VANGUARD Security Review

Reviewer: Muhammad Faran **Date:** Feb 22, 2025 **Repo:** CodeHawks-Contests/2025-09-bid-beasts

---

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

About the Project

Vanguard is a Uniswap V4 hook implementation that provides anti-bot protection for token launches. The protocol implements a phased fee structure to prevent manipulation during the initial launch period, with configurable limits, cooldowns, and penalties for excessive selling.

The hook intercepts swap operations and enforces dynamic fees based on the launch phase: - Phase 1: Strict limits on sell amounts with high penalties for violations - Phase 2: Relaxed limits with moderate penalties - Post-launch: Standard Uniswap fees apply

This creates a fair launch environment that protects early participants while allowing natural price discovery.

Project Scope

- [TokenLaunchHook](./src/TokenLaunchHook.sol)
- [DeployHookScript](./script/deployLaunchHook.s.sol)

Actors

There are 2 main actors in this protocol:

1. owner:
- RESPONSIBILITIES:

- deploys hook contract with launch parameters (durations, limits, penalties)
- can modify fee parameters via administrative functions
- has full administrative control over launch configuration
- monitors launch progress

2. swapper:

- RESPONSIBILITIES:
- can execute swaps through Uniswap V4 pools utilizing this hook
- must comply with per-block swap limits during launch phases
- LIMITATIONS:
- cannot bypass hook-enforced limits and penalties
- cannot modify launch parameters
- subject to pool liquidity constraints
- must use pools that implement the hook

FINDING: Deployment Failure Due to Mismatched Hook Flags

Description

The deployLaunchHook.s.sol script is responsible for mining a salt to generate a specific contract address that encodes the hook's permissions. Uniswap V4 requires the hook's address to contain specific "flags" (bits set to 1) that strictly match the permissions returned by the hook's getHookPermissions() function.

However, the deployment script requests the wrong flag:

1. **The Script** uses Hooks.BEFORE_INITIALIZE_FLAG.
2. **The Contract** (TokenLaunchHook.sol) implements logic for afterInitialize and sets afterInitialize: true in its permissions struct.

When the PoolManager (or the deployment verification) attempts to validate the hook, it will detect that the address claims to handle beforeInitialize (via the flag), but the contract logic explicitly states it does not (beforeInitialize:

false). This mismatch causes the deployment or pool initialization transaction to revert.

```
    // deployLaunchHook.s.sol
    function run() public {
        // hook contracts must have specific flags encoded in the address
        //@audit: This sets the bit for BEFORE_INITIALIZE
        uint160 flags = uint160(Hooks.BEFORE_SWAP_FLAG |
Hooks.BEFORE_INITIALIZE_FLAG);

        // ... HookMiner finds an address starting with specific bits ...
    }
```

Risk

Likelihood: High

- This is a hard-coded configuration error.
- Every attempt to run the deployment script `forge script script/deployLaunchHook.s.sol` will result in a hook address that is incompatible with the contract code.

Impact: Critical

- **Deployment Blocked:** The protocol cannot be deployed or initialized.
- **Protocol Failure:** The `PoolManager` will reject the hook, preventing any liquidity pools from being created with this hook attached.

Proof of Concept

1. **Contract Code (TokenLaunchHook.sol)**: The contract explicitly defines its permissions to use `afterInitialize`.

```
function getHookPermissions() public pure override returns (Hooks.Permissions
memory) {
    return Hooks.Permissions({
        beforeInitialize: false, // <--- Contract says FALSE
        afterInitialize: true,   // <--- Contract says TRUE
        // ...
    });
}
```

2. **Deployment Script (deployLaunchHook.s.sol)**: The script attempts to mine an address for `beforeInitialize`.

```
// The script forces the address to have the BEFORE_INITIALIZE bit set
uint160 flags = uint160(Hooks.BEFORE_SWAP_FLAG |
Hooks.BEFORE_INITIALIZE_FLAG);
```

3. **The Failure**: When `HookMiner` generates the address, it will look like `0x...` where the leading bits indicate `BEFORE_INITIALIZE`. When the

PoolManager checks this address against getHookPermissions, the validation fails because the flags do not match the return struct.

4. **Validating My Claim**: In the run function of TokenLaunchHookUnit.t.sol there is the following line for the flag;

```
uint160 flags = uint160(Hooks.AFTER_INITIALIZE_FLAG |
Hooks.BEFORE_SWAP_FLAG);
```

When it has Hooks.AFTER_INITIALIZE_FLAG the test suite runs. But when the I replaced it with Hooks.BEFORE_INITIALIZE_FLAG the test suite failed to run and I got the following error;

```
Failing tests:
Encountered 1 failing test in
test/TokenLaunchHookUnit.t.sol:TestTokenLaunchHook
[FAIL:
HookAddressNotValid(0xB68723B2b5E1Ef43e9a32dC5708d65A5234c6080)]
setUp() (gas: 0)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Recommended Mitigation

Update the flags variable in deployLaunchHook.s.sol to use Hooks.AFTER_INITIALIZE_FLAG, matching the logic inside TokenLaunchHook.sol.

```
    function run() public {
        // hook contracts must have specific flags encoded in the address
-       uint160 flags = uint160(Hooks.BEFORE_SWAP_FLAG |
Hooks.BEFORE_INITIALIZE_FLAG);
+       uint160 flags = uint160(Hooks.BEFORE_SWAP_FLAG |
Hooks.AFTER_INITIALIZE_FLAG);

        // Mine a salt that will produce a hook address with the correct flags
        bytes memory constructorArgs = abi.encode(
```

FINDING: Compilation Failure Due to Undefined CREATE2_FACTORY Variable

Description

The deployLaunchHook.s.sol script relies on the HookMiner library to pre-calculate (mine) the contract address before deployment. This process

requires the address of the deterministic CREATE2 factory that will be used for the actual deployment.

In the provided code, the constant `CREATE2_FACTORY` is declared but is commented out. However, this variable is subsequently referenced in the `run()` function inside the `HookMiner.find(...)` call. This causes the Solidity compiler to fail immediately with an "Undeclared identifier" error, making it impossible to compile or execute the deployment script.

```
// deployLaunchHook.s.sol

    // CREATE2 factory address for deploying hooks with specific flags
    // address constant CREATE2_FACTORY =
0x4e59b44847b379578588920cA78FbF26c0B4956C; <--- COMMENTED OUT

    function run() public {
        // ...

        // @audit Variable 'CREATE2_FACTORY' is used here but is not defined in
scope
        (address hookAddress, bytes32 salt) =
            HookMiner.find(CREATE2_FACTORY, flags,
type(TokenLaunchHook).creationCode, constructorArgs);
```

Risk

Likelihood: Low

- The compiler will strictly enforce variable declaration rules. The script cannot be run or tested in its current state.

Impact: Low

- **Denial of Service (Development):** Developers and auditors cannot compile the codebase or run the deployment scripts.
- **Deployment Failure:** The protocol cannot be deployed to any network until this syntax error is resolved.

Proof of Concept

1. **Navigate to** `script/deployLaunchHook.s.sol`.
2. **Observe Line 14-15**: The `CREATE2_FACTORY` definition is commented out.
3. **Observe Line 34**: The code attempts to access `CREATE2_FACTORY`.
4. **Run Command**: `forge build` or `forge script script/deployLaunchHook.s.sol`.
5. **Result**: Compiler Error.

```
Error: Undeclared identifier.
--> script/deployLaunchHook.s.sol:34:28:
```

```
|
| HookMiner.find(CREATE2_FACTORY, flags,
type(TokenLaunchHook).creationCode, constructorArgs);
|                ^^^^^^^^^^^^^^^^^
```

Recommended Mitigation

Uncomment the definition of the CREATE2_FACTORY constant.

```
-   // address constant CREATE2_FACTORY =
0x4e59b44847b379578588920cA78FbF26c0B4956C;
+   address constant CREATE2_FACTORY =
0x4e59b44847b379578588920cA78FbF26c0B4956C;
```

FINDING: Denial of Service (DoS) via Shared Router Address Limit Collisions

Description

The TokenLaunchHook is designed to enforce swap limits and penalties on individual users during the launch phases. It tracks user activity using the sender parameter provided by the _beforeSwap function.

However, in the Uniswap V4 architecture, the sender address calling the PoolManager is typically a **Swap Router contract** (such as the Universal Router), not the actual end-user's wallet address (EOA).

Because the hook relies on addressSwappedAmount[sender] to calculate limits, it treats the Router contract as a single "user." As soon as the aggregate volume of all users swapping through that Router exceeds the limit, the Router itself is flagged as having reached the cap. Consequently, **all subsequent users** attempting to swap through that Router will be unfairly penalized or blocked, resulting in a Denial of Service for legitimate participants.

```
    function _beforeSwap(address sender, PoolKey calldata key, SwapParams
calldata params, bytes calldata)
        internal
        override
        returns (bytes4, BeforeSwapDelta, uint24)
    {
        // ...

        // @audit 'sender' is the Router address, not the User address
        addressSwappedAmount[sender] += swapAmount;
        addressLastSwapBlock[sender] = block.number;
```

Risk

Likelihood: High

- The vast majority of Uniswap trades are executed via Router contracts to handle multi-hop swaps and slippage protection. Direct calls to `PoolManager` by users are rare.

Impact: Critical

- **Protocol Broken:** The core functionality (individual limits) fails immediately.
- **Denial of Service:** Legitimate users are unable to swap without paying penalties intended for whales/bots because the Router's shared limit is exhausted instantly.
- **Unfair Penalties:** Innocent users will be charged the "Penalty Fee" simply because they used a popular Router.

Proof of Concept

Scenario Parameters:

- **Phase 1 Limit:** 1,000 Tokens.
- **Phase 1 Penalty:** 5%.

The Exploit Flow:

1. **Alice** wants to buy 600 tokens. She sends a transaction via the `UniversalRouter`.
- The Hook sees `sender` = `UniversalRouter`.
- `addressSwappedAmount[UniversalRouter]` becomes 600.
2. **Bob** wants to buy 500 tokens. He sends a transaction via the `UniversalRouter`.
- The Hook sees `sender` = `UniversalRouter`.
- The Hook checks: `addressSwappedAmount[UniversalRouter]` (600) + `currentSwap` (500) = 1,100.
- **1,100 > 1,000 (Limit)**.
3. **Result:** The Hook flags the transaction as "Over Limit."
- Bob is charged the **5% Penalty Fee** even though he individually only bought 500 tokens (well under the limit).
- If the limit was a "Hard Cap" (reverting), Bob's transaction would simply fail.

Recommended Mitigation

The hook must identify the origin of the transaction rather than the immediate caller.

Option 1: Use tx.origin (Simplest) Replace sender with tx.origin to track the EOA that signed the transaction. *Note: This may not work correctly for Account Abstraction / Smart Contract Wallets, but it fixes the Router issue for standard users.*

```
-       if (addressLastSwapBlock[sender] > 0) {
+        if (addressLastSwapBlock[tx.origin] > 0) {
-           uint256 blocksSinceLastSwap = block.number -
addressLastSwapBlock[sender];
+           uint256 blocksSinceLastSwap = block.number -
addressLastSwapBlock[tx.origin];
            if (blocksSinceLastSwap < phaseCooldown) {
                applyPenalty = true;
            }
        }
-       if (!applyPenalty && addressSwappedAmount[sender] + swapAmount >
maxSwapAmount) {
+       if (!applyPenalty && addressSwappedAmount[tx.origin] + swapAmount >
maxSwapAmount) {
            applyPenalty = true;
        }

-       addressSwappedAmount[sender] += swapAmount;
+       addressSwappedAmount[tx.origin] += swapAmount;
-       addressLastSwapBlock[sender] = block.number;
+       addressLastSwapBlock[tx.origin] = block.number;
```

Option 2: Require User Address via HookData (Robust) Modify the hook to decode the actual user's address passed via the bytes calldata parameter (HookData) from the Router. This requires the frontend/Router to support passing this data.

FINDING: Broken Phase Reset Logic Causes Persistent Swap Limits (DoS)

Description

The TokenLaunchHook is designed to reset user swap limits when the protocol transitions from Phase 1 to Phase 2. This is intended to give users a "fresh start" with the new phase's limits.

However, the _resetPerAddressTracking function is implemented incorrectly. In Solidity, deleting a key in a mapping (like addressSwappedAmount[address(0)] = 0) does **not** clear the entire mapping. It only clears that specific key. Consequently, all user swap history from Phase 1 persists into Phase 2.

Code at Fault:

```
function _resetPerAddressTracking() internal {
    // @audit Only clears the zero address. All actual user data remains.
    addressSwappedAmount[address(0)] = 0;
    addressLastSwapBlock[address(0)] = 0;
}
```

Risk

Likelihood: High

- This logic runs on every phase transition. The bug is intrinsic to how Solidity mappings work.

Impact: High

- **Denial of Service:** Users active in Phase 1 will enter Phase 2 with their limits already partially or fully consumed.
- **Broken Core Logic:** The concept of “Phased Limits” is fundamentally broken as usage accumulates globally rather than per-phase.

Proof of Concept

Add this function to your TestTokenLaunchHook contract. It uses the exact same variable naming and struct initialization as your working tests.

```
function test_PoC_BrokenLimitReset_UserUsageCarriesOver() public {
    // 1. SETUP: Give user1 funds
    vm.deal(user1, 10 ether);

    // Define swap params (Selling 0.05 ETH)
    // Phase 1 Limit is 1% of 10 ETH = 0.1 ETH. We use 50% of it.
    uint256 swapAmountPhase1 = 0.05 ether;

    SwapParams memory params = SwapParams({
        zeroForOne: true,
        amountSpecified: -int256(swapAmountPhase1),
        sqrtPriceLimitX96: TickMath.MIN_SQRT_PRICE + 1
    });

    PoolSwapTest.TestSettings memory testSettings =
        PoolSwapTest.TestSettings({takeClaims: false, settleUsingBurn: false});

    // 2. ACTION: User1 swaps in Phase 1
    vm.startPrank(user1);
    swapRouter.swap{value: swapAmountPhase1}(key, params, testSettings,
ZERO_BYTES);
    vm.stopPrank();
```

```
        // Check Phase 1 usage is recorded
        // Note: Checking 'swapRouter' address due to the known Router DoS bug
        uint256 usageAfterPhase1 =
antiBotHook.addressSwappedAmount(address(swapRouter));
        assertEq(usageAfterPhase1, swapAmountPhase1, "Phase 1 usage should be
recorded");

        // 3. TRANSITION: Move to Phase 2
        // Phase 1 duration is 100 blocks. We roll past it.
        vm.roll(block.number + phase1Duration + 1);

        // 4. ACTION: User1 swaps in Phase 2
        // This triggers _beforeSwap -> newPhase detected ->
_resetPerAddressTracking called
        uint256 swapAmountPhase2 = 0.01 ether;

        // Update params for the new amount
        params.amountSpecified = -int256(swapAmountPhase2);

        vm.startPrank(user1);
        swapRouter.swap{value: swapAmountPhase2}(key, params, testSettings,
ZERO_BYTES);
        vm.stopPrank();

        // 5. ASSERTION: Check if usage was reset
        uint256 usageTotal =
antiBotHook.addressSwappedAmount(address(swapRouter));

        // EXPECTED (If Reset worked): usageTotal == swapAmountPhase2 (0.01)
        // ACTUAL (Bug): usageTotal == swapAmountPhase1 + swapAmountPhase2
(0.06)

        console.log("Expected Usage if Reset worked:", swapAmountPhase2);
        console.log("Actual Usage (Accumulated):   ", usageTotal);

        assertGt(usageTotal, swapAmountPhase2, "FAIL: Phase 1 usage was not
cleared!");
        assertEq(usageTotal, swapAmountPhase1 + swapAmountPhase2, "FAIL:
Limits are cumulative across phases");
    }
```

Recommended Mitigation

Since mappings cannot be cleared in O(1) time in Solidity, you must include the **Phase ID** in the mapping key. This effectively creates a new, empty limits bucket for every phase.

1. Update the Mapping:

```
    // Change mapping(address => uint256) to:
    mapping(uint256 => mapping(address => uint256)) public
phaseAddressSwappedAmount;
```

2. Update _beforeSwap:

```
    // Remove _resetPerAddressTracking() call entirely.

    // access data using currentPhase
    if (!applyPenalty && phaseAddressSwappedAmount[currentPhase][sender] +
swapAmount > maxSwapAmount) {
        // ...
    }
    phaseAddressSwappedAmount[currentPhase][sender] += swapAmount;
```

FINDING: View Function Returns Incorrect Phase at Boundary Blocks, Leading to Unexpected Penalties

Description

The getCurrentPhase function is intended to report the current status of the launch (Phase 1, Phase 2, or Phase 3) to external callers, such as frontends or other smart contracts. This data is typically used to estimate fees and limits before a user submits a transaction.

However, there is a logic inconsistency between how getCurrentPhase calculates the phase and how the core logic _beforeSwap calculates it.

- **_beforeSwap (Execution):** Uses inclusive comparison (<=).
- Logic: if (blocksSinceLaunch <= phase1Duration)
- **getCurrentPhase (View):** Uses strict inequality (<).
- Logic: if (blocksSinceLaunch < phase1Duration)

At the exact boundary block where blocksSinceLaunch == phase1Duration, the view function reports that the protocol has moved to the **next phase**, while the execution logic still enforces the **previous phase**.

This mismatch applies to both the transition from Phase 1 to 2, and Phase 2 to 3.

```
    // TokenLaunchHook.sol

    function getCurrentPhase() public view returns (uint256) {
```

```
        // ...
        uint256 blocksSinceLaunch = block.number - launchStartBlock;

        // @audit Uses '<' instead of '<='
        if (blocksSinceLaunch < phase1Duration) {
            return 1;
        } else if (blocksSinceLaunch < phase1Duration + phase2Duration) { // @audit
Same error here
            return 2;
        } else {
            return 3;
        }
    }
```

Risk

Likelihood: Low

- This occurs exactly on the specific blocks that mark the end of a phase. While infrequent, in a high-volume launch, it is statistically probable that transactions will be included in these blocks.

Impact: Low

- **Financial Discrepancy:**
- **Phase 1 -> 2:** User sees Phase 2 (lower fee/penalty). User submits tx. User pays Phase 1 (higher fee/penalty).
- **Phase 2 -> 3:** User sees Phase 3 (Standard 0.3% Fee). User submits tx. User pays Phase 2 (e.g., 2% Penalty).
- **Failed Transactions:** If the Phase 1 limit is stricter than Phase 2, a user might submit a swap size valid for Phase 2 (as reported by the view) which reverts under Phase 1 rules.

Proof of Concept

Scenario:

- launchStartBlock = 1000.
- phase1Duration = 100 blocks.
- phase1Penalty = 5%.
- phase2Penalty = 2%.

At Block 1100:

1. **Calculation:** blocksSinceLaunch = 1100 - 1000 = 100.
2. **Frontend Check (getCurrentPhase):**

- 100 < 100 evaluates to **FALSE**.

- Function returns **Phase 2**.
- **User Expectation:** “I will pay 2% penalty.”

3. **Execution (_beforeSwap):**

- 100 <= 100 evaluates to **TRUE**.
- Function enforces **Phase 1**.
- **Actual Result:** User pays **5% penalty**.

Recommended Mitigation

Update the comparison operators in getCurrentPhase to use <= to strictly match the logic in _beforeSwap.

```
    function getCurrentPhase() public view returns (uint256) {
        if (launchStartBlock == 0) return 0;
        uint256 blocksSinceLaunch = block.number - launchStartBlock;
-       if (blocksSinceLaunch < phase1Duration) {
+        if (blocksSinceLaunch <= phase1Duration) {
            return 1;
-       } else if (blocksSinceLaunch < phase1Duration + phase2Duration) {
+        } else if (blocksSinceLaunch <= phase1Duration + phase2Duration) {
            return 2;
        } else {
            return 3;
        }
    }
```

FINDING: totalPenaltyFeesCollected is Never Updated, Leading to Failed Accounting

Description

The contract declares a public state variable totalPenaltyFeesCollected, intended to track the cumulative value of penalty fees levied against users during the launch phases.

However, in the _beforeSwap function, while the penalty fee is applied via feeOverride, there is no logic to calculate the fee amount or update this variable. As a result, totalPenaltyFeesCollected will permanently remain 0, breaking the protocol’s ability to track the efficacy of its anti-bot measures or report revenue.

Furthermore, the implementation uses LPFeeLibrary.OVERRIDE_FEE_FLAG, which directs the penalty fees to the **Liquidity Providers** of the pool. If the intention of totalPenaltyFeesCollected was for the *Protocol* to withdraw these fees later, the current implementation fails to capture that revenue entirely.

```
// TokenLaunchHook.sol

// @audit Variable declared here
uint256 public totalPenaltyFeesCollected;

function _beforeSwap(...) {
    // ...
    if (applyPenalty) {
        feeOverride = uint24((phasePenaltyBps * 100));
        // @audit MISSING: totalPenaltyFeesCollected += calculatedFeeAmount;
    }
    return (..., feeOverride | LPFeeLibrary.OVERRIDE_FEE_FLAG);
}
```

Risk

Likelihood: High

- The variable is mathematically unreachable in the current code.

Impact: Dead Code

- **Broken Metrics:** The owner cannot see how much penalty has been enforced.
- **Potential Revenue Loss:** If this variable was intended to track funds withdrawable by the owner, those funds are currently lost to LPs.

Recommended Mitigation

If the goal is simply to track how much LPs earned from penalties, implement the math to estimate the fee based on the swap amount.

Note: Calculating exact fees in beforeSwap can be complex depending on whether the swap is ExactInput or ExactOutput.

```
    if (applyPenalty) {
        feeOverride = uint24((phasePenaltyBps * 100));

+       // Approximate fee calculation (Assuming ExactInput for simplicity)
+       uint256 feeAmount = (swapAmount * phasePenaltyBps) / 10000;
+       totalPenaltyFeesCollected += feeAmount;
    }
```

If the goal was for the **Protocol** to keep the fees, the entire fee mechanism needs to be changed (e.g., using Hooks.beforeDonate or taking a cut via BeforeSwapDelta).
