Security Review Report
NM-0639 - Kensei

NETHERMIND
SECURITY

(October 8, 2025)

Contents

1 Executive Summary

This document presents the results of a security review conducted by Nethermind Security for Kensei contracts. The protocol enables users to deploy tokens which can initially be bought and sold on a local price curve managed by the token itself. After enough demand exists and the internal price has been pushed to a specified threshold, the token will "migrate" with all remaining liquidity being moved to a Sushiswap V3 pool. Once the migration has completed to the Sushiswap pool, all further trading can only be done through the pool rather than through the token itself.

The audit comprises 1024 lines of Solidity code. **The audit was performed using** (a) manual analysis of the codebase, and (b) automated analysis tools, and (c) simulation and creation of test cases.

Along this document, we report seven points of attention, where two are classified as `Medium` and five are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tools used. Section 9 concludes the document.

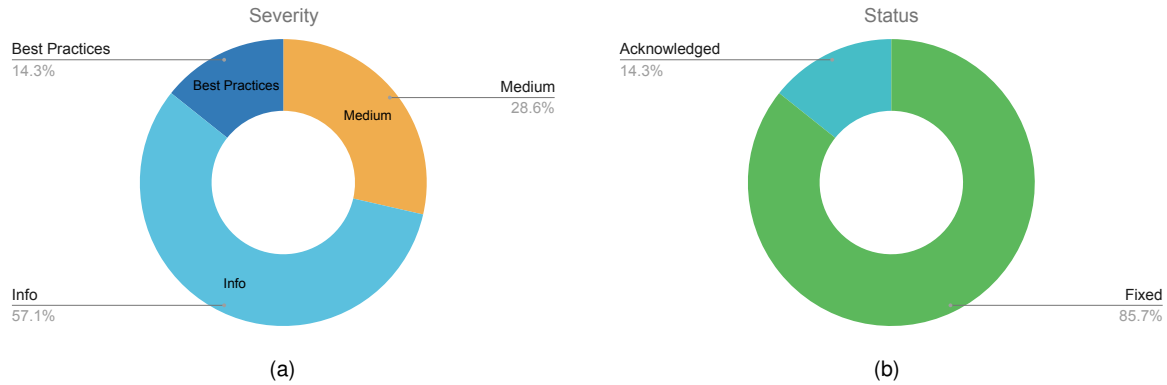


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (0), **Undetermined** (0), **Informational** (4), **Best Practices** (1). **Distribution of status: Fixed** (6), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	October 3, 2025
Final Report	October 8, 2025
Initial Commit	6cbfd7cc72007fd48d058ff749557f6fab4a78f9
Final Commit	9e78fddda94406aa7387cb9e1081309a427c6c2f
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/MemeFactoryV3.sol	353	5	1.4%	51	409
2	src/MemeTokenV3.sol	410	48	11.7%	102	560
3	src/interfaces/IUniswapV3Factory.sol	6	1	16.7%	4	11
4	src/interfaces/IMemeFactory.sol	127	1	0.8%	16	144
5	src/interfaces/IMemeTokenV3.sol	59	2	3.4%	12	73
6	src/interfaces/IUniswapV3Pool.sol	23	1	4.3%	5	29
7	src/interfaces/INonfungiblePositionManager.sol	46	17	37.0%	10	73
	Total	**1024**	**75**	**7.3%**	**200**	**1299**

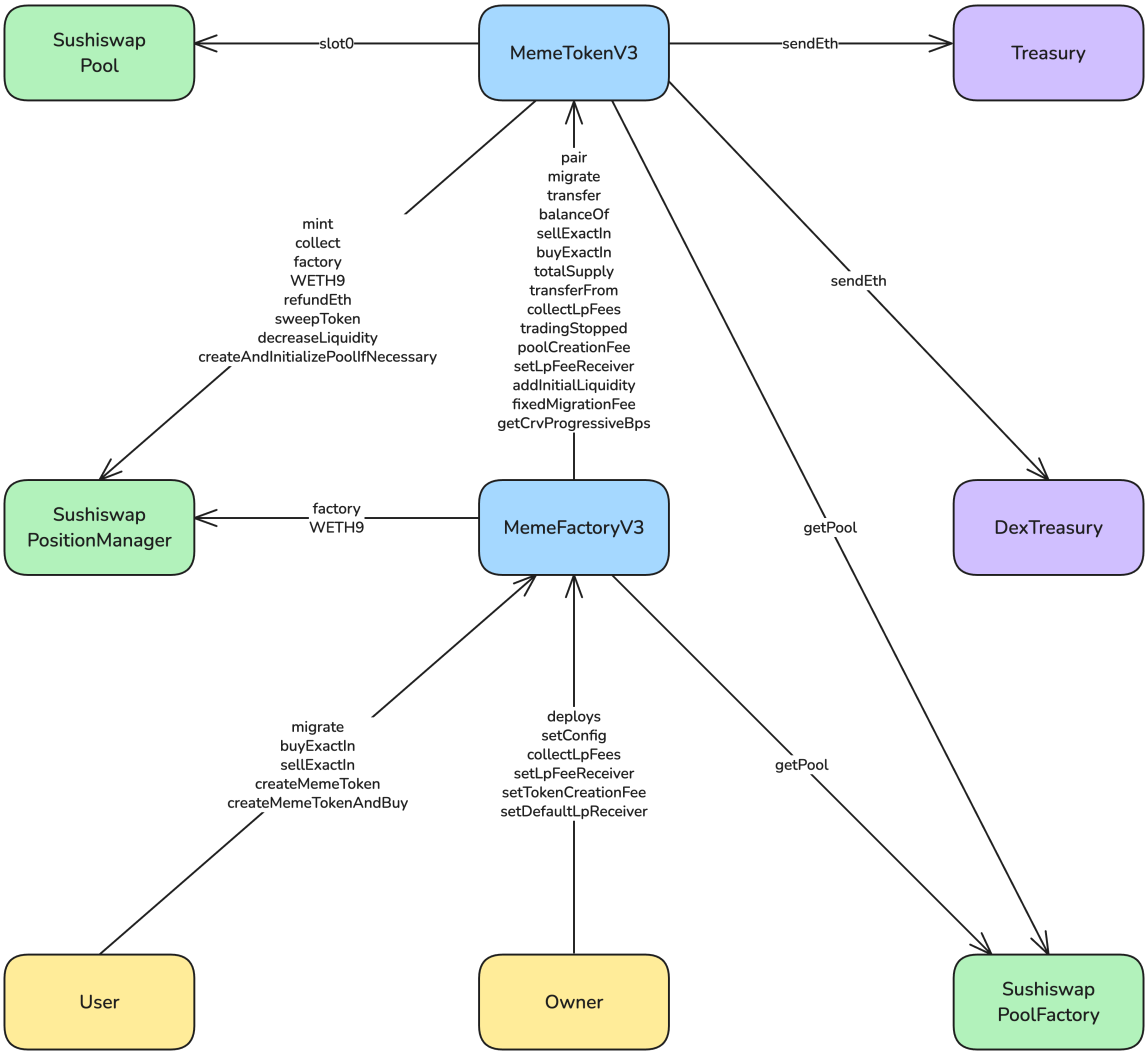
3 Summary of Issues

	Finding	Severity	Update
1	Ambiguity in `abi.encodePacked` allows creating tokens with unauthorized names and symbols	Medium	Fixed
2	Token creations are susceptible to denial-of-service	Medium	Acknowledged
3	Precision loss in fee calculation leaves dust ETH in the contract	Info	Fixed
4	Storage variable `initialTokenSupply` can be immutable	Info	Fixed
5	Token approval during migrate can use internal `_approve`	Info	Fixed
6	Token constructor has unnecessary storage loads	Info	Fixed
7	Function `sellExactIn` is marked payable but shouldn't accept native assets	Best Practices	Fixed

4 Protocol Overview

The Kensei protocol is a decentralized token launchpad designed for the creation and initial trading of "meme tokens". It employs a two-phase lifecycle for each token. The first phase involves trading on a contract-internal automated market maker (AMM) that follows a bonding curve model. The second phase begins once a token achieves a specified market capitalization, triggering a migration process that establishes a permanent liquidity pool on Sushiswap.

The system's architecture is centered around two primary smart contracts: a singleton factory contract, `MemeFactoryV3`, and a template token contract, `MemeTokenV3`, which is instantiated for each new token.



4.1 MemeFactoryV3

The Kensei Factory is the central hub and single entry point for all user interactions with the protocol, from token creation to trading and migration.

Key Responsibilities

- **Token deployment:** The factory is solely responsible for creating new Kensei Token contracts. This process is gated by a signature verification mechanism, requiring a valid signature from a trusted off-chain `signer` to authorize the creation of a new token. It uses the `CREATE2` opcode to deploy tokens to a predictable address.
- **Configuration management:** It holds the global protocol parameters, such as fee structures, market capitalization thresholds, and treasury addresses. These can be updated by the contract `owner`, but changes only affect tokens created after the update.
- **Trade routing:** It acts as a router for all bonding curve trades. Users interact with the factory's `buyExactIn(...)` and `sellExactIn(...)` functions, which in turn call the corresponding functions on the target Kensei Token contract. This centralizes interaction and simplifies the user experience.
- **State tracking:** The factory maintains a mapping of all valid Kensei Tokens (`isMemeToken`) and tracks which tokens have reached their migration threshold and are ready to be moved to Sushiswap (`readyForMigration`).

4.2 MemeTokenV3.sol

The Kensei Token is a ERC20 contract that encapsulates the entire lifecycle of a meme token, from its initial price discovery on a bonding curve to its final state as a fully liquid asset on Sushiswap.

Key Characteristics

- **Integrated bonding curve:** Each token contract manages its own internal bonding curve using virtual token and collateral reserves (`virtualTokenReserves`, `virtualCollateralReserves`). The price is determined algorithmically using a constant product formula. Trading functions are restricted, allowing calls only from the Kensei Factory.
- **Immutable configuration:** Upon deployment, all economic parameters for a token—such as its total supply, fee rates, and market cap limits—are stored as immutable variables, ensuring they cannot be altered post-launch.
- **Sushiswap integration:** The contract is tightly integrated with Sushiswap. At the moment of its creation, it programmatically creates and initializes its own Sushiswap pool. The initial price is deterministically set based on the projected reserves after a certain number of tokens are sold on the bonding curve.
- **Migration process:** The token contains all the necessary logic to transition its liquidity from the bonding curve to Sushiswap. The `migrate()` function orchestrates this process, which culminates in locking the liquidity permanently by holding the Sushiswap LP NFT within the token contract itself.
- **Transfer checks:** To ensure the integrity of the bonding curve phase, the token contract's `transfer` and `transferFrom` functions are overridden to prevent transfers to or from the associated Sushiswap pool address until the official migration is complete.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Ambiguity in `abi.encodePacked` allows creating tokens with unauthorized names and symbols

File(s): src/MemeFactoryV3.sol

Description: The `MemeFactoryV3` contract relies on a signature from a trusted signer to authorize the creation of new meme tokens. This signature is verified in the `_checkSignatureAndStore(...)` function against a message hash composed of multiple parameters, including the token's `_name` and `_symbol`.

The function uses `abi.encodePacked(_name, _symbol, ...)` to construct the data that gets hashed. `_name` and `_symbol` are dynamic `string` types; `abi.encodePacked` concatenates them directly without encoding their individual lengths. This creates an ambiguity where different combinations of `_name` and `_symbol` can produce the same output. For instance, `abi.encodePacked("TokenA", "B")` results in the same byte sequence as `abi.encodePacked("Token", "AB")`.

This ambiguity allows a user to request a signature for one set of parameters (e.g., `_name = "TokenA"`, `_symbol = "B"`), and then use that same signature to create a token with a different, unauthorized set of parameters (e.g., `_name = "Token"`, `_symbol = "AB"`).

```
function _checkSignatureAndStore(
    string memory _name,
    string memory _symbol,
    uint256 _nonce,
    bytes memory _signature
) internal {
    // ...
    // @audit-issue The use of abi.encodePacked with two consecutive dynamic types
    //              (_name and _symbol) creates a hash collision vulnerability.
    bytes32 message = keccak256(abi.encodePacked(
        _name, _symbol, _nonce, address(this), block.chainid, msg.sender
    ));

    if (
        !SignatureChecker.isValidSignatureNow(
            signer, MessageHashUtils.toEthSignedMessageHash(message), _signature
        )
    ) {
        revert InvalidSignature();
    }
    // ...
}
```

Recommendation(s): Consider replacing `abi.encodePacked` with `abi.encode` for hashing the message parameters. The `abi.encode` encodes the length of each dynamic type, which produces a unique output for each distinct set of parameters. Alternatively, consider hashing each string independently and using the results to create the final message.

Status: Fixed.

Update from the client: Fixed in commit 982d6b.

6.2 [Medium] Token creations are susceptible to denial-of-service

File(s): src/MemeFactoryV3.sol

Description: When a token is created through the functions `createMemeToken` or `createMemeTokenAndBuy` the logic flow is to create the constructor parameters and predict the deployed token address. The predicted address is then checked against the Sushiswap pool factory to determine if pool exists for token address and the `WETH` address for a specific pool fee. This check is shown below:

```
function createMemeToken(...) external payable returns (address) {
    // ...
    IMemeTokenV3.ConstructorParams memory params = IMemeTokenV3.ConstructorParams(...);
    address predicted = _predictMemeTokenAddress(params, _salt);
    _revertIfPoolExists(predicted);
    MemeTokenV3 token = new MemeTokenV3{salt: _salt}(params);
    // ...
}
```

The pool existence check queries the Sushiswap pool factory and if the returned address is zero, then no pool exists and execution can continue. If a nonzero address is returned, then a pool already exists and the transaction will revert. The assumption in this logic flow is that a Sushiswap pool would not exist for a token that has not yet been created, however it is actually possible to call `createPool` on the Sushiswap factory to create a pool for a token that doesn't exist.

An attacker can leverage this behavior by monitoring transactions to create tokens on the mempool, and upon identifying such a token creation they can calculate the address of the new token and frontrun the user transaction with their own which will create the pool before the token has been created. When the user transaction eventually executes afterwards, it will revert. This could be used to censor transactions from particular wallets, token names, or all token creations.

Recommendation(s): Consider creating a guide for users on how to overcome this situation by using private mempool services.

Status: Acknowledged.

6.3 [Info] Precision loss in fee calculation leaves dust ETH in the contract

File(s): src/MemeTokenV3.sol

Description: When a user buys tokens via the `buyExactIn(...)` function, the `_calculateFeeFromTotalAmount(...)` helper function is called to determine how the user's payment (`msg.value`) is split between the actual swap amount and protocol fees.

The calculation for the `swapAmount` involves an integer division: `swapAmount = (totalAmount * MULTIPLIER) / (MULTIPLIER + feeBPS)`. Due to the nature of integer arithmetic, if `(totalAmount * MULTIPLIER)` is not perfectly divisible by `(MULTIPLIER + feeBPS)`, a small remainder is lost in the calculation. The same error may occur in the computation of the fee, which may be already being computed on a smaller `swapAmount`. This "dust" represents a portion of the user's payment that is neither allocated to the `swapAmount` nor distributed as fees.

This dust amount remains as ETH balance within the `MemeTokenV3` contract. While these funds are not permanently lost, they accumulate over many transactions. The remaining ETH balance is only swept to the `treasury` address when the `migrate()` function is eventually called but not distributed as fees in the way they should.

Recommendation(s): Consider adjusting the fee calculation logic to ensure the full `totalAmount` sent by the user is accounted for in every transaction.

Status: Fixed.

Update from the client: Fixed in commit 9e78fd.

6.4 [Info] Storage variable `initialTokenSupply` can be immutable

File(s): src/MemeTokenV3.sol

Description: The contract `MemeTokenV3` has a storage variable `initialTokenSupply` which is only set in the constructor, and is not changed after the contract is deployed. Given that the storage variable will never change, it could be changed to `immutable` to reduce gas costs by removing the need to execute an `SLOAD` operation.

```
constructor(ConstructorParams memory _params) ERC20(...) {
    // ...
    // Once set this is never changed
    initialTokenSupply = _params.totalSupply;
    // ...
}
```

Recommendation(s): Consider changing the storage variable `initialTokenSupply` to `immutable`.

Status: Fixed.

Update from the client: Fixed in commit 982d6b.

6.5 [Info] Token approval during migrate can use internal _approve

File(s): src/MemeTokenV3.sol

Description: During a migration the remaining tokens are set as an approval to the position manager, so that the tokens can be used to provide liquidity to a target pool. The token approval is done using `this.approve(...)` which means that the contract does a call to itself. The code is shown below:

```
function migrate() external onlyFactory returns (...) {
    // ...
    this.approve(address(positionManager), tokensRemaining);
    // ...
}
```

This approach makes sense, since a call to self will make the `msg.sender` be the contract itself which makes the approval for itself to the position manager. However, this means that an additional call is executed and more gas must be consumed as the function dispatcher must be executed again. Instead, the internal `_approve` could be used which will achieve exactly the same result, but without the additional call which will reduce the gas costs in the `migrate` function.

Recommendation(s): Consider changing the `this.approve(...)` to use the ERC20 internal `_approve(...)` function.

Status: Fixed.

Update from the client: Fixed in commit 977a26.

6.6 [Info] Token constructor has unnecessary storage loads

File(s): src/MemeTokenV3.sol

Description: When a `MemeTokenV3` contract is deployed, the constructor sets many storage variables based on the provided arguments from the factory and a pool is created and initialized on Sushiswap at a particular tick based on the virtual token and collateral reserves. To calculate the correct tick to use, the internal function `_reservesAfterTokensSold` is called as shown below:

```
constructor(ConstructorParams memory _params) ERC20(_params.name, _params.symbol) {
    //...
    virtualCollateralReserves = _params.virtualCollateralReserves;
    virtualTokenReserves = _params.virtualTokenReserves;
    // ...
    uint256 vt0 = virtualTokenReserves;
    uint256 vc0 = virtualCollateralReserves;
    uint256 migrationThreshold = _params.tokensMigrationThreshold;
    (uint256 vtAtMcLower, uint256 v cAtMcLower) = _reservesAfterTokensSold(vt0, vc0, migrationThreshold);

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

The arguments `vt0` and `vt0` are set using the storage variables `virtualTokenReserves` and `virtualCollateralReserves` respectively. Since storage variables are used, this incurs an `SLOAD` operation for each local variable assignment. Storage reads are expensive, and since those two storage variable will not change between the time they are first assigned and then read from later, it is possible to use the values from the constructor params directly. This will remove two unnecessary `SLOAD` operations and save gas for users when deploying tokens.

Recommendation(s): Consider changing the assignment of `vt0` and `vc0` to use the constructor parameters instead of the storage variables to reduced gas costs.

Status: Fixed.

Update from the client: Fixed in commit 2458bd.

6.7 [Best Practices] Function `sellExactIn` is marked payable but shouldn't accept native assets

File(s): src/MemeTokenV3.sol

Description: The function `sellExactIn` in the `MemeTokenV3` is marked as `payable` and is only callable by the factory contract. There are no `msg.value` checks in this function and the factory is programmed to call this function without any native asset transfer, as shown below. This makes it impossible for the `sellExactIn` function to ever accept native assets, and as a result the `payable` marker can be removed.

```
// MemeFactoryV3.sol
function sellExactIn(...) external nonReentrant {
    // ...
    // The factory cannot call sellExactIn with any value
    (
        uint256 collateralToReceiveMinusFee,
        uint256 helioFee,
        uint256 dexFee
    ) = MemeTokenV3(payable(_token)).sellExactIn(_tokenAmount, _amountCollateralMin);
    // ...
}
```

Recommendation(s): Consider removing the `payable` marker from the `sellExactIn` function in the `MemeTokenV3` contract since it cannot accept native assets.

Status: Fixed.

Update from the client: Fixed in commit 982d6b.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Kensei's contract documentation

The Kensei team provided an overview of the core system components during meetings, presenting a clear explanation of the intended functionalities and addressing questions raised by the Nethermind Security team. Additionally, the Kensei team provided written documentation in the form of markdown files describing the protocol's different flows.

8 Test Suite Evaluation

8.1 Tests Output

```
forge test --match-contract MemeFlowV3Test
[] Compiling...
No files changed, compilation skipped

Ran 42 tests for test/MemeFlowV3.t.sol:MemeFlowV3Test
[PASS] test_BuyExactIn_TransferToUser_AndRefund() (gas: 3127149)
[PASS] test_CollectLpFees_AfterMigration_SendsToReceiver() (gas: 3179872)
[PASS] test_CollectLpFees_OnlyOwner_AndRevertsBeforeMigration() (gas: 2959027)
[PASS] test_CreateMemeTokenAndBuy_Succeeds() (gas: 3114528)
[PASS] test_CreateMemeToken_Succeeds() (gas: 2960480)
[PASS] test_Create_EdgeCase_ZeroTotalSupply() (gas: 119478)
[PASS] test_Create_EdgeCase_ZeroVirtualReserves() (gas: 118988)
[PASS] test_CurveProgressBps() (gas: 8067943)
[PASS] test_Events_BuyExactIn() (gas: 3128194)
[PASS] test_FactorySetConfig_OnlyOwner() (gas: 39748)
[PASS] test_FactorySetConfig_Validations() (gas: 50509)
[PASS] test_FeesCalculation() (gas: 3129676)
[PASS] test_GetAmountOutAndFee_PaymentTokenIn() (gas: 2953662)
[PASS] test_GetAmountOutAndFee_PaymentTokenOut() (gas: 2953560)
[PASS] test_InsufficientTokenReserves_BuyExactIn() (gas: 2972331)
[PASS] test_InvalidSignature_Reverts() (gas: 54094)
[PASS] test_LpFeeReceiver_DefaultAndSettable() (gas: 2958437)
[PASS] test_MarketCapCalculation() (gas: 3128017)
[PASS] test_MarketCapUpperLimit() (gas: 8070028)
[PASS] test_Migrate_ContractState_AfterMigration() (gas: 3176026)
[PASS] test_Migrate_ExistingPool_InitializedWithWrongPrice_ProceedsWithExistingPool() (gas: 3485005)
[PASS] test_Migrate_ExistingPool_Uninitialized_InitializesAndSucceeds() (gas: 3462028)
[PASS] test_Migrate_FeeCalculation_EdgeCases() (gas: 3174069)
[PASS] test_Migrate_HappyPath() (gas: 3173226)
[PASS] test_Migrate_InsufficientContractBalanceForFees() (gas: 3173936)
[PASS] test_Migrate_LiquidityProvision_EdgeCases() (gas: 3171970)
[PASS] test_Migrate_MarketCapCalculation_DuringMigration() (gas: 3171713)
[PASS] test_Migrate_NotReadyForMigration_Reverts() (gas: 2951995)
[PASS] test_Migrate_PoolNotExists_FallbackCreatesNewPool() (gas: 3160626)
[PASS] test_Migrate_RefundETH_EdgeCase() (gas: 3171345)
[PASS] test_Migrate_SweepToken_Called() (gas: 3171772)
[PASS] test_OnlyFactory_EnforcedOnToken() (gas: 2955745)
[PASS] test_PoolInitializedAtTokenCreation_WithEstimatedMigrationPrice() (gas: 2957434)
[PASS] test_ReceiveFunctionality() (gas: 15137)
[PASS] test_SellExactIn_ReturnsETHToUser() (gas: 8145035)
[PASS] test_SlippageProtection_BuyExactIn() (gas: 3040003)
[PASS] test_TokenBurn_OnMigration() (gas: 3170647)
[PASS] test_TradingStopped_WhenMcLowerReached() (gas: 8131002)
[PASS] test_TransferToPairBeforeMigration_Reverts() (gas: 3126129)
[PASS] test_UniswapV3Pair_SwapsBlockedBeforeMigration() (gas: 8241743)
[PASS] test_UsedSignature_Reverts() (gas: 2957823)
[PASS] test_VirtualReserves_Updates() (gas: 3129026)
Suite result: ok. 42 passed; 0 failed; 0 skipped; finished in 16.07ms (91.46ms CPU time)

Ran 1 test suite in 156.09ms (16.07ms CPU time): 42 tests passed, 0 failed, 0 skipped (42 total tests)
```

8.2 Automated Tools

8.2.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our cryptography Research team conducts cutting-edge internal research and collaborates closely with external partners on cryptographic protocols, consensus design, succinct arguments and folding schemes, elliptic curve-based STARK protocols, post-quantum security and zero-knowledge proofs (ZKPs). Our research has led to influential contributions, including Zinc (Crypto '25), Mova, FLI (Asiacrypt '24), and foundational results in Fiat-Shamir security and STARK proof batching. Complementing this theoretical work, our engineering expertise is demonstrated through implementations such as the Latticefold aggregation scheme, the Labrador proof system, zkvm-benchmarks, and Plonk Verifier in Cairo. This combined strength in theory and engineering enables us to deliver cutting-edge cryptographic solutions to partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.