# High Severity 1

## Treasury ETH Transfer Can Cause Complete Transaction Reversion

HIGH

The zapETH function uses a low-level call to send ETH to the treasury address without proper error handling. If the treasury is a contract that reverts on receive (e.g., due to gas limit, logic failure, or malicious behavior), the entire zap transaction will revert, causing users to lose gas and preventing them from providing liquidity. This creates a denial-of-service vector where a compromised or malfunctioning treasury can brick the entire protocol.

### Affected Function:

*// Send 10% + any leftover ETH (router refunds) to treasury*

uint256 leftover = self.balance;

uint256 payout = treasuryDue + leftover;

if (payout > 0) {

(bool ok, ) = treasury.call{value: payout}("");

if (!ok) revert BadRouterRefund();

}

#### Impact:

Complete denial of service for all users trying to zap ETH into liquidity. A malicious owner could set treasury to a contract that selectively reverts, griefing specific users or blocking the protocol entirely. Users would lose gas fees on failed transactions.

#### PoC:

contract MaliciousTreasury {

receive() external payable {

*// Selectively revert to grief users*

if (tx.origin == targetUser) {

revert("Blocked");

}

}

}

#### *// Attack:*

*// 1. Owner queues MaliciousTreasury as new treasury*

*// 2. After 12 hours, applies the malicious treasury*

*// 3. All zapETH calls now revert when trying to pay treasury*

### Fix:

*// Implement a pull pattern for treasury payments*

mapping(address => uint256) public treasuryBalance;

function zapETH(...) external payable nonReentrant returns (uint256 lpMinted) {

*// ... existing code ...*

*// Instead of direct transfer, accumulate treasury balance*

treasuryBalance[treasury] += payout;

emit Zapped(...);

return liquidity;

}

function withdrawTreasury() external {

uint256 amount = treasuryBalance[msg.sender];

treasuryBalance[msg.sender] = 0;

(bool ok, ) = msg.sender.call{value: amount}("");

require(ok, "Transfer failed");

}

# Medium Severity 3

## Optimal Swap Calculation Can Return Zero Leading to Inefficient Liquidity Addition

MEDIUM

The \_optimalSwapAmt function can return 0 when the mathematical formula produces a negative or zero result (when b <= c). In this case, the contract falls back to a naive 50/50 split which is suboptimal and results in worse price impact and less LP tokens minted. This particularly affects small deposits or edge cases in pool reserves.

View Details

### Affected Function:

function \_optimalSwapAmt(uint256 amountIn, uint256 rIn) internal pure returns (uint256) {

uint256 a = A\_NUM \* amountIn + B\_NUM \* rIn;

uint256 b = \_sqrt(rIn \* a);

uint256 c = C1 \* rIn;

if (b <= c) return 0; *// Falls back to 50/50 split*

return (b - c) / C2;

}

*// In zapETH:*

if (ethToSwap == 0) {

ethToSwap = ethForZap / 2; *// Suboptimal fallback*

}

#### Impact:

Users receive fewer LP tokens than optimal when the calculation fails, resulting in value loss. The 50/50 split can cause significant slippage in imbalanced pools, potentially failing slippage checks or giving users poor execution.

#### PoC:

*// Scenario: Very small deposit or specific reserve ratios*

*// If \_optimalSwapAmt returns 0, contract uses 50/50 split*

*// Example with imbalanced pool:*

*// ETH reserves: 100 ETH*

*// Token reserves: 10000 tokens*

*// User deposits: 0.001 ETH*

*// Optimal swap: ~0.0005 ETH*

*// Actual swap (50/50): 0.0005 ETH (different ratio, more slippage)*

### Fix:

function \_optimalSwapAmt(uint256 amountIn, uint256 rIn) internal pure returns (uint256) {

*// Add minimum bounds check*

if (amountIn < 1000) return amountIn \* 45 / 100; *// Use 45% for very small amounts*

uint256 a = A\_NUM \* amountIn + B\_NUM \* rIn;

uint256 b = \_sqrt(rIn \* a);

uint256 c = C1 \* rIn;

if (b <= c) {

*// Better fallback based on reserve ratio*

return amountIn \* 45 / 100; *// More conservative than 50%*

}

return (b - c) / C2;

}

## Timelock Can Be Bypassed Through Ownership Transfer

MEDIUM

While treasury updates have a 12-hour timelock, the owner can instantly transfer ownership to a new address that could then queue another treasury change. This allows circumventing the timelock's protective purpose through a shell game of ownership transfers.

View Details

### Affected Function:

function transferOwnership(address newOwner) external onlyOwner {

pendingOwner = newOwner; *// No timelock*

}

function queueTreasury(address payable newTreasury) external onlyOwner {

*// Has timelock, but new owner can queue immediately*

pendingTreasury = PendingAddress({ value: newTreasury, applyAfter: uint64(block.timestamp) + MIN\_DELAY });

}

#### Impact:

The timelock mechanism's security benefit is reduced as a malicious owner can rapidly change ownership to queue multiple treasury changes in parallel or reset the timelock by transferring to a fresh address.

#### PoC:

*// Day 1: Owner queues malicious treasury*

*// Day 1: Community notices and objects*

*// Day 1: Owner transfers ownership to newOwner*

*// Day 1: newOwner queues different treasury*

*// Now two treasury changes are pending*

*// Owner can keep doing this to confuse monitoring*

### Fix:

*// Add timelock to ownership transfers*

struct PendingOwner {

address value;

uint64 applyAfter;

}

PendingOwner public pendingOwnerTimelock;

function transferOwnership(address newOwner) external onlyOwner {

pendingOwnerTimelock = PendingOwner({

value: newOwner,

applyAfter: uint64(block.timestamp) + MIN\_DELAY

});

}

## Slippage Calculation Based on Stale Reserves

MEDIUM

The contract calculates expected output and slippage protection based on reserves queried at the beginning of the transaction. However, these reserves can change between the query and actual swap execution due to other transactions in the same block, leading to unexpected slippage or failed transactions.

View Details

### Affected Function:

*// Reserves queried here*

(uint112 r0, uint112 r1, ) = IUniswapV2Pair(pair).getReserves();

*// Used for calculation*

uint256 expectedOut = \_getAmountOut(ethToSwap, rETH, rTOKEN);

uint256 minOut = expectedOut \* (BPS - swapSlippageBps) / BPS;

*// But actual swap happens later with potentially different reserves*

router.swapExactETHForTokensSupportingFeeOnTransferTokens{value: ethToSwap}(

minOut,

path,

self,

deadline

);

#### Impact:

In volatile markets or during MEV attacks, the actual reserves during swap execution could be significantly different from the queried reserves, causing transactions to fail or execute with worse slippage than users expected.

#### PoC:

*// Block N:*

*// 1. User's transaction queries reserves: 100 ETH, 10000 tokens*

*// 2. MEV bot front-runs with large swap*

*// 3. Reserves now: 150 ETH, 8000 tokens*

*// 4. User's swap executes with much worse rate*

*// 5. Transaction may revert due to slippage protection*

### Fix:

*// Use tighter slippage bounds or implement a two-step process:*

*// 1. Query reserves in view function*

*// 2. Pass expected reserves as parameters*

*// 3. Validate reserves haven't changed significantly*

function zapETH(

uint256 swapSlippageBps,

uint256 addLpSlippageBps,

uint256 deadline,

uint256 minLPMinted,

uint256 maxReserveChange *// New parameter*

) external payable nonReentrant returns (uint256 lpMinted) {

*// Check reserves haven't changed more than maxReserveChange*

}

# Low Severity 5

## Incorrect Error Message on Treasury Transfer Failure

LOW

When the treasury ETH transfer fails, the contract reverts with 'BadRouterRefund()' error instead of a more appropriate error. This misleading error message could confuse users and developers debugging failed transactions, as the failure is related to treasury transfer, not router refunds.

View Details

### Affected Function:

if (payout > 0) {

(bool ok, ) = treasury.call{value: payout}("");

if (!ok) revert BadRouterRefund(); *// Wrong error for treasury failure*

}

#### Impact:

Confusion during debugging and monitoring. Users and developers may incorrectly diagnose transaction failures as router-related issues when they are actually treasury transfer failures. This could delay problem resolution and impact user experience.

#### PoC:

When treasury.call fails (e.g., treasury is a contract that reverts), users see 'BadRouterRefund' error which is semantically incorrect and misleading about the actual failure point.

### Fix:

error TreasuryTransferFailed();

if (payout > 0) {

(bool ok, ) = treasury.call{value: payout}("");

if (!ok) revert TreasuryTransferFailed();

}

## MIN\_ETH\_FOR\_ZAP Too Low Allows Dust Griefing

LOW

The MIN\_ETH\_FOR\_ZAP constant is set to only 2 wei, which is effectively no minimum. This allows griefing attacks where attackers can spam the contract with dust amounts that still emit events and consume gas but provide negligible liquidity. Additionally, such small amounts can cause precision loss in calculations and potentially revert due to Uniswap's minimum liquidity requirements.

View Details

### Affected Function:

uint256 public constant MIN\_ETH\_FOR\_ZAP = 2; *// wei; keep tiny to avoid surprising users*

*// In zapETH:*

if (ethForZap < MIN\_ETH\_FOR\_ZAP) revert InsufficientAmount();

#### Impact:

Event log spam making it difficult to monitor legitimate transactions. Potential for transactions to revert at the Uniswap level due to insufficient amounts. Gas griefing where attackers force the contract to perform expensive calculations for dust amounts.

#### PoC:

*// Attacker sends multiple transactions with tiny amounts*

for (uint i = 0; i < 100; i++) {

zapContract.zapETH{value: 100 wei}(100, 100, deadline, 0);

*// Each call:*

*// - Emits events polluting logs*

*// - Performs sqrt and optimal calculations*

*// - Likely reverts at Uniswap but after consuming gas*

}

### Fix:

*// Set a more reasonable minimum (e.g., 0.001 ETH)*

uint256 public constant MIN\_ETH\_FOR\_ZAP = 0.001 ether;

*// This prevents dust attacks while still allowing small legitimate deposits*

## Receive Function Allows Unrestricted ETH From WETH Contract

LOW

The receive function accepts ETH from both the router and WETH contracts. However, WETH can be called by anyone to unwrap ETH to any address. An attacker could unwrap WETH to this contract, potentially interfering with accounting or causing unexpected behavior when the contract calculates leftover balances.

View Details

### Affected Function:

receive() external payable {

if (msg.sender != address(router) && msg.sender != WETH) revert BadRouterRefund();

}

#### Impact:

Attackers can forcibly send ETH to the contract via WETH unwrapping, potentially interfering with the leftover calculation in zapETH. This could cause more ETH than intended to be sent to treasury, though the impact is limited as it would benefit the treasury rather than the attacker.

#### PoC:

*// Attacker can send ETH to contract outside of normal flow:*

WETH.deposit{value: 1 ether}();

WETH.transfer(zapContract, 1 ether);

*// Later someone could unwrap this WETH to the contract*

*// This ETH becomes part of 'leftover' in next zapETH call*

### Fix:

*// Track expected refunds more precisely*

uint256 private expectedRefund;

receive() external payable {

if (msg.sender != address(router)) revert BadRouterRefund();

*// Only accept from router during active zap*

}

*// Or implement pull pattern for any accumulated ETH*

## No Validation of Router Liquidity Return Values Beyond Minimums

LOW

The contract checks that returned amounts meet minimum thresholds but doesn't validate that the router actually used all provided tokens/ETH efficiently. If the router has a bug or is upgraded maliciously, it could return valid but suboptimal amounts while keeping the difference.

View Details

### Affected Function:

(uint256 amountToken, uint256 amountETH, uint256 liquidity) = router.addLiquidityETH{value: ethForLP}(

token,

tokensBought,

tokenMin,

ethMin,

msg.sender,

deadline

);

if (liquidity == 0) revert LPNotMinted();

if (amountToken < tokenMin || amountETH < ethMin) revert InsufficientOutput();

*// No check that amountToken <= tokensBought or amountETH <= ethForLP*

#### Impact:

If the router is compromised or contains bugs, it could use less than the provided amounts for liquidity while keeping the remainder, as long as it meets the minimum requirements. Users would receive valid but suboptimal LP positions.

#### PoC:

A malicious router could:

1. Receive 1 ETH and 100 tokens for liquidity

2. Only use 0.9 ETH and 90 tokens (meeting minimums)

3. Keep 0.1 ETH and 10 tokens

4. Return valid liquidity amount

### Fix:

*// Add upper bound validation*

if (amountToken > tokensBought) revert ExcessiveTokenUsage();

if (amountETH > ethForLP) revert ExcessiveETHUsage();

*// Or track exact refunds*

uint256 tokenRefund = tokensBought - amountToken;

uint256 ethRefund = address(this).balance - treasuryDue;

## No Mechanism to Recover Accidentally Sent Tokens

LOW

If users accidentally send ERC20 tokens directly to the contract (not through zapETH), there's no way to recover them. While the contract is designed to handle ETH and specific token interactions, it lacks a recovery mechanism for mistakenly sent tokens.

View Details

### Affected Function:

No recovery function exists. The contract can permanently trap tokens sent by mistake.

#### Impact:

Permanent loss of any ERC20 tokens accidentally sent to the contract address. This is particularly problematic if users confuse the contract address with another address or if there are UI bugs in integrating applications.

#### PoC:

*// User accidentally sends 1000 USDC to contract*

USDC.transfer(zapContract, 1000e6);

*// These tokens are now permanently locked*

### Fix:

function recoverToken(address token, uint256 amount) external onlyOwner {

require(token != address(this.token), "Cannot recover protocol token");

require(token != WETH, "Cannot recover WETH");

IERC20(token).transfer(treasury, amount);

emit TokenRecovered(token, amount);

}

# Gas Severity 2

## Gas Optimization: Redundant Balance Check for Treasury Payment

GAS

The contract reads address(this).balance to calculate leftover ETH after all operations, but this value is always known to be 0 or a refund amount since the contract doesn't hold ETH otherwise. The balance check adds unnecessary gas cost.

View Details

### Affected Function:

*// Send 10% + any leftover ETH (router refunds) to treasury*

uint256 leftover = self.balance; *// Unnecessary BALANCE opcode*

uint256 payout = treasuryDue + leftover;

#### Impact:

Unnecessary gas consumption for the BALANCE opcode (approximately 100 gas). While minor, this is called on every zap transaction.

#### PoC:

The contract only receives ETH from router refunds during the transaction, so leftover could be calculated differently.

### Fix:

*// Track expected values instead of reading balance*

uint256 routerRefund = ethForLP - amountETH; *// Calculate refund*

uint256 payout = treasuryDue + routerRefund;

*// Or if you must check balance, cache it once*

uint256 finalBalance = address(this).balance;

if (finalBalance > 0) {

*// Send finalBalance + treasuryDue*

}

## Sqrt Function Can Be Optimized Using Solidity 0.8 Features

GAS

The custom \_sqrt implementation uses a Newton-Raphson method that could be optimized using more modern Solidity patterns or replaced with a more gas-efficient implementation.

View Details

### Affected Function:

function \_sqrt(uint256 x) private pure returns (uint256 y) {

if (x == 0) return 0;

uint256 z = (x + 1) / 2;

y = x;

while (z < y) { y = z; z = (x / z + z) / 2; }

}

#### Impact:

Higher gas costs for every zap operation. The sqrt function is called for every transaction, so optimization would provide cumulative savings.

#### PoC:

Current implementation uses multiple divisions and iterations which consume significant gas.

### Fix:

function \_sqrt(uint256 x) private pure returns (uint256) {

if (x == 0) return 0;

*// Use bit manipulation for initial guess (more efficient)*

uint256 result = 1;

uint256 xAux = x;

if (xAux >= 0x100000000000000000000000000000000) { xAux >>= 128; result <<= 64; }

if (xAux >= 0x10000000000000000) { xAux >>= 64; result <<= 32; }

if (xAux >= 0x100000000) { xAux >>= 32; result <<= 16; }

if (xAux >= 0x10000) { xAux >>= 16; result <<= 8; }

if (xAux >= 0x100) { xAux >>= 8; result <<= 4; }

if (xAux >= 0x10) { xAux >>= 4; result <<= 2; }

if (xAux >= 0x4) { result <<= 1; }

*// Refine with Newton's method (fewer iterations needed)*

result = (result + x / result) >> 1;

result = (result + x / result) >> 1;

result = (result + x / result) >> 1;

return result < x / result ? result : x / result;

}

informational Severity

1

Missing Event for Pause State Changes

# Informational

The setPaused function emits a PausedSet event, but there's no indexed parameter to efficiently filter for pause/unpause events. This makes it harder for off-chain monitoring systems to track when the contract is paused or resumed.

View Details

### Affected Function:

event PausedSet(bool isPaused); *// Not indexed*

function setPaused(bool \_paused) external onlyOwner {

paused = \_paused;

emit PausedSet(\_paused);

}

#### Impact:

Monitoring and indexing services cannot efficiently query for pause state changes, potentially missing critical security events or failing to update UI appropriately.

#### PoC:

Off-chain services must scan all PausedSet events rather than being able to filter for `PausedSet(true)` or `PausedSet(false)` efficiently.

### Fix:

event PausedSet(bool indexed isPaused);

*// Or use separate events:*

event Paused();

event Unpaused();

function setPaused(bool \_paused) external onlyOwner {

paused = \_paused;

if (\_paused) emit Paused();

else emit Unpaused();

}