33AUDITS & CO.

BackGeoOracle Audit Report

Introduction

A time-boxed security review of the protocol was done by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits as the Lead Senior Security Researcher and Bumble as the Security Researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security researcher company and development group. We conduct audits a as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @solidityauditor.

About BackGeoOracle

BackGeoOracle is a Universal Oracle for DeFi built as a Uniswap v4 Hook. The system aims to provide a Proof of Stake (PoS) resistant oracle by leveraging Uniswap V4 hooks to mitigate price manipulation through automated backrunning of transactions. Key components of the system include:

- Single oracle per trading pair to reduce complexity and manipulation vectors
- Maximum tick spacing with full range liquidity to minimize manipulation potential
- Enhanced observation tracking through tick delta impact analysis
- Automated backrunning mechanism to counter price manipulation attempts
- Fee-based settlement system for hook deltas
- Support for both partial and full backrunning based on tick movement thresholds

Severity Definitions

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

Scope

The audit covered the following contracts: a8a4534316a61148a5ffe8b2cb0984f51ea69a13

Contract	Description	Link
BackGeoOracle.sol	Main oracle contract implementing the backrunning mechanism and observation tracking	View
Oracle.sol	Oracle contract for storing price observations	View

Findings Summary

ID	Title	Severity	Status
[H-01]	Backrun Compensation (ERC6909) Minted to Router Instead of User	High	Fixed
[H-02]	Incorrect Backrun Amount Calculation Leads to Inefficient Backrunning and Potential Forced Participation	High	Fixed
[L-01]	Backrun Amount Rounding Can Cause Small Swaps to Revert	Low	Acknowledged
[L-02]	Oracle Observations Missing Final Price State After Backruns	Low	Acknowledged

HIGH

[H-1] Backrun Compensation (ERC6909) Minted to Router Instead of User

Description

The `BackGeoOracle` hook implementation incorrectly mints ERC6909 tokens to the router contract instead of the original user when executing a backrun operation. This vulnerability stems from the hook receiving the router's address as the `sender` parameter, which represents the address that called the `PoolManager`. The current implementation lacks a mechanism to access the original user's address.

Impact

High - This vulnerability results in users being forced to pay the full swap amount without receiving their backrun compensation, leading to:

1. Users losing their specified amount in native currency
2. Backrun compensation tokens being permanently locked in the router contract
3. Disruption of the intended economic incentives for the backrun mechanism

Proof of Concept

The following test demonstrates the vulnerability:

```
function testMistakenBackrunCompensation() public {
    // Create a new user
    address user = makeAddr("user");

    // Give user ETH
    vm.deal(user, 1000 ether);

    // Get initial balances
    uint256 initialBalance0 = user.balance;
    uint256 initialBalance1 =
MockERC20(Currency.unwrap(currency1)).balanceOf(user);

    console2.log("Initial balance - amount0 (ETH): %d", initialBalance0);
    console2.log("Initial balance - amount1: %d", initialBalance1);

    // Approve tokens for swap
    MockERC20(Currency.unwrap(currency1)).approve(address(swapRouter),
type(uint256).max);

    // Execute swap as user
    vm.startPrank(user);
    bool zeroForOne = true;
    int256 amountSpecified = -500e18;
    BalanceDelta swapDelta = swap(key, zeroForOne, amountSpecified,
ZERO_BYTES);
    vm.stopPrank();

    // Get final balances
    uint256 finalBalance0 = user.balance;
    uint256 finalBalance1 =
MockERC20(Currency.unwrap(currency1)).balanceOf(user);

    console2.log("Final balance - amount0 (ETH): %d", finalBalance0);
    console2.log("Final balance - amount1: %d", finalBalance1);

    console2.log("Swap delta - amount0: %d", swapDelta.amount0());
    console2.log("Swap delta - amount1: %d", swapDelta.amount1());

    // Calculate net changes
    int256 netDelta0 = int256(finalBalance0) - int256(initialBalance0);
    int256 netDelta1 = int256(finalBalance1) - int256(initialBalance1);

    console2.log("Net delta - amount0: %d", netDelta0);
    console2.log("Net delta - amount1: %d", netDelta1);
    // Assert ERC6909 balance for the backrun compensation
    uint256 erc6909Balance = manager.balanceOf(address(swapRouter),
CurrencyLibrary.toId(CurrencyLibrary.ADDRESS_ZERO));
    console2.log("ERC6909 balance (ETH): %d", erc6909Balance);
    assertEq(erc6909Balance, 499950000000000000000, "PoolSwapTest receives
the 499.95 ETH as ERC6909 token, which was supposed to go to the user");
}
```

The vulnerability manifests in the following sequence:

1. User initiates a swap of 500 ETH for tokens through a router
2. The swap triggers a backrun due to large price movement
3. The hook executes the backrun and mints 499.95 ETH as ERC6909 tokens
4. The tokens are incorrectly minted to the router instead of the user due to the following code:

```
// In BackGeoOracle.sol
poolManager.mint(sender, CurrencyLibrary.toId(currencySpecified),
uint256(int256(backAmountSpecified)));
```

As a result:

- User spends 500 ETH
- User receives only ~9.992e16 tokens instead of the expected 1.237e20 tokens
- User loses their 499.95 ETH backrun compensation
- The router contract holds the ERC6909 tokens that should have been sent to the user

Recommendation

Implement the solution described in the Uniswap V4 documentation for accessing the original sender address. The hook should:

1. Maintain a registry of verified routers that implement the `msgSender()` function
2. Call `msgSender()` to retrieve the original user's address
3. Mint the ERC6909 tokens to the original user's address instead of the router

Reference: Uniswap V4 Documentation - Accessing msg.sender using hook

Resolution:

Fixed.

[H-2] Incorrect Backrun Amount Calculation Leads to Inefficient Backrunning and Potential Forced Participation

Description

The `BackGeoOracle::_backrun` function contains a critical flaw in its backrun amount calculation logic. The function incorrectly utilizes the original `amountSpecified` parameter instead of the actual swap

`BalanceDelta` when determining backrun amounts. This design flaw creates a fundamental mismatch between the actual price movement and the attempted backrun operation. Furthermore, when users possess the unspecified currency in their wallet (e.g., USDC when selling ETH), they may be involuntarily subjected to backruns with incorrect and unintended amounts.

Impact

High - This vulnerability significantly impacts the protocol's core functionality and user experience:

1. Failed or inefficient backruns that fail to properly stabilize prices, undermining the oracle's primary purpose
2. Users being forced to participate in backruns with incorrect amounts, potentially leading to unexpected losses
3. Potential manipulation of the backrun mechanism by malicious actors
4. Incorrect price stabilization, which is critical for the oracle's functionality and reliability

Proof of Concept

The vulnerability can be demonstrated through the following scenarios:

Scenario 1: Inefficient Backrun Due to Amount Mismatch

```
// Initial swap parameters
amountSpecified = -1000 ETH  // User attempts to sell 1000 ETH for USDC

// Actual execution due to low liquidity
actualSwapAmount = 500 ETH   // Only 500 ETH is actually swapped

// Backrun calculation (incorrect)
backrunAmount = 999 ETH      // 9999/10000 of original 1000 ETH

// Result: Backrun fails or executes inefficiently because:
// - Price movement was based on 500 ETH
// - Backrun attempts to move 999 ETH
// - Creates mismatch between actual price impact and attempted backrun
```

Scenario 2: Forced Participation with Incorrect Amounts

```
// User has USDC in wallet and attempts to swap
initialSwap = 1000 ETH for USDC
actualSwap = 500 ETH         // Only 500 ETH swapped due to liquidity

// Backrun calculation (incorrect)
backrunAmount = 999 ETH      // Based on original 1000 ETH

// If user has sufficient USDC:
// - Forced to participate in backrun
// - Amount calculated in wrong currency (ETH instead of USDC)
// - Results in unintended backrun participation
```

The root cause of this vulnerability lies in the `_backrun` function's implementation:

```
function _backrun(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams memory params,
    BalanceDelta swapDelta
) private returns (BalanceDelta hookDelta, bool shouldBackrun) {
    // ... existing tick delta calculation ...

    if (shouldBackrun) {
        // Incorrect: Uses original amountSpecified instead of swapDelta
        params.amountSpecified = params.amountSpecified * 9999 / 10000;

        params.zeroForOne = !params.zeroForOne;
        params.amountSpecified = -params.amountSpecified;
        // ... rest of the function
    }
}
```

Recommendation

The following solutions are proposed to address this vulnerability:

Solution 1: Pass Swap Delta to Backrun (Recommended)

```
// In BackGeoOracle.sol
function _afterSwap(
    address sender,
    PoolKey calldata key,
    IPoolManager.SwapParams calldata params,
    BalanceDelta swapDelta,
    bytes calldata
) internal override onlyPoolManager returns (bytes4, int128) {
    // Pass swapDelta to _backrun for accurate amount calculation
    (BalanceDelta hookDelta, bool isBackrun) = _backrun(sender, key,
params, swapDelta);

    if (isBackrun) {
        // ... rest of the function
    }
}

function _backrun(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams memory params,
    BalanceDelta swapDelta
) private returns (BalanceDelta hookDelta, bool shouldBackrun) {
    // ... existing tick delta calculation ...

    if (shouldBackrun) {
        // Use actual swap delta for backrun amount calculation
        params.amountSpecified = swapDelta.amount0() * 9999 / 10000;

        params.zeroForOne = !params.zeroForOne;
        params.amountSpecified = -params.amountSpecified;
        // ... rest of the function
    }
}
```

Solution 2: Implement Slippage Protection

```
// In BackGeoOracle.sol
function _backrun(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams memory params,
    BalanceDelta swapDelta
) private returns (BalanceDelta hookDelta, bool shouldBackrun) {
    // ... existing tick delta calculation ...

    if (shouldBackrun) {
        // Add slippage check to prevent incorrect backruns
        int128 intendedAmount = params.amountSpecified;
        int128 actualAmount = swapDelta.amount0();
        if (actualAmount * 10000 / intendedAmount < 9999) {
            // Slippage too high, skip backrun
            return (BalanceDeltaLibrary.ZERO_DELTA, false);
        }

        params.amountSpecified = swapDelta.amount0() * 9999 / 10000;
        // ... rest of the function
    }
}
```

The first solution is recommended as it provides the most direct and effective fix by:

1. Ensuring backrun amounts are proportional to actual price movement
2. Preventing forced participation in backruns with incorrect amounts
3. Calculating backrun amounts in the correct currency
4. Minimizing economic impact by using actual swap amounts

Resolution:

Fixed.

LOW

[L-01] Backrun Amount Rounding Can Cause Small Swaps to Revert

Description

The `BackGeoOracle::_backrun` function contains a precision issue in its amount calculation logic that can cause small swap transactions to revert. The vulnerability arises from the backrun amount calculation mechanism, which multiplies the original amount by 9999/10000 (or a similar fraction in partial backrun cases). While this 1 basis point reduction is intended to prevent rounding issues, for very small amounts (1 wei), the multiplication followed by integer division can result in zero, triggering the `SwapAmountCannotBeZero` check in `PoolManager::swap`.

Impact

Low - The impact is limited to edge cases involving minimal swap amounts:

1. Only affects swaps of 1 wei that trigger a backrun
2. Results in transaction reverts rather than fund loss
3. Does not impact normal trading operations
4. Minimal effect on protocol functionality

Proof of Concept

The vulnerability can be demonstrated through the following sequence:

```
// Initial swap parameters
amountSpecified = -1; // 1 wei exact input

// Backrun calculation
backrunAmount = (-1 * 9999) / 10000; // Results in -0.9999
// Integer division rounds down to 0

// Transaction reverts with:
// Error: SwapAmountCannotBeZero
```

The root cause lies in the `_backrun` function's implementation:

```
function _backrun(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams memory params,
    BalanceDelta swapDelta
) private returns (BalanceDelta hookDelta, bool shouldBackrun) {
    // ... existing tick delta calculation ...
    if (shouldBackrun) {
        // For 1 wei amounts, this calculation results in 0
        params.amountSpecified = params.amountSpecified * 9999 / 10000;

        params.zeroForOne = !params.zeroForOne;
        params.amountSpecified = -params.amountSpecified;
        // ... rest of the function
    }
}
```

Recommendation

Implement a minimum threshold check to skip backruns for very small amounts:

```
function _backrun(
    address,
    PoolKey calldata key,
    IPoolManager.SwapParams memory params,
    BalanceDelta swapDelta
) private returns (BalanceDelta hookDelta, bool shouldBackrun) {
    // ... existing tick delta calculation ...

    if (shouldBackrun) {
        // Skip backrun for amounts that would round to zero
        if (params.amountSpecified * 9999 / 10000 == 0) {
            return (BalanceDeltaLibrary.ZERO_DELTA, false);
        }

        params.amountSpecified = params.amountSpecified * 9999 / 10000;
        params.zeroForOne = !params.zeroForOne;
        params.amountSpecified = -params.amountSpecified;
        // ... rest of the function
    }
}
```

Resolution:

Team acknowledged the issue as impact is low.

[L-02] Oracle Observations Missing Final Price State After Backruns

Description

The `BackGeoOracle` contract implements an incomplete price observation mechanism that fails to record the final price state after backrun operations. The contract updates price observations in `_beforeSwap` but omits the crucial update after backrun execution in `_afterSwap`. This implementation gap results in only the initial tick being recorded in the oracle observations when a swap triggers a backrun, with the final tick being recorded only upon the next swap operation, creating a temporary gap in price data accuracy.

Impact

Low - The impact is mitigated by several factors:

1. Backruns are only triggered for significant tick movements (> 912 ticks)
2. Backrun amounts are proportionally limited to 99.99% of the original swap
3. Maximum price impact from backruns is capped at 10% of the original swap
4. The resulting gap in TWAP calculations is minimal and short-lived

Proof of Concept

The vulnerability can be demonstrated through the following sequence:

```
// Initial state
initialTick = 1000
targetTick = 1920
tickDelta = 920  // > MIN_ABS_TICK_MOVE (912)

// Observation recording sequence
1. _beforeSwap records tick 1000
2. Swap executes, moving tick to 1920
3. Backrun triggers and executes
4. _afterSwap omits observation update
5. Final tick only recorded on next swap

// Result: Gap in price observations between backrun and next swap
```

The root cause lies in the `_afterSwap` function's implementation:

```
function _afterSwap(
    address sender,
    PoolKey calldata key,
    IPoolManager.SwapParams calldata params,
    BalanceDelta delta,
    bytes calldata
) internal override onlyPoolManager returns (bytes4, int128) {
    // ... existing code ...

    if (isBackrun) {
        // Backrun executes but no observation update
        // ... existing backrun code ...
    }

    return (BaseHook.afterSwap.selector, -backAmountUnspecified);
}
```

Recommendation

Implement one of the following solutions to ensure accurate price observations:

Update Observations After Backrun

```
function _afterSwap(
    address sender,
    PoolKey calldata key,
    IPoolManager.SwapParams calldata params,
    BalanceDelta delta,
    bytes calldata
) internal override onlyPoolManager returns (bytes4, int128) {
    // ... existing code ...

    if (isBackrun) {
        // ... existing backrun code ...

        // Update oracle observations after backrun
        _updatePool(key);
    }

    return (BaseHook.afterSwap.selector, -backAmountUnspecified);
}
```

Resolution:

Team acknowledged issue but impact is low.