Addendum to Code4rena Audit Report

Subject: Resolution and Judge Validation of Issues S122 and S410
Project: BLACKHOLE DEX
Original Audit: https://code4rena.com/reports/2025-05-blackhole
Date: 07/03/2025

Overview:

This document serves as an addendum to the public Code4rena audit report <>. This is issued by BLACKHOLE DEX with approval from MrPotatoMagic who served as the panel of judges for the C4 competitive audit mitigation, clarifying the status of two previously listed issues(**M-05** and **M-10)** which were marked as **unmitigated** in the final C4 audit report. Based on a follow-up and additional courtesy review conducted by MrPotatoMagic we would like to confirm that **M-05** and **M-10** have been **mitigated**. Details furnished below.

[M-05] Griefing attack on `GenesisPoolManager.sol::depositNativeToken` leading to Denial of Service

Submitted by FavourOkerri, also found by AvantGard

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/factories/GenesisPoolFactory.sol#L56-L67

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/factories/PairFactory.sol#L139-L151

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GenesisPoolManager.sol#L100-L116

Finding description

The process of depositing native token to a Genesis Pool is vulnerable to a griefing attack, leading to a denial of service (DoS) for legitimate pool creations.

The vulnerability arises from the interaction of the following factors:

- Public information: The `nativeToken` and `fundingToken` addresses intended for a new Genesis Pool become public information when the `GenesisPoolFactory.createGenesisPool` transaction enters the mempool or through the `GenesisCreated` event it emits.
- Unrestricted pair creation: The `pairFactory.createPair` function allows any external actor to create a new liquidity pair for any given tokenA, tokenB, and stable combination, provided a pair for that combination doesn't already exist.

```
function createPair(address tokenA, address tokenB, bool stable) external
returns (address pair) {

        require(tokenA != tokenB, "IA"); // Pair: IDENTICAL_ADDRESSES

        (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB)
: (tokenB, tokenA);

        require(token0 != address(0), "ZA"); // Pair: ZERO_ADDRESS

        require(getPair[token0][token1][stable] == address(0), "!ZA");

        pair = IPairGenerator(pairGenerator).createPair(token0, token1,
stable);

        getPair[tokenA][tokenB][stable] = pair;

        getPair[tokenB][tokenA][stable] = pair; // Store in reverse direction

        allPairs.push(pair);

        isPair[pair] = true;



        emit PairCreated(token0, token1, stable, pair, allPairs.length);

    }
```

- Vulnerable validation: In `GenesisPoolManager.depositNativeToken()`, the protocol checks whether the pair already exists. If so, it requires that both token balances in the pair are zero — but does not verify whether the pair was created by the protocol itself.

```
function depositNativeToken(

        address nativeToken,

        uint256 auctionIndex,

        GenesisInfo calldata genesisPoolInfo,

        TokenAllocation calldata allocationInfo
```

```
    ) external nonReentrant returns (address genesisPool) {

        //.................code


        address pairAddress = pairFactory.getPair(nativeToken, _fundingToken,
_stable);

        if (pairAddress != address(0)) {

            require(IERC20(nativeToken).balanceOf(pairAddress) == 0, "!ZV");

            require(IERC20(_fundingToken).balanceOf(pairAddress) == 0,
"!ZV");

        }

       //.......................more code
```

Attack Scenario:

An attacker can observe a pending `createGenesisPool` transaction (or the `GenesisCreated` event). The attacker can:

- Call `pairFactory.createPair(nativeToken, fundingToken, stable)` to create a new, empty LP pair for the target tokens.
- Immediately after, transfer a minimal non-zero amount (e.g., 1 wei) of both `nativeToken` and `fundingToken` directly to this newly created pair contract address.

Impact

When the legitimate depo`sitNativeToken transaction executes,` pairFactory.getPair`will return the attacker's pair address. The` `require`(IERC20(token).balanceOf(pairAddress) == 0, "!ZV");`checks will then fail because the pair now holds a non-zero balance of tokens. This causes the` `legitimate`depositNativeToken` transaction to revert, resulting in:

- Denial of Service (DoS): Legitimate projects are repeatedly blocked from launching their intended Genesis Pools.
- Griefing: An attacker can perform this attack with minimal gas cost, making it a highly effective and low-cost method to disrupt the protocol's core functionality.

Recommended mitigation steps

Add pair ownership or Origin validation. Instead of just checking `balance == 0`, validate that your protocol created the pair, or that the pair was expected.

Blackhole mitigated

Status: Mitigation Complete. A courtesy review was conducted by MrPotatoMagic to verify that this is mitigated.

[M-10] ERC-2612 permit front-running in RouterV2 enables DoS of liquidity operations

Submitted by PolarizedLight

The `removeLiquidityWithPermit()` and `removeLiquidityETHWithPermit()` functions in RouterV2 are vulnerable to front-running attacks that consume user permit signatures, causing legitimate liquidity removal transactions to revert and resulting in gas fee losses and a DOS.

Finding description

The RouterV2 contract implements ERC-2612 permit functionality without protection against front-running attacks. The vulnerability stems from the deterministic nature of permit signatures and the lack of error handling when permit calls fail.

This vulnerability follows a well-documented attack pattern that has been extensively researched. According to Trust Security's comprehensive disclosure in January 2024: https://www.trust-security.xyz/post/permission-denied

"Consider, though, a situation where permit() is part of a contract call:

```
function deposit(uint256 amount, bytes calldata _signature) external {
    // This will revert if permit() was front-run
    token.permit(msg.sender, address(this), amount, deadline, v, r, s);
    // User loses this functionality when permit fails
    stakingContract.deposit(msg.sender, amount);
}
```

This function deposits in a staking contract on behalf of the user. But what if an attacker extracts the _signature parameters from the `deposit()` call and frontruns it with a direct `permit()`? In this case, the end result is harmful, since the user loses the functionality that follows the `permit()`."

In RouterV2's `removeLiquidityWithPermit()` follows this exact vulnerable pattern:

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L475-L475

```
IBaseV1Pair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);

// User loses liquidity removal functionality when permit fails

(amountA, amountB) = removeLiquidity(tokenA, tokenB, stable, liquidity, amountAMin, amountBMin, to, deadline);
```

As Trust Security formally identified: "In fact, any function call that unconditionally performs `permit()` can be forced to revert this way."

The root cause of the issue within the blackhole protocol is that RouterV2 makes unprotected external calls to LP token permit functions without any fallback mechanism or error handling:

```
// Line 475 - No error handling

IBaseV1Pair(pair).permit(msg.sender, address(this), value, deadline, v, r, s);
```

This creates a systematic vulnerability where any permit-based transaction can be griefed by extracting and front-running the permit signature.

Code Location

- `RouterV2.sol#L475` - `removeLiquidityWithPermit()` function
- `RouterV2.sol#L493` - `removeLiquidityETHWithPermit()` function

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L475-L475

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L493

Impact

This vulnerability enables attackers to systematically deny service to users attempting to remove liquidity through permit-based functions, resulting in direct financial losses and protocol dysfunction.

- Users lose transaction fees (typically `$5-50` per transaction depending on network congestion) when their transactions revert.
- Users must pay additional gas for traditional approve+remove patterns.
- Liquidity removal becomes unreliable, forcing users to abandon efficient permit-based operations.

Attack Prerequisites:

- Minimal: Attacker needs only basic mempool monitoring capability and gas fees.
- No special permissions or large capital requirements.
- Attack can be automated and executed repeatedly.
- Works against any user attempting permit-based operations.

Recommended mitigation steps

Implement a different method of permit handling by wrapping the permit call in a try-catch block and only reverting if both the permit fails and the current allowance is insufficient for the operation.

```
function removeLiquidityWithPermit(
    address tokenA,
    address tokenB,
    bool stable,
    uint liquidity,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline,
    bool approveMax,
    uint8 v,
    bytes32 r,
    bytes32 s
) external virtual override returns (uint amountA, uint amountB) {
    address pair = pairFor(tokenA, tokenB, stable);
    uint value = approveMax ? type(uint).max : liquidity;

    // Try permit, but don't revert if it fails
    try IBaseV1Pair(pair).permit(msg.sender, address(this), value, deadline,
v, r, s) {
```

```
        // Permit succeeded
    } catch {
        // Permit failed, check if we have sufficient allowance
        require(
            IBaseV1Pair(pair).allowance(msg.sender, address(this)) >=
liquidity,
            "Insufficient allowance"
        );
    }

    (amountA, amountB) = removeLiquidity(tokenA, tokenB, stable, liquidity,
amountAMin, amountBMin, to, deadline);
}
```

This approach follows the industry-standard mitigation pattern that successfully resolved this vulnerability across 100+ affected codebases.

Proof of Concept

Theoretical attack walkthrough:

1. Setup Phase:
    - Alice holds 1000 LP tokens in pair `0xABC...`
    - Alice wants to remove liquidity efficiently using `removeLiquidityWithPermit()`
    - Bob (attacker) monitors the mempool for permit-based transactions
2. Signature Creation:
    - Alice creates permit signature: `permit(alice, routerV2, 1000, deadline, nonce=5)`
    - Signature parameters: `v=27, r=0x123..., s=0x456...`
3. Transaction Submission:
    - Alice submits: `removeLiquidityWithPermit(tokenA, tokenB, false, 1000, 950, 950, alice, deadline, false, 27, 0x123..., 0x456...)`
    - Transaction enters mempool with 20 gwei gas price
4. Front-Running Execution:
    - Bob extracts parameters from Alice's pending transaction
    - Bob submits direct call: `IBaseV1Pair(0xABC...).permit(alice, routerV2, 1000, deadline, 27, 0x123..., 0x456...)` with 25 gwei gas

    - Bob's transaction mines first, consuming Alice's nonce (nonce becomes 6)
5. Victim Transaction Failure:
    - Alice's transaction executes but fails at line 475
    - RouterV2 calls `permit()` with already-used signature
    - LP token rejects due to invalid nonce (expects 6, gets signature for nonce 5)
    - Entire transaction reverts with "Invalid signature" or similar error
6. Attack Result:
    - Alice loses ~`$15` in gas fees (failed transaction cost)
    - Alice cannot remove liquidity via permit method
    - Bob spent ~`$3` in gas to grief Alice
    - Attack can be repeated indefinitely against any permit user

Impact Example:

- During high network activity (50+ gwei), failed transactions cost `$25-75` each
- Systematic attacks against 100 users = `$2,500-7,500` in direct user losses
- No recourse for affected users; losses are permanent

Blackhole mitigated

Status: Mitigation Complete. A courtesy review was conducted by MrPotatoMagic to verify that this is mitigated.

👨‍⚖️ Judge Review:

The re-evaluation was performed by the competition judge MrPotatoMagic, an experienced auditor whose credentials and prior work can be reviewed here.

> “
>
> *Approved, the two respective issues (M-05 and M-10 from the public Code4rena Blackhole DEX Report) outlined in this addendum have been mitigated.*
>
> ”
>
> — *Judge’s Comment*
