Three Sigma

Code Audit

HOTKEY

HotCurves Token Launcher

Disclaimer

Code Audit

HotCurves Token Launcher

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

HotCurves Token Launcher

Table of Contents

Summary

Code Audit

HotCurves Token Launcher

Summary

Three Sigma audited HotCurves in a 2 person week engagement. The audit was conducted from 17/02/2025 to 21/02/2025.

Protocol Description

HotCurves.fun is a decentralized launchpad built upon Uniswap V4 infrastructure, providing a secure and automated environment for omnichain token launches and decentralized token swaps. Central to its functionality is the $HOTKEY token, which serves both as the ecosystem's governance token and as the liquidity pair for all launched tokens, ensuring a unified and trustless trading experience. HotCurves.fun leverages advanced on-chain mechanisms to maintain liquidity permanence, mitigate manipulation risks, and uphold price stability, enhancing fairness and reliability across its platform.

Scope

Code Audit

HotCurves Token Launcher

Scope

Filepath	nSLOC
v3-contracts-hotcurves/ethereum/Multicall.sol	333
v3-contracts-hotcurves/ethereum/contracts/HotCurves.sol	283
v3-contracts-hotcurves/ethereum/HotCurvesFactory.sol	98
v3-contracts-hotcurves/ethereum/contracts/ERC20.sol	85
v3-contracts-hotcurves/ethereum/contracts/HotCurvesInfo.sol	84
v3-contracts-hotcurves/ethereum/lib/Address.sol	81
v3-contracts-hotcurves/ethereum/lib/SafeERC20.sol	58
v3-contracts-hotcurves/ethereum/contracts/Coin.sol	45
v3-contracts-hotcurves/ethereum/lib/SafeMath.sol	32
v3-contracts-hotcurves/ethereum/contracts/Ownable.sol	27
v3-contracts-hotcurves/ethereum/abstract/ReentrancyGuard.sol	15
v3-contracts-hotcurves/ethereum/context/Context.sol	6
Total	**1147**

Assumptions

External libraries are considered safe.

Methodology

Code Audit

HotCurves Token Launcher

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

HotCurves Token Launcher

Project Dashboard

Application Summary

Name	HotCurves
Commit	cbe0777
Language	Solidity
Platform	Ethereum, BSC, Base, Arbitrum

Engagement Summary

Timeline	17/02/2025 to 21/02/2025
№ of Auditors	2
Review Time	2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	2	2	0
High	2	2	0
Medium	0	0	0
Low	2	2	0

None	5	5	0

Category Breakdown

Suggestion	5
Documentation	0
Bug	6
Optimization	0
Good Code Practices	0

Findings

Code Audit

HotCurves Token Launcher

Findings

3S-HotCurves-C01

Frontrunning Pool Initialization Enables Price Manipulation Attack During Liquidity Migration

Id	3S-HotCurves-C01
Classification	Critical
Severity	Critical
Likelihood	High
Category	Bug
Status	Addressed in #9150c03.

Description

When the **`HotCurves`** contract's internal calculation of market cap exceeds the **`finalMarketCap`**, the function **`finalize()`** is triggered. Within **`finalize()`**, a Uniswap V3 pool for **`(HotKey, Token)`** is created and initialized if it does not already exist, using **`createAndInitializePoolIfNecessary()`**. However, if an attacker has preemptively created and initialized that pool at a highly skewed price, the contract will not overwrite the attacker's price.

After detecting the pool is already initialized, the contract simply calls **`mintPosition()`** to provide all of its **`HotKey`** and **`Token`** liquidity with no price or slippage checks (**`amount0Min = 0`** and **`amount1Min = 0`**). As a result, the skewed price set by the attacker remains in place, causing the contract to deposit liquidity at an extremely unfavorable ratio.

Proof of Concept (PoC)

See the PoC here

1. The attacker observes that the market cap is approaching the threshold of 11,5 ETH, which triggers **`finalize()`**.

2. The attacker notices that upcoming buy transactions will push the market cap from 11.49 ETH to 11.51 ETH, surpassing the 11.5 ETH threshold and triggering **`finalize()`**.

3. Before the contract reaches this threshold, the attacker manually deploys a **`(hotKey, Token)`** pool on Uniswap V3 with a 1% fee tier **`(fee = 10,000)`**. However, instead of setting a fair price (e.g., 1 hotKey = 1 Token), they initialize the pool with an extreme skew: 1 Token = 1e18 hotKey.

- To achieve this, the attacker calls the Uniswap V3 Factory to create the pool.

- They then call the pool's **`initialize()`** function with a **`sqrtPriceX96`** value corresponding to a 1e18:1 price ratio in favor of WETH.

4. Next, the attacker makes a purchase on the HotCurves contract, moving the internal market cap from 11.49 ETH to 11.51 ETH, surpassing the threshold and triggering **`finalize()`**.

5. Inside **`finalize()`**, the contract calls **`createAndInitializePoolIfNecessary()`**. This function checks if the pool already exists and is initialized. Since the attacker preemptively initialized it at an extreme price ratio, the function simply returns the existing pool without resetting the price.

Reference: Uniswap V3 PoolInitializer.sol

6. The contract then calls **`mintPosition()`** without slippage protection:

```
amount0Min = 0;
amount1Min = 0;
```

As a result, the contract blindly deposits all its **`hotKey`** and **`Token`** into the already-skewed pool, providing liquidity at the attacker's extreme rate of 1 Token per 1e18 hotKey.

7. The attacker can now execute their profit-taking sequence:

- First, they swap their **`Token`** holdings (obtained during step 4) for **`HotKey`** tokens in the newly created Uniswap V3 pool

- Due to the skewed price ratio, this swap extracts a disproportionate amount of **`HotKey`** tokens from the pool

- Finally, they swap the obtained **`HotKey`** tokens for **`WETH`** using the existing UniswapV2 and other UniswapV3 pools, resulting in a profitable ETH position

---

Recommendation

To mitigate this attack, the following measures should be implemented:

1. Hard cap the market cap on the bonding curve, refunding the exceeding part of the last purchase to the buyer. The last **`buyToken()`** transaction should satisfy **`currentMarketCap = finalMarketCap`** so that the amount of liquidity that will be migrated to UniswapV3 is known in advance.

2. Since the amount of liquidity to be migrated is known in advance, the initial price ratio is known too. The factory, when creating new **`Coin`** and **`HotCurves`** contracts, should also create the pool and initialize it at the final expected price ratio to prevent attackers from front-running the initialization.

3S-HotCurves-C02

Lack of slippage protection during ETH to Hotkey swap in pool finalization

Id	3S-HotCurves-C02
Classification	Critical
Severity	Critical
Likelihood	High
Category	Bug
Status	Addressed in #3bca66e.

Description

Upon reaching a certain market cap of the newly created token, finalization phase is triggered, during which a Uniswap V3 pool is created, and liquidity is deposited into it. During finalization, the available ETH is swapped into Hotkey tokens because the pair of the newly created pool is Token/Hotkey.

The problem with the implementation of this swap is that the parameter **`amountOutMinimum: 0`** is used, which means there is no slippage protection.

```
ISwapRouter.ExactInputSingleParams memory params =
ISwapRouter.ExactInputSingleParams({
    tokenIn: weth,
    tokenOut: hotKey,
    fee: 10000,
    recipient: address(this),
    amountIn: wethBalance, // All ETH balance
    amountOutMinimum: 0,
    sqrtPriceLimitX96: 0
});
```

This could lead to an unfavorable exchange rate and loss of funds. A malicious user could exploit this by executing a sandwich attack and extracting value from the process. An example scenario is as follows:

1) The malicious user can trigger the finalization process in two ways: by executing **`buyToken()`** with enough ETH to exceed the **`finalMarketCap`**, or by frontrunning a

transaction that would trigger finalization. They then perform a swap, purchasing Hotkey tokens for a certain amount of ETH from the Hotkey/ETH Uniswap pool.

2) The HotCurves contract swaps its ETH for Hotkey, which increases the price of Hotkey measured in ETH. Due to the purchase in step 1), this happens at a very unfavorable rate for the protocol.

3) The malicious user swaps the Hotkey tokens purchased in step 1 back to ETH, receiving more ETH than they initially spent.

Recommendation

The solution is to implement slippage protection using the **`amountOutMinimum`** parameter. This minimum output amount should be based on a TWAP price (e.g., 5-minute interval) retrieved from the Uniswap pool with a reasonable tolerance to protect against price manipulation and ensure a fair exchange rate.

3S-HotCurves-H01

Incorrect fee calculation and token accounting in sellToken function

Id	3S-HotCurves-H01
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #6368abb.

Description

The **`sellToken`** function in the **`HotCurves`** contract is designed to facilitate the sale of tokens by users, applying fees and distributing them to a referrer and a designated fee address. However, the current implementation contains a flaw in the fee calculation and token accounting logic. The function incorrectly computes the conversion from tokens to ETH multiple times: once for the referrer fee, once for the fee address, and once for the remaining token amount. This repeated conversion over the same portion of the curve leads to inflated ETH amounts being calculated and distributed, resulting in financial discrepancies. Additionally, the **`virtualTokenLp`** is only increased by the portion of tokens not related to the fees, which is incorrect. The entire **`_amount`** of tokens should be accounted for in the liquidity pool, as all tokens are effectively entering the pool. Similarly, **`virtualEthLp`** and **`realEthLp`** are only decreased by the eth amount unrelated to the fees, which is also incorrect.

Recommendation

To resolve these issues, the function should first compute the total ETH equivalent of the **`_amount`** using the **`ethAmount`** function, obtaining **`_ethAmount`**. The fees should then be calculated as a proportion of **`_ethAmount`**. The ETH portion related to the fees should be distributed to the referrer and the fee address, while the remaining ETH should be sent to the **`msg.sender`**. Furthermore, ensure that **`virtualTokenLp`** is increased by the full **`_amount`** to accurately reflect the total tokens entering the pool, and that both **`virtualEthLp`** and **`realEthLp`** are decreased by **`_ethAmount`**.

3S-HotCurves-H02

Lack of slippage protection in **`buyToken()`** and **`sellToken()`** leads to sandwich attacks and potential loss of funds

Id	3S-HotCurves-H02
Classification	High
Severity	High
Likelihood	High
Category	Bug
Status	Addressed in #0eded13.

Description

The HotCurves contract utilizes a bonding curve with the standard constant product formula **`(k = x * y)`** to determine token prices during buy and sell transactions. This means that when tokens are purchased, their price increases, and when they are sold, their price decreases.

The **`buyToken()`** and **`sellToken()`** functions lack slippage protection, making transactions vulnerable to sandwich attacks where a malicious user can manipulate the execution price, leading to unexpectedly high costs or loss of funds for regular users. The following is an example of how such an attack occurs:

1) A malicious user detects a buy transaction in the mempool. They frontrun this transaction by executing their own purchase first, artificially inflating the token price.

2) The regular user's transaction is then executed, but at a higher price, causing them to receive fewer tokens for their ETH than expected and potentially incurring significant financial loss.

3) The malicious user sells the tokens acquired in step 1 at the inflated price, securing a risk-free profit at the expense of the regular user, who has overpaid due to the manipulated price.

Such an attack can also be executed during token sales.

Recommendation

This issue can be mitigated by introducing slippage protection using a parameter that specifies the minimum number of tokens the user must receive for the transaction to succeed. This prevents users from executing trades at highly unfavorable rates and protects against unexpected fund losses due to manipulated prices.

3S-HotCurves-L01

Hardcoded pool parameters in **`swapEthToHotkey`** function may lead to suboptimal liquidity

Id	3S-HotCurves-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Addressed in #147cb9d.

Description

The **`swapEthToHotkey`** function is designed to convert the Ether (ETH) accumulated in the contract into the HotKey token. This conversion is performed using a Uniswap V4 pool. Within the function, the pool parameters such as **`fee`** and **`tickSpacing`** are hardcoded. Specifically, the **`fee`** is set to 10000 and **`tickSpacing`** to 200. These hardcoded values may not correspond to the most liquid pool available for ETH-HotKey swaps, potentially leading to suboptimal trading conditions and increased slippage.

Recommendation

To ensure optimal liquidity and minimize slippage during swaps, it is recommended to make the **`fee`** and **`tickSpacing`** parameters configurable by the contract owner. This flexibility will allow the contract to adapt to the most liquid pool available for ETH-HotKey swaps, thereby improving the efficiency of the conversion process. Consider adding setter functions that allow the owner to update these parameters as needed.

3S-HotCurves-L02

Inconsistent referrer fee accounting in HotCurves

Id	3S-HotCurves-L02
Classification	Low
Severity	Low
Likelihood	High
Category	Bug
Status	Addressed in #646b8d5.

Description

The functions **`buyToken()`** and **`sellToken()`** in the HotCurves contract apply a fee on each execution as a percentage of the transaction amount. It is possible to provide a referrer as a parameter, who receives half of these fees.

To track the total amount each referrer has received in fees and the overall amount distributed to referrers, the variables **`refAmounts`** and **`totalRefAmounts`** are used. Whenever a referrer is specified, the corresponding fee amount is added to these variables. These variables are shared between both **`buyToken()`** and **`sellToken()`** functions.

```
if (_ref != address(0) && _ref != msg.sender)
{
    uint256 referFee = feeAmount / 2;
    feeAmount -= referFee;
    payable(_ref).transfer(referFee); // @audit може да се използва за
атаката
    if(refAmounts[_ref] == 0){
        refCount ++;
    }
    refAmounts[_ref] += referFee;
        totalRefAmounts += referFee;
}
```

```
if(_ref != address(0) && _ref != msg.sender)
```

```
{
    uint256 referFee = feeAmount / 2;
    feeAmount -= referFee;
    IERC20(token).transfer(_ref, referFee);
    if(refAmounts[_ref] == 0){
        refCount ++;
    }
    refAmounts[_ref] += referFee;
    totalRefAmounts += referFee;
}
```

The issue arises because, in **`buyToken()`**, the fee is calculated based on ETH amount, meaning the fee itself is also in ETH, while in **`sellToken()`**, the fee is calculated based on the token amount, meaning the fee is measured in that token. This inconsistency results in incorrect values being stored in **`refAmounts`** and **`totalRefAmounts`**. These values are not currently used in any on-chain logic, but they are likely used for off-chain operations, making this a potential issue.

Recommendation

The simplest solution is to introduce a second set of variables to separately track referrer fees in ETH and in the Token. This would eliminate the need for conversion between different assets and ensure accurate accounting.

3S-HotCurves-N01

Potential Gas limit and RPC limit issues in array-reading functions

Id	3S-HotCurves-N01
Classification	None
Category	Suggestion
Status	Addressed in #c0c03d8.

Description

The **`Multicall`** contract and the **`HotCurvesFactory`** and **`HotCurves`** contracts contain several view functions that iterate over potentially long storage arrays. Specifically, the **`Multicall`** contract's view functions, **`HotCurvesFactory.getAllAddresses`**, and **`HotCurves.getAllPrices`** are designed to retrieve data from arrays that may grow significantly over time. As these arrays increase in size, the gas required to execute these functions may exceed the block gas limit, leading to failed transactions. Additionally, when these functions are called via RPC, they may hit RPC limits, resulting in incomplete data retrieval or errors.

Recommendation

To mitigate the risk of hitting gas or RPC limits, implement an optional pagination feature for these functions. This can be achieved by adding parameters to specify the start index and the number of entries to return.

3S-HotCurves-N02

Redundant `receive()` Function in HotCurves, HotCurvesFactory, and Coin Contracts

Id	3S-HotCurves-N02
Classification	None
Category	Suggestion
Status	Addressed in #0aec54d.

Description

The `receive()` function is implemented in the HotCurves, HotCurvesFactory, and Coin contracts. This function is automatically invoked when the contract receives Ether without any accompanying data. In the context of these contracts, the `receive()` function does not serve any specific purpose or contribute to the intended functionality of the contracts. Its presence can lead to unintended Ether transfers to the contract, which may result in the loss of funds due to user mistakes.

Recommendation

Remove the `receive()` function from the HotCurves, HotCurvesFactory, and Coin contracts to prevent accidental Ether transfers that do not serve any functional purpose.

3S-HotCurves-N03

`buyToken()` and `sellToken()` Functions Don't Follow CEI (Checks-Effects-Interactions) Pattern

Id	3S-HotCurves-N03
Classification	None
Category	Suggestion
Status	Addressed in #5494a42.

Description

The `buyToken()` and `sellToken()` functions in the `HotCurves` contract do not follow the CEI (Checks-Effects-Interactions) pattern, which is a crucial best practice in Solidity for preventing reentrancy attacks and other potential vulnerabilities.

While both functions use the `nonReentrant` modifier which provides some protection, not following CEI pattern doesn't adhere to best practices and could potentially introduce subtle edge cases or read-only reentrancy vulnerabilities.

Recommendation

Adopt the CEI pattern more rigorously:

Checks: Verify conditions and calculate amounts.

Effects: Update all relevant state variables (`virtualTokenLp`, `virtualEthLp`, `volume`, etc.).

Interactions: Transfer ETH and tokens to fee addresses or referrers only after updating contract state.

3S-HotCurves-N04

Market Cap Check Fails to Finalize Liquidity Migration at Exact Threshold

Id	3S-HotCurves-N04
Classification	None
Category	Suggestion
Status	Addressed in #7bed796.

Description

The contract's logic for triggering the **`finalize()`** function relies on the following condition:

```
if (finalMarketCap < currentMarketCap) {
    finalize();
}
```

However, this condition only triggers **`finalize()`** when **`currentMarketCap`** exceeds **`finalMarketCap`**. If the market cap reaches exactly **`finalMarketCap`**, the condition does not hold (**`finalMarketCap < currentMarketCap`** evaluates to **`false`**), and **`finalize()`** is not called.

Recommendation

To ensure **`finalize()`** is called as soon as **`currentMarketCap`** reaches or exceeds the limit, modify the condition to:

```
- if (finalMarketCap < currentMarketCap) {
+ if (finalMarketCap <= currentMarketCap) {
    finalize();
}
```

This guarantees that **`finalize()`** executes at the correct threshold and prevents further trading beyond the cap.

3S-HotCurves-N05

Sending tokens to **`address(0)`** does not reduce totalSupply

Id	3S-HotCurves-N05
Classification	None
Category	Suggestion
Status	Addressed in #b6ce9ce.

Description

HotCurves uses tokens implemented in the Coin contract, which inherits from ERC20. This contract implements the standard token functions such as **`mint()`**, **`transfer()`**, **`approve()`**, etc. However, there is no dedicated **`burn()`** function for permanently removing tokens from circulation.

In practice, tokens can still be "burned" by sending them to **`address(0)`**. In most ERC20 implementations, including OpenZeppelin's, when tokens are sent to **`address(0)`**, the **`totalSupply`** is reduced, as these tokens become irrecoverable.

However, in this ERC20 implementation, sending tokens to **`address(0)`** does not decrease **`totalSupply`**, which can lead to an incorrect market cap calculation since these tokens are effectively removed from circulation but still counted as part of the supply.

Recommendation

Modify the **`_transfer()`** function to decrease **`totalSupply`** when tokens are sent to **`address(0)`**, ensuring accurate market cap calculations.