BLOCKSEC

Security Audit Report for kensei contract v3

Date: September 26, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	kenseion katana
Target	kensei contract v3

Version History

Version	Date	Description
1.0	September 26, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-09-26 13:50:30+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of kensei contract v3 of kenseion katana.

The protocol is a token launchpad that uses a bonding curve for initial trading. It's a "pump.fun" style model that migrates tokens to Uniswap V3 when they hit a market cap milestone. The system uses a constant product curve and charges fees on trades. Once a token reaches a specific market cap, it automatically migrates, locking liquidity permanently to prevent rug pulls. The protocol features server-signed creation and immutable per-token parameters for security and fair launch.

Note this audit only focuses on the smart contracts in the following directories/files:

- src/

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
kensei contract v3	Version 1	f9eb6c6d1d22b0c741438dfe77c667cfd09e0579
	Version 2	d42e0d9bcb40e8f82c47e21a2244f95e55fa802f
	Version 3	88ee4bee4251093982e1985270e07cfd482872a1

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/kenseionkatana/kensei-contract-v3

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **twelve** potential security issues. Besides, we have **six** recommendations and **five** notes.

- High Risk: 4
- Medium Risk: 6
- Low Risk: 2
- Recommendation: 6
- Note: 5

ID	Severity	Description	Category	Status
1	High	Integer overflow vulnerability in price calculation	Security Issue	Fixed
2	High	Uniswap V3 pool initialization without depositing tokens allows price manipulation	Security Issue	Fixed
3	High	Lack of validations of the parameter `_token`	Security Issue	Fixed
4	High	Uninitialized liquidity allows attackers to manipulate `MemeToken` prices	Security Issue	Fixed
5	Medium	Unused meme tokens are not returned to the user	Security Issue	Fixed
6	Medium	Inaccurate calculation of returned amounts due to lack of fee deductions	Security Issue	Fixed
7	Medium	Market cap can be manipulated by burning tokens	Security Issue	Fixed
8	Medium	Inconsistent fee charging	Security Issue	Fixed
9	Medium	Incorrect fee configuration validations	Security Issue	Fixed
10	Medium	Potential DoS on migration due to donation attack	Security Issue	Fixed
11	Low	Trading can be stopped without a corresponding migration trigger	Security Issue	Fixed
12	Low	Incorrect rounding direction leads to underpayment	Security Issue	Fixed
13	-	Remove redundant code	Recommendation	Fixed
14	-	Apply CEI pattern in the function `buyExactOut()`	Recommendation	Fixed
15	-	Sweep and burn meme token after the initial liquidity addition	Recommendation	Fixed
16	-	Revise the variable name	Recommendation	Fixed
17	-	Revise code typos	Recommendation	Fixed
18	-	Use state-of-the-art implementation for function `_sqrt()`	Recommendation	Fixed

19	-	Custom constants `MIN_TICK` and `MAX_TICK`	Note	-
20	-	Ensure reasonable `MemeTokenV3` initialization parameters setting	Note	-
21	-	Meme tokens should have identical symbol and name	Note	-
22	-	Potential centralization risks	Note	-
23	-	Hardcoded value and contracts deployment chain	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Integer overflow vulnerability in price calculation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_getSqrtPriceX96()` is designed to compute the square root price for a token pair. The contract attempts to determine the price by calculating either `(amountETH << 192) / amountToken` or `(amountToken << 192) / amountETH`.

However, Solidity's left shift operator does not automatically handle overflow, which can cause the computed `priceX192` to be incorrect. Taking `(amountETH << 192) / amountToken` as an example, if `amountETH` is greater than 2^{64} (approximately `18.44 ether`), the left shift will overflow.

Therefore, the result of `_getSqrtPriceX96()` becomes incorrect, which in turn affects other computations that depend on it, potentially causing contract logic errors or loss of funds.

```
function _getSqrtPriceX96(
    uint256 amountToken,
    uint256 amountETH,
    bool token0IsThis
) internal pure returns (uint160) {
    // price = token1/token0
    // If token0 is this token, price = ETH / token
    // If token0 is WETH, price = token / ETH
    uint256 priceX192;
    if (token0IsThis) {
        // price = amountETH / amountToken
        priceX192 = (amountETH << 192) / amountToken;
    } else {
        // price = amountToken / amountETH
        priceX192 = (amountToken << 192) / amountETH;
    }
    return uint160(_sqrt(priceX192));
}
```

Listing 2.1: src/MemeTokenV3.sol

Impact This vulnerability could cause the result to overflow, disrupting core functionality and leading to potential loss of funds.

Suggestion Revise the logic accordingly. This link can be referred to.

2.1.2 Uniswap V3 pool initialization without depositing tokens allows price manipulation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MemeTokenV3`, the function `migrate()` creates a Uniswap V3 pool for the new token and `WETH`, with an initial price `sqrtPriceX96`.

However, if the pool already exists and already has a current price, the function `createAndInitializePoolIfNecessary()` in the contract `positionManager` will return the existing pool address and will not update the current price with the provided `sqrtPriceX96`.

Therefore, an attacker can front-run the migration by invoking the function `createAndInitializePoolIfNecessary()` in the contract `positionManager` directly, which allows a pool to be initialized without depositing any tokens. This enables the attacker to set an arbitrary price, which could allow them to purchase meme tokens at an artificially low price and then sell them at a manipulated higher price after the migration. This vulnerability could be exploited to manipulate the price of the newly created meme token, resulting in financial loss for honest users.

```
        uint160 sqrtPriceX96 = _getSqrtPriceX96(tokensToMigrate, collateralAmount, token0 ==
            address(this));
        address pool = positionManager.createAndInitializePoolIfNecessary(token0, token1, poolFee,
            sqrtPriceX96);
        pair = pool;
```

Listing 2.2: src/MemeTokenV3.sol

Impact This vulnerability could be exploited to manipulate the price of the newly created meme token, resulting in financial loss for honest users.

Suggestion Revise the logic accordingly.

2.1.3 Lack of validations of the parameter `_token`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MemeFactoryV3`, the swap functions (i.e., `buyExactOut()`, `buyExactIn()`, `sellExactIn()`, `sellExactOut()`) and the `migrate()` function process token operations without verifying if the input `_token` was legitimately created by this factory. However, the design is problematic due to its failure to validate token origins. Specifically, any contract implementing the `IMemeTokenV3` interface can interact with these functions regardless of its creation source. As a result, this design can cause:

1. Protocol revenue loss: Rogue tokens can bypass fee mechanisms through custom implementations, undermining the protocol's revenue model.
2. Off-Chain system disruption: Off-chain monitoring systems may misinterpret false events emitted by rogue tokens, causing operational issues.
3. Phishing attacks: Users may unknowingly interact with malicious tokens mistaken for legitimate ones, increasing the risk of phishing scams.

```
function buyExactOut(
    address _token,
    uint256 _tokenAmount,
    uint256 _maxCollateralAmount
) external payable nonReentrant {
    (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee) = IMemeTokenV3(_token).buyExactOut{
        value: msg.value
    }(_tokenAmount, _maxCollateralAmount);

    IMemeTokenV3(_token).transfer(msg.sender, _tokenAmount);

    uint256 refund = address(this).balance;
    if (refund > 0) {
        (bool sent, ) = msg.sender.call{value: refund}("");
        if (!sent) revert FailedToSendETH();
    }

    emit BuyExactOut(
        msg.sender,
        _token,
        _tokenAmount,
        MemeTokenV3(payable(_token)).totalSupply() - IMemeTokenV3(_token).balanceOf(address(_token)),
        collateralToPayWithFee,
        refund,
        helioFee,
        dexFee,
        IMemeTokenV3(_token).getCurveProgressBps()
    );

    if (MemeTokenV3(payable(_token)).tradingStopped()) {
        readyForMigration[_token] = true;
        emit MarketcapReached(_token);
    }
}
```

Listing 2.3: src/MemeFactoryV3.sol

```
function buyExactIn(address _token, uint256 _amountOutMin) external payable nonReentrant {
```

```
        (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee) = IMemeTokenV3(_token).buyExactIn{
            value: msg.value
        }(_amountOutMin);

        uint256 tokensOut = IMemeTokenV3(_token).balanceOf(address(this));
        IMemeTokenV3(_token).transfer(msg.sender, tokensOut);

        uint256 refund = address(this).balance;
        if (refund > 0) {
            (bool sent, ) = msg.sender.call{value: refund}("");
            if (!sent) revert FailedToSendETH();
        }

        emit BuyExactIn(
            msg.sender,
            _token,
            tokensOut,
            MemeTokenV3(payable(_token)).totalSupply() - IMemeTokenV3(_token).balanceOf(address(_token)),
            collateralToPayWithFee,
            helioFee,
            dexFee,
            IMemeTokenV3(_token).getCurveProgressBps()
        );

        if (MemeTokenV3(payable(_token)).tradingStopped()) {
            readyForMigration[_token] = true;
            emit MarketcapReached(_token);
        }
    }
```

Listing 2.4: src/MemeFactoryV3.sol

```
    function sellExactIn(address _token, uint256 _tokenAmount, uint256 _amountCollateralMin) external nonReentrant {
        MemeTokenV3(payable(_token)).transferFrom(msg.sender, address(this), _tokenAmount);
        (uint256 collateralToReceiveMinusFee, uint256 helioFee, uint256 dexFee) = MemeTokenV3(payable(_token)).sellExactIn(
            _tokenAmount,
            _amountCollateralMin
        );

        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        if (!sent) revert FailedToSendETH();

        emit SellExactIn(
            msg.sender,
            _token,
            _tokenAmount,
            MemeTokenV3(payable(_token)).totalSupply() - MemeTokenV3(payable(_token)).balanceOf(address(_token)),
            collateralToReceiveMinusFee,
```

```
            helioFee,
            dexFee,
            IMemeTokenV3(_token).getCurveProgressBps()
        );
    }
```

Listing 2.5: src/MemeFactoryV3.sol

```
    function sellExactOut(address _token, uint256 _tokenAmountMax, uint256 _amountCollateral)
         external nonReentrant {
        MemeTokenV3(payable(_token)).transferFrom(msg.sender, address(this), _tokenAmountMax);
        (uint256 collateralToReceiveMinusFee, uint256 tokensOut, uint256 helioFee, uint256 dexFee)
             = MemeTokenV3(payable(
            _token
        )).sellExactOut(_tokenAmountMax, _amountCollateral);

        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        if (!sent) revert FailedToSendETH();

        emit SellExactOut(
            msg.sender,
            _token,
            tokensOut,
            MemeTokenV3(payable(_token)).totalSupply() - MemeTokenV3(payable(_token)).balanceOf(
                address(_token)),
            collateralToReceiveMinusFee,
            helioFee,
            dexFee,
            IMemeTokenV3(_token).getCurveProgressBps()
        );
    }
```

Listing 2.6: src/MemeFactoryV3.sol

```
    function migrate(address _token) external {
        if (!readyForMigration[_token]) revert NotReadyForMigration();

        (uint256 tokensToMigrate, uint256 tokensToBurn, uint256 collateralAmount) = MemeTokenV3(
             payable(_token)).migrate();
        emit Migrated(
            _token,
            tokensToMigrate,
            tokensToBurn,
            collateralAmount,
            MemeTokenV3(payable(_token)).fixedMigrationFee() + MemeTokenV3(payable(_token)).
                poolCreationFee(),
            MemeTokenV3(payable(_token)).pair()
        );
    }
```

Listing 2.7: src/MemeFactoryV3.sol

```
    function collectLpFees(address _token, uint128 amount0Max, uint128 amount1Max)
```

```
        external
        onlyOwner
        returns (uint256 amount0, uint256 amount1)
    {
        return IMemeTokenV3(_token).collectLpFees(amount0Max, amount1Max);
    }
```

Listing 2.8: src/MemeFactoryV3.sol

Impact 1. Protocol revenue loss: Rogue tokens can bypass fee mechanisms through custom implementations, undermining the protocol's revenue model. 2. Off-Chain system disruption: Off-chain monitoring systems may misinterpret false events emitted by rogue tokens, causing operational issues. 3. Phishing attacks: Users may unknowingly interact with malicious tokens mistaken for legitimate ones, increasing the risk of phishing scams.

Suggestion Implement comprehensive token validations.

2.1.4 Uninitialized liquidity allows attackers to manipulate `MemeToken` prices

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `MemeTokenV3`, the `constructor()` function initializes the price when creating the Uniswap V3 pool but does not provide any liquidity. As a result, attackers can freely manipulate the pool price. During migration, the contract may deposit `MemeToken` into the pool at an attacker-controlled price, potentially causing abnormal MemeToken pricing.

```
        uint256 vt0 = virtualTokenReserves;
        uint256 vc0 = virtualCollateralReserves;
        uint256 migrationThreshold = _params.tokensMigrationThreshold;
        (uint256 vtAtMcLower, uint256 vcAtMcLower) = _reservesAfterTokensSold(vt0, vc0,
            migrationThreshold);

        uint160 sqrtPriceX96Init = _getSqrtPriceX96(vtAtMcLower, vcAtMcLower, token0 == address(
            this));
        pair = _initOrValidatePool(token0, token1, sqrtPriceX96Init, true);
```

Listing 2.9: src/MemeTokenV3.sol

Impact This may cause the MemeToken price in the Uniswap V3 pool to be manipulated.

Suggestion Revise the logic accordingly.

2.1.5 Unused meme tokens are not returned to the user

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `sellExactOut()` in the contract `MemeFactoryV3` facilitates a meme token sale where a user specifies the exact amount of collateral they want to receive. The user must first transfer a maximum meme token amount to the factory contract before the sale is executed. However, the factory contract does not return any unused meme tokens to the user, which results in the user losing meme tokens that were transferred beyond what was needed for the trade.

```
function sellExactOut(address _token, uint256 _tokenAmountMax, uint256 _amountCollateral) external nonReentrant {
    MemeTokenV3(payable(_token)).transferFrom(msg.sender, address(this), _tokenAmountMax);
    (uint256 collateralToReceiveMinusFee, uint256 tokensOut, uint256 helioFee, uint256 dexFee) = MemeTokenV3(payable(
        _token
    )).sellExactOut(_tokenAmountMax, _amountCollateral);

    (bool sent, ) = msg.sender.call{value: address(this).balance}("");
    if (!sent) revert FailedToSendETH();

    emit SellExactOut(
        msg.sender,
        _token,
        tokensOut,
        MemeTokenV3(payable(_token)).totalSupply() - MemeTokenV3(payable(_token)).balanceOf(address(_token)),
        collateralToReceiveMinusFee,
        helioFee,
        dexFee,
        IMemeTokenV3(_token).getCurveProgressBps()
    );
}
```

Listing 2.10: src/MemeFactoryV3.sol

```
function sellExactOut(
    uint256 _tokenAmountMax,
    uint256 _amountCollateral
)
    external
    payable
    onlyFactory
    sellChecks
    returns (uint256 collateralToReceiveMinusFee, uint256 tokensOut, uint256 helioFee, uint256 dexFee)
{
    (helioFee, dexFee) = _calculateFee(_amountCollateral);
    collateralToReceiveMinusFee = _amountCollateral - helioFee - dexFee;

    _transferCollateral(treasury, helioFee);
    _transferCollateral(dexTreasury, dexFee);

    tokensOut = (_amountCollateral * virtualTokenReserves) / (virtualCollateralReserves - _amountCollateral);

    if (tokensOut > _tokenAmountMax) revert SlippageCheckFailed();
    _transfer(msg.sender, address(this), tokensOut);
```

```

    virtualTokenReserves += tokensOut;
    virtualCollateralReserves -= _amountCollateral;

    _transferCollateral(msg.sender, collateralToReceiveMinusFee);
}
```

Listing 2.11: src/MemeTokenV3.sol

Impact This will result in the user losing meme tokens that were transferred beyond what was needed for the trade.

Suggestion Add the refund logic of meme tokens.

2.1.6 Inaccurate calculation of returned amounts due to lack of fee deductions

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `getAmountOutAndFee()` calculates the `amountOut` incorrectly. When `_paymentTokenIsIn` is true, the `_amountIn` used in the calculation doesn't have the fee deducted, which causes `amountOut` to be larger than it should be. When `_paymentTokenIsIn` is false, the returned `amountOut` also doesn't have the fee deducted, which similarly causes `amountOut` to be larger than it should be. The function `getAmountInAndFee()` has a similar issue.

```
function getAmountOutAndFee(
    uint256 _amountIn,
    uint256 _reserveIn,
    uint256 _reserveOut,
    bool _paymentTokenIsIn
) external view returns (uint256 amountOut, uint256 fee) {
    if (_paymentTokenIsIn) {
        (uint256 helioFee, uint256 dexFee) = _calculateFee(_amountIn);
        fee = helioFee + dexFee;

        amountOut = (_amountIn * _reserveOut) / (_reserveIn + _amountIn);
    } else {
        amountOut = (_amountIn * _reserveOut) / (_reserveIn + _amountIn);

        (uint256 helioFee, uint256 dexFee) = _calculateFee(amountOut);
        fee = helioFee + dexFee;
    }
}

function getAmountInAndFee(
    uint256 _amountOut,
    uint256 _reserveIn,
    uint256 _reserveOut,
    bool _paymentTokenIsOut
) external view returns (uint256 amountIn, uint256 fee) {
    if (_paymentTokenIsOut) {
```

```
        (uint256 helioFee, uint256 dexFee) = _calculateFee(_amountOut);
        fee = helioFee + dexFee;

        amountIn = (_amountOut * _reserveIn) / (_reserveOut - _amountOut);
    } else {
        amountIn = (_amountOut * _reserveIn) / (_reserveOut - _amountOut);
        (uint256 helioFee, uint256 dexFee) = _calculateFee(amountIn);

        fee = helioFee + dexFee;
    }
}
```

Listing 2.12: src/MemeTokenV3.sol

Impact This results in the returned `amountOut` and `amountIn` values being incorrect, which could lead to discrepancies between what a user expects and what the contract's internal calculations predict.

Suggestion Revise the logic accordingly.

2.1.7 Market cap can be manipulated by burning tokens

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `getMarketCap()` is designed to calculate a token's market capitalization, which is then used to determine migration eligibility and trading limits. The function uses `totalSupply()` to calculate the market cap, which is a value that can be decreased by burning tokens. An attacker can burn a small number of tokens they own, which manipulates the `totalSupply()` and further manipulates the market cap. This could allow attackers to bypass intended trading restrictions and limits, or it could prevent the token from ever migrating, causing a loss to other users.

```
function getMarketCap() public view returns (uint256) {
    uint256 mc = (virtualCollateralReserves * 10 ** 18 * totalSupply()) / virtualTokenReserves;
    return mc / 10 ** 18;
}
```

Listing 2.13: src/MemeTokenV3.sol

Impact This could allow attackers to bypass intended trading restrictions and limits, or it could prevent the token from ever migrating, causing a loss to other users.

Suggestion Revise the logic accordingly.

2.1.8 Inconsistent fee charging

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MemeTokenV3`, the function `buyExactOut()` calculates the fee based on the amount of `Ether` required to purchase the specified number of tokens, whereas the function `buyExactIn()` calculates the fee based on the total amount of `Ether` actually provided by the user. Since both functions apply the fee calculation logic, when purchasing the same number of `MemeToken` under the same `Ether`/`MemeToken` reserves, the amount of `Ether` a user pays through the two functions differs, resulting in inconsistent trading outcomes.

For example, assuming that purchasing `10 MemeToken` currently requires `1 Ether` and the fee rate is $\frac{1}{6}$, a user buying `10 MemeToken` through the function `buyExactOut()` would need to pay a total of $1 + \frac{1}{6} \approx 1.17$ `Ether`. However, when purchasing `10 MemeToken` through the function `buyExactIn()`, the user would need to pay `1.2 Ether`.

```
function buyExactOut(
    uint256 _tokenAmount,
    uint256 _maxCollateralAmount
)
    external
    payable
    onlyFactory
    buyChecks
    returns (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee)
{
    if (balanceOf(address(this)) <= _tokenAmount) revert InsufficientTokenReserves();

    uint256 collateralToSpend = (_tokenAmount * virtualCollateralReserves) / (
        virtualTokenReserves - _tokenAmount);
    (helioFee, dexFee) = _calculateFee(collateralToSpend);

    collateralToPayWithFee = collateralToSpend + helioFee + dexFee;

    if (collateralToPayWithFee > _maxCollateralAmount) revert SlippageCheckFailed();
    _transferCollateral(treasury, helioFee);
    _transferCollateral(dexTreasury, dexFee);

    virtualTokenReserves -= _tokenAmount;
    virtualCollateralReserves += collateralToSpend;

    uint256 refund;
    if (msg.value > collateralToPayWithFee) {
        refund = msg.value - collateralToPayWithFee;
        _transferCollateral(msg.sender, refund);
    } else if (msg.value < collateralToPayWithFee) {
        revert NotEnoughtETHToBuyTokens();
    }

    _transfer(address(this), msg.sender, _tokenAmount);
}
```

Listing 2.14: src/MemeTokenV3.sol

Impact Under the same conditions, users will receive inconsistent token amounts when purchasing through different functions.

Suggestion The two functions should use different fee calculation logic.

2.1.9 Incorrect fee configuration validations

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The fee validations in the function `_setConfig()` use existing state variables (i.e., `feeBasisPoints`, `dexFeeBasisPoints`) instead of the new parameters (i.e., `_feeBasisPoints`, `_dexFeeBasisPoints`), which is incorrect.

Specifically, the following two issues can be raised.

1. Invalid fee settings ($\geq$100% or $\geq$25%) may be accepted if the current state is valid, violating protocol invariants.

2. Valid updates may blocked if the current state is invalid. For example, setting 110% fees (invalid) is accepted when current fees are 10% (valid), while updating from 30% to 20% fees (valid) is blocked if current fees are above the 25% limit.

```
        if (dexFeeBasisPoints >= 10_000) revert FeeBPSCheckFailed();
        if (feeBasisPoints >= MAX_BPS) revert FeeBPSCheckFailed();
```

Listing 2.15: src/MemeFactoryV3.sol

Impact The invalid fee configurations can prevent legitimate parameter updates.

Suggestion Revise the logic accordingly.

2.1.10 Potential DoS on migration due to donation attack

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_tokensToMigrate()` calculates the amount of `MemeToken` based on `address(this).balance`. However, an attacker could artificially inflate this value through a donation attack, causing `tokensToMigrate` to exceed `tokensRemaining`, which could lead to DoS in the function `migrate()`.

```
    function _tokensToMigrate() internal view returns (uint256) {
        uint256 collateralDeductedFee = address(this).balance - fixedMigrationFee - poolCreationFee;
        return (virtualTokenReserves * collateralDeductedFee) / virtualCollateralReserves;
    }
```

Listing 2.16: src/MemeTokenV3.sol

```
        tokensToMigrate = _tokensToMigrate();
        tokensToBurn = tokensRemaining - tokensToMigrate;
```

Listing 2.17: src/MemeTokenV3.sol

Impact Potential DoS on migration via donation attack.

Suggestion Revise the function logic.

2.1.11 Trading can be stopped without a corresponding migration trigger

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `buyExactIn()` checks if a token's trading should be stopped after a buy operation is completed. If the `tradingStopped()` check is true, the `readyForMigration` flag is set to true. However, this logic is missing from the function `createMemeTokenAndBuy()`, which also allows users to buy tokens. This inconsistency could lead to a DoS, as a token's market cap could exceed the threshold and halt trading without the `readyForMigration` flag being set, preventing the token from being migrated to Uniswap V3.

```
function createMemeTokenAndBuy(
    string memory _name,
    string memory _symbol,
    uint256 _nonce,
    uint256 _tokenAmountMin,
    bytes memory _signature
) external payable nonReentrant returns (address) {
    _checkSignatureAndStore(_name, _symbol, _nonce, _signature);

    MemeTokenV3 token = new MemeTokenV3(
        IMemeTokenV3.ConstructorParams(
            _name,
            _symbol,
            msg.sender,
            totalSupply,
            virtualTokenReserves,
            virtualCollateralReserves,
            feeBasisPoints,
            dexFeeBasisPoints,
            migrationFeeFixed,
            poolCreationFee,
            mcLowerLimit,
            mcUpperLimit,
            tokensMigrationThreshold,
            treasury,
            NONFUNGIBLE_POSITION_MANAGER,
            dexTreasury,
            POOL_FEE,
            lpFeeReceiver
        )
    );

    (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee) = token.buyExactIn{value
        : msg.value}(
        _tokenAmountMin
```

```
    );

    uint256 tokenAmount = token.balanceOf(address(this));
    token.transfer(msg.sender, tokenAmount);

    moonshotTokens.push(address(token));
    emit NewMemeTokenAndBuy(
        address(token),
        msg.sender,
        _signature,
        tokenAmount,
        collateralToPayWithFee,
        helioFee,
        dexFee,
        token.getCurveProgressBps()
    );
    return address(token);
}
```

Listing 2.18: src/MemeFactoryV3.sol

```
function buyExactIn(address _token, uint256 _amountOutMin) external payable nonReentrant {
    (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee) = IMemeTokenV3(_token).
        buyExactIn{
        value: msg.value
    }(_amountOutMin);

    uint256 tokensOut = IMemeTokenV3(_token).balanceOf(address(this));
    IMemeTokenV3(_token).transfer(msg.sender, tokensOut);

    uint256 refund = address(this).balance;
    if (refund > 0) {
        (bool sent, ) = msg.sender.call{value: refund}("");
        if (!sent) revert FailedToSendETH();
    }

    emit BuyExactIn(
        msg.sender,
        _token,
        tokensOut,
        MemeTokenV3(payable(_token)).totalSupply() - IMemeTokenV3(_token).balanceOf(address(
            _token)),
        collateralToPayWithFee,
        helioFee,
        dexFee,
        IMemeTokenV3(_token).getCurveProgressBps()
    );

    if (MemeTokenV3(payable(_token)).tradingStopped()) {
        readyForMigration[_token] = true;
        emit MarketcapReached(_token);
    }
}
```

Listing 2.19: src/MemeFactoryV3.sol

Impact This inconsistency could lead to a DoS, as a token's market cap could exceed the threshold and halt trading without the `readyForMigration` flag being set, preventing the token from being migrated to Uniswap V3.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The following code is recommended to be removed to improve readability and save gas.

1. The function `buyExactIn()` in the contract `MemeFactoryV3` does not need to refund `Ether`.

```
        uint256 refund = address(this).balance;
        if (refund > 0) {
            (bool sent, ) = msg.sender.call{value: refund}("");
            if (!sent) revert FailedToSendETH();
        }
```

Listing 2.20: src/MemeFactoryV3.sol

Suggestion Remove the redundant code.

2.2.2 Apply CEI pattern in the function `buyExactOut()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `buyExactOut()` in the contract `MemeFactoryV3` updates the state after refunding `Ether` to users, which violates the Checks-Effects-Interactions(CEI) pattern. It is recommended to apply such a pattern to follow the development best practice.

```
        uint256 refund = address(this).balance;
        if (refund > 0) {
            (bool sent, ) = msg.sender.call{value: refund}("");
            if (!sent) revert FailedToSendETH();
        }

        emit BuyExactOut(
            msg.sender,
            _token,
            _tokenAmount,
            MemeTokenV3(payable(_token)).totalSupply() - IMemeTokenV3(_token).balanceOf(address(
                _token)),
```

```
            collateralToPayWithFee,
            refund,
            helioFee,
            dexFee,
            IMemeTokenV3(_token).getCurveProgressBps()
        );

        if (MemeTokenV3(payable(_token)).tradingStopped()) {
            readyForMigration[_token] = true;
            emit MarketcapReached(_token);
        }
```

Listing 2.21: src/MemeFactoryV3.sol

Suggestion Apply CEI pattern in the function `buyExactOut()`

2.2.3 Sweep and burn meme token after the initial liquidity addition

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `MemeTokenV3`, the function `migrate()` only invokes `positionManager.refundETH()` but does not invoke function `sweepToken`() to collect residual tokens. it is recommended to add a call to function `positionManager.sweepToken()` and subsequently burn the collected tokens.

```
        positionManager.refundETH();
```

Listing 2.22: src/MemeTokenV3.sol

Suggestion Add a call to `positionManager.sweepToken()` and subsequently burn the collected tokens after the initial liquidity addition.

2.2.4 Revise the variable name

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The constant value `MAX_BPS` in the contract `MemeTokenV3` should be revised. To our knowledge, the `MAX_BPS` should refer to the maximum configurable fee rate, instead of a basis point. It is recommended to revise it to `MULTIPLIER` for example.

```
    uint256 public constant MAX_BPS = 10_000;
```

Listing 2.23: src/MemeTokenV3.sol

```
    function _calculateFee(uint256 _amount) internal view returns (uint256 treasuryFee, uint256
         dexFee) {
        treasuryFee = (_amount * feeBPS) / MAX_BPS;
        dexFee = (treasuryFee * dexFeeBPS) / MAX_BPS;
        treasuryFee -= dexFee;
    }
```

Listing 2.24: src/MemeTokenV3.sol

Suggestion Revise the variable naming.

2.2.5 Revise code typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several code typos, which affect the readability of the code and is recommended to revise them. Specifically:

1. The variable name `initalTokenSupply` is misspelled.

```
uint256 public initalTokenSupply;
```

Listing 2.25: src/MemeTokenV3.sol

```
    initalTokenSupply = _params.totalSupply;
```

Listing 2.26: src/MemeTokenV3.sol

```
    uint256 progress = ((initalTokenSupply - balanceOf(address(this))) * MAX_BPS) /
        tokensMigrationThreshold;
```

Listing 2.27: src/MemeTokenV3.sol

2. The revert errors are misordered.

```
    if (_mcLowerLimit == 0) revert McUpperLimitZeroValue();
    if (_mcUpperLimit == 0) revert McLowerLimitZeroValue();
```

Listing 2.28: src/MemeFactoryV3.sol

Suggestion Revise the code typos.

2.2.6 Use state-of-the-art implementation for function `_sqrt()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_sqrt()` in `MemeTokenV3` currently implements the square root using the Newton-Raphson method. However, the current implementation's gas usage is not well controlled. It is recommended to use the state-of-the-art one (i.e., OpenZeppelin implementation link)

```
function _sqrt(uint256 x) internal 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;
```

```
        }
    }
```

Listing 2.29: src/MemeTokenV3.sol

Suggestion It is recommended to use the state-of-the-art implementation (e.g., OpenZeppelin implementation link)

2.3 Note

2.3.1 Custom constants `MIN_TICK` and `MAX_TICK`

Introduced by `Version 1`

Description The constants `MIN_TICK` and `MAX_TICK` defined in the contract `MemeTokenV3` are set to be -887220 and 887220 respectively, which correspond to the maximum range for the 0.3% fee tier rather than the Uniswap's native supported range of [-887272, 887272].

```
int24 internal constant MIN_TICK = -887220;
int24 internal constant MAX_TICK = 887220;
```

Listing 2.30: src/MemeTokenV3.sol

Feedback from the project The project clarifies that the `POOL_FEE` is fixed at 3000 (i.e., fee tier 0.3%).

2.3.2 Ensure reasonable `MemeTokenV3` initialization parameters setting

Introduced by `Version 1`

Description The protocol must ensure reasonable initialization parameters(e.g., `initalTokenSupply`, `tokensMigrationThreshold`, `virtualCollateralReserves`, `virtualTokenReserves`, `mcLowerLimit`) configured through the contract `MemeFactoryV3` for the contract `MemeTokenV3`.

Specifically, insufficient parameter validation may result in a discrepancy between the current price derived from the contract's constant product curve during migration and the initial Uniswap V3 pool price. This leads to two issues:

1. To meet the specified `tokensMigrationThreshold`, later users must purchase Meme Token at a higher price than the initial Uniswap V3 pool price. Given market dynamics, this discourages purchases, hindering Meme Token migration.

2. At migration, if the constant product curve's current price is lower than the initial Uniswap V3 pool price, it creates a potential risk-free arbitrage opportunity for the last user purchasing Meme Tokens, which is unfair to other users.

```
        MemeTokenV3 token = new MemeTokenV3(
            IMemeTokenV3.ConstructorParams(
                _name,
                _symbol,
                msg.sender,
                totalSupply,
                virtualTokenReserves,
```

```
            virtualCollateralReserves,
            feeBasisPoints,
            dexFeeBasisPoints,
            migrationFeeFixed,
            poolCreationFee,
            mcLowerLimit,
            mcUpperLimit,
            tokensMigrationThreshold,
            treasury,
            NONFUNGIBLE_POSITION_MANAGER,
            dexTreasury,
            POOL_FEE,
            lpFeeReceiver
        )
    );
```

Listing 2.31: src/MemeFactoryV3.sol

```
    initalTokenSupply = _params.totalSupply;
    virtualCollateralReserves = _params.virtualCollateralReserves;
    virtualCollateralReservesInitial = _params.virtualCollateralReserves;
    virtualTokenReserves = _params.virtualTokenReserves;
```

Listing 2.32: src/MemeTokenV3.sol

```
    mcLowerLimit = _params.mcLowerLimit;
    mcUpperLimit = _params.mcUpperLimit;
```

Listing 2.33: src/MemeTokenV3.sol

2.3.3 Meme tokens should have identical symbol and name

Introduced by `Version 1`

Description Since the protocol does not enforce strict validation for the uniqueness of token symbols and names, multiple tokens with the same symbol or name can be created by invoking the function `createMemeToken()` or `createMemeTokenAndBuy()`. As it can lead to user confusion, potential phishing risks and incorrect token interactions, the `signer` should ensure that the meme tokens have identical symbol and name.

```
function createMemeToken(
    string memory _name,
    string memory _symbol,
    uint256 _nonce,
    bytes memory _signature
) external returns (address) {
    _checkSignatureAndStore(_name, _symbol, _nonce, _signature);
    MemeTokenV3 token = new MemeTokenV3(
        IMemeTokenV3.ConstructorParams(
            _name,
            _symbol,
            msg.sender,
            totalSupply,
```

```
            virtualTokenReserves,
            virtualCollateralReserves,
            feeBasisPoints,
            dexFeeBasisPoints,
            migrationFeeFixed,
            poolCreationFee,
            mcLowerLimit,
            mcUpperLimit,
            tokensMigrationThreshold,
            treasury,
            NONFUNGIBLE_POSITION_MANAGER,
            dexTreasury,
            POOL_FEE,
            lpFeeReceiver
        )
    );

    moonshotTokens.push(address(token));
    emit NewMemeToken(address(token), msg.sender, _signature);
    return address(token);
}
```

Listing 2.34: src/MemeFactoryV3.sol

```
function createMemeTokenAndBuy(
    string memory _name,
    string memory _symbol,
    uint256 _nonce,
    uint256 _tokenAmountMin,
    bytes memory _signature
) external payable nonReentrant returns (address) {
    _checkSignatureAndStore(_name, _symbol, _nonce, _signature);

    MemeTokenV3 token = new MemeTokenV3(
        IMemeTokenV3.ConstructorParams(
            _name,
            _symbol,
            msg.sender,
            totalSupply,
            virtualTokenReserves,
            virtualCollateralReserves,
            feeBasisPoints,
            dexFeeBasisPoints,
            migrationFeeFixed,
            poolCreationFee,
            mcLowerLimit,
            mcUpperLimit,
            tokensMigrationThreshold,
            treasury,
            NONFUNGIBLE_POSITION_MANAGER,
            dexTreasury,
            POOL_FEE,
            lpFeeReceiver
```

```
        )
    );

    (uint256 collateralToPayWithFee, uint256 helioFee, uint256 dexFee) = token.buyExactIn{value
        : msg.value}(
        _tokenAmountMin
    );

    uint256 tokenAmount = token.balanceOf(address(this));
    token.transfer(msg.sender, tokenAmount);

    moonshotTokens.push(address(token));
    emit NewMemeTokenAndBuy(
        address(token),
        msg.sender,
        _signature,
        tokenAmount,
        collateralToPayWithFee,
        helioFee,
        dexFee,
        token.getCurveProgressBps()
    );
    return address(token);
}
```

Listing 2.35: src/MemeFactoryV3.sol

2.3.4 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., owner, signer) can conduct sensitive operations, which introduces potential centralization risks. For example, the owner can change the protocol's configuration by invoking the function setConfig(), which may block the meme tokens launching process. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.5 Hardcoded value and contracts deployment chain

Introduced by Version 1

Description In this project, a hardcoded value UNISWAP_V3_POOL_INIT_CODE_HASH is used in the contract MemeTokenV3. The project must ensure this hardcoded value is correct and all contracts are deployed on the proper chain for use.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS