory {
+    error BuzzTokenFactory_InvalidSuffix();

+    bytes public suffix;

-    constructor(address _owner, address _createDeployer, address _feeManager) {
+    constructor(
+         address _owner,
+         address _createDeployer,
+         address _feeManager,
+         bytes memory _suffix)
+    {
         ...
+        suffix = _suffix;
         ...
         emit FeeManagerSet(_feeManager);
     }

     function _deployToken(
         string calldata name,
         string calldata symbol,
         address baseToken,
         address vault,
         bytes32 salt,
         uint256[2] calldata raiseData
     ) internal returns (address token) {
         ...
         isDeployed[token] = true;
+        _verifySuffix(token);
         ...
     }

+    function _verifySuffix(address token) internal view {
+       bytes memory tokenBytes = abi.encodePacked(token);
+        bytes memory cachedSuffix = suffix;
+
+        uint256 suffixLength = cachedSuffix.length;
+        uint256 tokenLength = tokenBytes.length;
+
+        for (uint256 i; i < suffixLength; ) {
+            if (cachedSuffix[i] != tokenBytes[tokenLength - suffixLength + i]) {
+                revert BuzzTokenFactory_InvalidSuffix();
+            }
```

```
+
+             unchecked {
+                 ++i;
+             }
+         }
+     }
}
```

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.

8.4.3 Protocol is incompatible with non-standard ERC20 tokens like USDT

Description

Some tokens, like `USDT`, implement an approval race protection mechanism requiring the pre-approval allowance to either be `0` or `type(uint256).max`. The protocol uses `safeApprove()` for `baseToken` approvals, however if the protocol ever decides to whitelist `USDT` as a `baseToken`, every `safeApprove()` call will revert if there's any unused allowance.

Context

- BexLiquidityManager.sol#L104
- BexLiquidityManager.sol#L105
- BuzzTokenFactory.sol#L192
- BuzzTokenFactory.sol#L315
- BuzzVault.sol#L410
- BuzzVault.sol#L414
- BuzzVault.sol#L415-L418
- BuzzVault.sol#L495
- BuzzVault.sol#L507
- BuzzVaultExponential.sol#L251

Impact

Low. Functions that use `safeApprove()` with non-standard `ERC20` tokens like `USDT` may revert, making them unusable.

Likelihood

Low. The issue only occurs if `USDT` is whitelisted and allowance amounts are not fully spent, but minor rounding errors could trigger it.

Recommendation

Use `forceApprove()` instead of `safeApprove()`.

bera.buzz: Fixed in PR 53.

Coverage: Confirmed.