Native DEX
Smart Contract Audit

NAT-001

Prepared for Native Labs

Dr. Nadim Kobeissi

Symbolic Software

August 23, 2023

Abstract

This audit report provides the results of a comprehensive security evaluation of Native Decentralized Exchange (DEX), a technology developed by Native Labs. Native DEX aims to democratize access to DEX creation, thereby enhancing the cryptocurrency swap experience for users and providing a comprehensive approach to token management for project teams. The technology is designed to address the inefficiency of heavy reliance on exchanges in the crypto ecosystem, thereby improving scalability. Native DEX acts as an invisible layer, allowing projects to build their own in-app DEX, eliminating the need for intermediaries, and enhancing the transaction process between projects and their respective communities.

The audit will provide an in-depth assessment of the inherent design, implementation, and operational effectiveness of Native's DEX technology, further scrutinizing its potential to realize the targeted future of cryptocurrency transactions. The audit will focus on technical validation, security analysis, performance evaluation, code quality inspection, interoperability review, evaluation of liquidity models, and user experience audit. The audit will cover core contracts, liquidity pool contracts, and other contracts of interest.

Contents

List of Acronyms

AMM	Automatic Market Maker	1
dApp	Decentralized Application	6
DEX	Decentralized Exchange	ii
EOA	Externally Owned Account	7
PMM	Professional Market Maker	1

1

Executive Summary

For the busy executive who just doesn't have the time!

This Executive Summary quickly collates together a list of relevant data points from this report.

1.1 | Target Summary

For more information, please review §2.

This report provides an overview and analysis of Native Labs's product, the Native DEX. The Native is an infrastructure that enables third-party entities to create their own decentralized exchange, democratizing access to DEX creation and improving the crypto swap experience. The objective of the audit is to evaluate the design, implementation, and operational effectiveness of the Native DEX technology.

The Native DEX operates by allowing projects to build their in-app DEX, eliminating intermediaries and enhancing transaction processes between projects and their communities. Its design removes the need for third-party exchange fees and resolves fragmentation in user experience, with an efficiency comparable to centralized exchanges.

This audit will examine the smart contract repository provided by Native, comprising a range of pricing and liquidity models. It allows for both on-chain and off-chain pricing via Automatic Market Maker (AMM)s and Professional Market Maker (PMM)s respectively. The evaluation will cover the workflow, which includes the pool setup and swap process, and the various smart contracts that make up the Native DEX system.

The audit's primary motivations include technical validation, security analysis, performance evaluation, code quality inspection, interoperability review, on-chain and off-chain analysis, evaluation of liquidity models, and a user experience audit.

The scope of the audit covers specific contracts of high and lower priority as indicated by the report. The audit was expected to kick off on July 18th and proceed with tasks such as audit planning, manual code review, static analysis, functional testing, and security assessment, with an expected delivery deadline of August 2nd, 2023.

1.2 | Static Analysis Results

For more information, please review §3.

In the audit of Native DeX's smart contracts, a static analysis was performed using Slither, a specialized framework for Ethereum smart contracts. The objective of this static analysis is to affirm the security and reliability of the DeX, bolstering its reputation within the blockchain ecosystem.

Two principal findings were identified:

1. **Immediately Overwritten Variables (NAT-01-001):** Multiple instances of immediately overwritten values were found in the `ExternalSwapRouterUpgradeable` contract's `swap1inch` function. This situation, which could lead to superfluous gas cost, less readable code, or unanticipated behavior, involves variables such as `amount`, `minReturn`, `amount_scope_0`, and `minReturn_scope_1`. It is advised to revisit these sections of the code to ensure the immediate overwrites are deliberate and avert potential vulnerabilities in the contract.

2. **Redundant Condition Check in Contract (NAT-01-002):** In the `NativePool` contract, a redundant condition check was discovered, where the `_fees[i]` variable was unnecessarily validated to be greater than or equal to zero. This is unwarranted as `_fees[i]`, an unsigned integer, can't be negative. Although not posing a security risk, this redundancy adds unnecessary complexity and can lead to confusion. It is recommended to simplify the condition check by eliminating the superfluous check to enhance the legibility of the code without affecting the intended functionality.

1.3 | Code Review Results

For more information, please review §4.

This part of the report highlights the findings from a thorough code review of the Native DEX smart contracts. The review focused on identifying potential issues that might compromise the functionality, security, and efficiency of the contracts. The discovered issues were categorized based on severity and potential impact, and each was furnished with a summary, detailed description, and actionable recommendations.

The review addressed issues such as stack depth limitations demanding custom compiler configurations, outdated dependencies with patched vulnerabilities, and unwarranted gas costs from on-chain hash operations. The primary aim of the review is to provide actionable recommendations for enhancing code quality and security, in addition to identifying potential vulnerabilities. Subsequent sections delve deeper into each finding for a more comprehensive understanding.

- **Stack Depth Issues Under Standard Compiler Configurations (NAT-01-003):** Several components in Native's DEX smart contract code exceeded Solidity's stack depth limit, demanding a custom and unsupported compiler configuration. The issue is commonly found in large contracts with complex functions and nested calls. We provided a thorough refactoring of three smart contracts to mitigate the risk of "stack depth exceeded" errors during compilation. This was achieved by extracting parts of the code into separate functions, reducing the number of variables in the function scope, and optimizing the code to use fewer stack slots.
- **Outdated Dependencies with Patched Vulnerabilities (NAT-01-004):** A routine check of smart contract dependencies revealed outdated versions of the `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable` packages. These older versions have had critical vulnerability disclosures, impacting the contract's security. The recommended action is to update the smart contract dependencies to their latest versions.
- **Unnecessary Gas Cost from On-Chain Hash Operation (NAT-01-005):** The `NativePool`, `NativePoolFactory`, and `NativeRouter` contracts have unnecessary on-chain keccak hashing for certain constants. This practice is inefficient due to the computational overhead and gas costs. The proposed solution is to replace these on-chain computations with precomputed hash values, enhancing contract efficiency and readability.

1.4 | Additional Analysis & Conclusion

For more information, please review §5.

This report concludes with an analysis of related aspects of the Native DEX smart contract stack. smart contracts. Six major aspects were evaluated: performance, code quality, interoperability, on-chain and off-chain transaction handling, liquidity models, and user experience.

- The performance of the Native DEX contracts was satisfactory, especially in terms of gas usage, scalability, and transaction speed. Although improvements can be made in gas usage optimization, scalability remains a key focus as transaction volumes increase.
- Despite some inconsistencies in coding style and sprawling function definitions, the code quality was satisfactory, aided significantly by comprehensive documentation. A more consistent coding style and refactored functions are recommended to improve readability and maintainability.
- The contracts exhibited robust interoperability capabilities, efficiently interacting with both internal and external entities. The team should continue to enhance this interoperability, particularly as the DeFi ecosystem evolves.
- The on-chain and off-chain transactions of the Native DEX contracts are effectively managed. Regular audits and reviews are recommended to ensure the accuracy, security, and effectiveness of these transactions.
- The liquidity models provided by Native are effective and user-friendly, but users should be informed about the potential risks, such as impermanent loss.
- The user experience audit showed that the contracts provide a positive user experience, with intuitive functions, accessibility, and transparency. Continuous improvements in these areas are recommended.

The conducted evaluations and reviews of the Native DEX smart contracts encompassed performance measurement, code scrutiny, interoperability assessment, on-chain and off-chain transaction analysis, liquidity model examination, and user experience auditing. This broad analysis has led to a comprehensive understanding of the operational effectiveness, code standard, interoperability, transaction processing, liquidity models, and user interaction of these contracts.

In the aspect of performance, the smart contracts presented acceptable gas usage, scalability, and transaction velocity metrics. The code analysis identified a number of inconsistencies in coding style and excessive complexity in function definitions, but the provided documentation was found to sufficiently compensate for these issues, leading to a positive assessment of the code quality. The smart contracts have proven to be robust in terms of interoperability, with successful interaction with both internal and external components.

The analysis of on-chain and off-chain transactions indicated a well-organized system that manages both transaction types, underpinning both the accuracy and security of transactions. The liquidity models offered by Native were found to be efficient and user-oriented. However, it is recommended that users fully understand the potential risks before participating in liquidity provision. The user experience audit indicates that the smart contracts have been designed with an emphasis on user accessibility, and the resulting transparency and intuitive functionality contribute to the overall user experience.

The results of this audit suggest that the Native DEX smart contracts have been developed with a significant degree of consideration and exactness. This study's findings should contribute to a better understanding of the integrity and reliability of the Native DEX platform for users and other relevant parties.

2

Target Overview

Native Labs produces the Native DEX [1], a technology that aims to allow third-parties to *"be their own DEX"*. This audit report evaluates the functionalities and effectiveness of Native DEX, a technology that aims to improve the cryptocurrency landscape by facilitating and democratizing access to DEX creation. The solution aims to enhance the cryptocurrency swap experience for users while providing a comprehensive approach to token management for project teams. By creating a balanced environment between decentralized and centralized finance, it aims to pave the way for exponential growth in the app layer.

Historically, the crypto ecosystem has demonstrated a heavy reliance on exchanges, with users often resorting to third-party intermediaries to transact with Decentralized Application (dApp)s and liquidity providers. Native DEX is designed to address this inefficiency, thereby improving scalability in the crypto space.

Acting as an invisible layer, Native allows projects to build their own in-app DEX, eliminating the need for intermediaries and enhancing the transaction process between projects and their respective communities. This technology eradicates the need for third-party exchange fees and the fragmentation associated with user experience, while ensuring efficiency comparable to centralized exchanges.

This audit will provide an in-depth assessment of the inherent design, implementation, and operational effectiveness of Native's DEX technology, further scrutinizing its potential to realize the targeted future of cryptocurrency transactions.

2.1 | Analysis Target Summary

Native's provided smart contract repository [2] contains a well-defined scope which acts as the basis for this audit engagement's target summary:

2.1.1 | Workflow

Native is an infrastructure that supports different kinds of pricing and liquidity models. For pricing models, Native supports both on-chain pricing (via AMMs) and off-chain pricing (PMMs). For liquidity source, Native supports private liquidity source (Externally Owned Account (EOA)), public liquidity source (liquidity pool contracts). Native also provides a variety of liquidity pool contracts to facilitate community-based liquidity sourcing. The workflow of Native is as follows:

2.1.1.1 | Pool Setup

1. The project owner deploys a `NativePool`, decides on the pricing model, sets the fees, and determines other pool configurations.
2. The project owner is required to provide liquidity support to the `NativePool` by granting allowance to the `NativePool` contract. This can either be through the project's own treasury EOA wallet or via a smart contract. Additionally, Native provides a variety of liquidity pool contracts to facilitate community-based liquidity sourcing.
3. Each `NativePool` includes a signer, which is a wallet used to sign orders. Only orders that have been signed can be executed on Native pools. This serves as a protective measure for the treasury, particularly for off-chain pricing. By default, the signer is created and hosted by Native, however, project owners also have the option to operate their own signer.

2.1.1.2 | Swap

1. The trader grants allowance to the `NativeRouter` and interacts with the Native off-chain API to receive the signed order.[1]

[1] An example for calling the API can be found here: https://docs.native.org/native-dev/native-v1/guide

2. The Native off-chain router reviews various liquidity sources, selects the route based on price, and returns the signed order. For hops using off-chain pricing (e.g., the PMM pricing model), the off-chain router obtains a quote from the market maker, fills in the `amountIn` and `amountOut` in the order, and then signs it.

3. Native incorporates a widget fee feature. Authorized partners can register the fee recipient with Native off-chain, and subsequently charge traders a fee for using their swap widget. The widgetFee signer, hosted by Native, signs the order to ensure the authenticity of the fee recipient and the fee rate.

4. Armed with the signed order, the trader can invoke the `NativeRouter` functions `exactInput` (for multi-hop) or `exactInputSingle` (for single hop) to execute the transaction on-chain.

5. The `NativeRouter` decodes the order and calls the `NativePool` for Native-supported liquidity from projects or other liquidity sources such as Uniswap v3, 1inch, and PancakeSwap.

2.1.2 | Smart Contracts

The Native smart contract repository contains the following contracts of interest:

2.1.2.1 | Core Contracts

- FILE `NativeRouter.sol`: Receive the order from trader with the 2 functions `exactInput` or `exactInputSingle`. Verify the order signature, process the widget fee and call corresponding liquidity source according to the order.
- FILE `ExternalSwapRouterUpgradable.sol`: Implement the external liquidity source that `NativeRouter` could call. Including PancakeSwap, Uniswap v3 and 1inch.
- FILE `PeripheryPayments.sol`: Include the logic for payment related to `NativeRouter` and also wrapping, unwrapping ETH.
- FILE `NativePoolFactory.sol`: Deploy the new NativePool with the configurations using `createNewPool`.

- FILE `NativePool.sol`: Define the pairs it supports and pricing model it uses. `swap` is called by `NativeRouter`. Treasury (EOA wallet or smart contract) will need to give allowance to this contract to support the swap.

2.1.2.2 | Liquidity Pool Contracts

These contracts are designed to accumulate liquidity from the community, incentivized by transaction fees and additional rewards provided by the project owner through the staking of LP tokens. Assets are deposited into these contracts, which should subsequently grant allowance to the corresponding `NativePool`. Native provides different types of liquidity models for project owners to choose from based on their specific needs.[2]

- FILE `NativeTreasury.sol`: A foundational contract intended to be inherited by other liquidity pools.
- FILE `NativeFixedPriceLiquidityPool.sol`: This liquidity model is appropriate for pegged assets or certain in-game tokens for which the project owner wishes to establish a fixed price. The value of LP tokens is calculated based on this fixed price.
- FILE `NativeUniswapV2LiquidityPool.sol`: This liquidity model operates similarly to the UniswapV2 model.
- FILE `NativePMMLiquidityPool.sol`: This liquidity model is utilized for collaboration with PMMs. It incorporates specific characteristics to facilitate PMM operation, such as withdrawal limits and the capability for PMMs to rebalance assets for hedging and liquidity management. This model is less permissionless and requires a higher degree of trust in the project owner and the market maker.
- FILE `NativeLPRewards.sol`: This contract enables the project team to deploy rewards for LP token staking, serving as an incentive for liquidity providers. Project owners can decide which LP tokens to incentivize and what rewards tokens to offer.

[2]Additional information can be found at https://docs.native.org/native-dev/native-v1/smart-contracts/liquidity-pools.

2.2 | Analysis Motivation

The primary motivations behind conducting this audit include:

1. **Technical Validation**: Confirmation of the technical functionality and integrity of the smart contracts to ensure they operate as expected and meet the objectives set by Native Labs.
2. **Security Analysis**: Identification of potential security vulnerabilities in the smart contract design and implementation, including but not limited to re-entrancy attacks, front-running, and overflow errors.
3. **Performance Evaluation**: Assessment of the operational efficiency of the smart contracts, focusing on aspects such as gas usage, scalability, and transaction speed.
4. **Code Quality Inspection**: Review of the code quality in terms of readability, maintainability, and adherence to established solidity and smart contract best practices.
5. **Interoperability Review**: Examination of the interaction and integration capabilities of the smart contracts with both internal (other contracts in the system) and external (third-party contracts and services) entities.
6. **On-chain and Off-chain Analysis**: Investigation of the handling of both on-chain and off-chain transactions, verifying the accuracy, security, and effectiveness of these operations.
7. **Evaluation of Liquidity Models**: Evaluation of the different liquidity models provided by Native, focusing on their effectiveness, usability, and potential risks.
8. **User Experience Audit**: Analysis of how the smart contracts impact the user experience in terms of ease of use, accessibility, and transparency.

2.3 | Audit Scope

The only contracts that are in scope for this audit are the contracts listed below, excluding any concerns regarding centralization or malicious administrator risk.

2.3.1 | High Priority

The following contracts are of high priority for this audit:

- FILE NativeRouter.sol
- FILE ExternalSwapRouterUpgradable.sol
- FILE PeripheryPayments.sol
- FILE NativePoolFactory.sol
- FILE NativePool.sol
- FILE NativePriceDecoupledLiquidityPool.sol
- FILE NativePMMLiquidityPool.sol
- FILE NativeLPRewards.sol

2.3.2 | Lower Priority

The following contracts are of lower priority for this audit:

- FILE NativeTreasury.sol
- FILE NativeUniswapV2LiquidityPool.sol
- FILE NativeFixedPriceLiquidityPool.sol

2.4 | Statement of Work

Symbolic Software's proposed privacy audit of the Native Contracts for Native Labs consists of one deliverable: a detailed report covering the following elements:

1. **Audit Planning**: This involves understanding the scope of the project and setting up audit objectives and procedures. Also, preliminary discussions with the Native DEX team to understand their requirements and expectations.

2. **Manual Code Review**: Review the provided smart contracts for structural flaws, bugs, code quality, compliance with coding standards, and possible security issues. This includes a thorough review of core contracts, liquidity pool contracts, and any other contracts of interest.

3. **Static Analysis**: Utilize tools like Echidna or similar to perform automated testing of the smart contracts. Identify vulnerabilities, check code coverage, verify functionalities, and test properties.

4. **Functional Testing**: Ensure that each function of the smart contracts works as intended. Verify the behavior of functions under both normal and adverse conditions.

5. **Workflow Testing**: Examine the flow of transactions and interactions between smart contracts, and verify they align with the intended design.

6. **Security Assessment**: Rigorously analyze the system for potential security flaws such as re-entrancy attacks, unchecked external calls, race conditions, and arithmetic overflows or underflows.

7. **Gas Optimization Analysis**: Examine the gas consumption of different functions and transactions. Suggest optimizations if necessary.

8. **Risk Assessment**: Identify and assess potential risks in the system. Suggest possible mitigations.

9. **Formulation of Audit Report**: After completing the audit, compile findings, observations, and recommendations into a comprehensive audit report.

10. **Post-Audit Support**: Discuss the report with the Native DEX team, provide clarifications if needed, and assist them in implementing the recommendations if required.

2.5 | Work Schedule

The following is an overview of the schedule through which this audit will be conducted. Tasks often overlap, so this is not an exact overview:

- **July 18**: Kickoff call, audit proposal for client review, audit preparation.
- **July 19 – July 29**: Audit planning, manual code review, static analysis, dynamic analysis, functional testing, workflow testing, security assessment, gas optimization analysis, risk assessment.
- **July 30 – August 2**: Report writeup, editing and delivery.

2.6 | Operational Notes

Any cryptographic file hashes mentioned in this report are 16-byte BLAKE3 hashes generated using version 1.4.0 of the `b3sum`[3] command: `b3sum -l 16 filename.txt`

[3] https://crates.io/crates/b3sum

3

Static Analysis

In the process of auditing the smart contracts for the Native DeX, we have chosen to conduct a static analysis using Slither [3], a static analysis framework specifically designed for Ethereum smart contracts. By leveraging Slither's capabilities, we aim to provide a thorough and accurate static analysis of Native's smart contracts. This will help ensure the security and reliability of the DeX, contributing to its overall trustworthiness in the blockchain ecosystem.

3.1 | NAT-001-001 Immediately Overwritten Variables

Severity: *Informational*
During our static analysis of the smart contracts, we identified an issue concerning immediately rewritten values in the FILE `ExternalSwapRouterUpgradeable` contract. This issue pertains to the `swap1inch` function, where certain variables are written twice in immediate succession, which could lead to wasted gas cost, less readable code, or in the worst case, unexepcted behavior.

- `amount`: written twice in the `swap1inch` function. Initially, it is assigned a value through the `abi.decode` function at line #276-279. Immediately after, at line #280, it is rewritten with the value of `sellerTokenAmount`.
- `minReturn`: written twice in immediate succession. It is first assigned a value through the `abi.decode` function at line #276-279, and then immediately overwritten with the value of `buyerTokenAmount` at line #281.
- `amount_scope_0`: first assigned a value through the `abi.decode` function at line #286-289, and then immediately overwritten with the value of `sellerTokenAmount` at line #290.

- `minReturn_scope_1`: first assigned a value through the `abi.decode` function at line #286-289, and then immediately overwritten with the value of `buyerTokenAmount` at line #291.

Recommendation: Declare variables once

These instances of immediately rewritten values could lead to potential issues in the contract's execution. It is recommended to review these sections of the code to ensure that the logic is correct and that the immediate overwrites are intentional. If they are not, it could lead to unexpected behavior or potential vulnerabilities in the contract.

3.2 | NAT-001-002 Redundant Condition Check in Contract

Severity: *Informational*

In the FILE `NativePool` contract, there's a redundant condition check for `_fees[i]` being greater than or equal to 0, which is unnecessary as `_fees[i]` is an unsigned integer and cannot be negative. This redundancy, while not a security risk, adds unnecessary complexity and could cause confusion, so it's recommended to remove this check to simplify the code.

During our static analysis of the NativePool contract, we identified an issue concerning a redundant condition check. This issue is located at line 280 in the FILE `NativePool` smart contract.

The code snippet in question is as follows:

```
require(
  (_fees[i] >= 0) && (_fees[i] <= 10000),
  "Fee should be between 0 and 10k basis points"
);
```

The variable `fees[i]` is of type `uint` (unsigned integer), which means it cannot hold negative values. Therefore, the condition check `(_fees[i] ⩾ 0)` is redundant as `_fees[i]` will always be greater than or equal to 0 by definition.

This redundancy does not pose a security risk, but it does add unnecessary complexity to the code and could potentially lead to confusion for developers or auditors reviewing the contract in the future.

Recommendation: Simplify condition check

Simplify the condition check by removing the redundant check. The revised code should look as follows:

```
require(
  _fees[i] <= 10000,
  "Fee should be between 0 and 10k basis points"
);

```

This change will make the code cleaner and easier to understand, without altering the intended functionality.

4

Code Review

This chapter presents the findings of a detailed code review of the Native DEX smart contracts. The review aims to identify and address potential issues in the code that could affect the functionality, security, and efficiency of the contracts. The findings are categorized based on their severity and potential impact. Each finding is accompanied by a brief summary, a detailed description of the issue, and a recommendation for resolving the issue.

The review covers a range of issues, from stack depth issues that require custom compiler configurations, outdated dependencies with patched vulnerabilities, to unnecessary gas costs from on-chain hash operations. The goal of this review is not only to identify potential problems but also to provide actionable recommendations to improve the overall quality and security of the code. The following sections will delve into each finding, providing a comprehensive understanding of the issues at hand and the proposed solutions.

4.1 | NAT-001-003 Stack Depth Issues Under Standard Compiler Configurations

Severity: *Low*
The Native DEX smart contract code was found to exceed Solidity's stack depth limit in several components, necessitating the use of a custom compiler configuration. To address this, Symbolic Software provided thorough code refactoring across three smart contracts to reduce function complexity and the number of local variables, aiding in adhering to the Solidity compiler's limit and mitigating the risk of "stack depth exceeded" errors during compilation.

Several components in Native's DEX smart contract code exceeded Solidity's stack depth limit upon compilation, requiring the use of a custom and unsup-

ported compiler configuration to compile the code. The problem arises when the Solidity compiler tries to allocate more than 1024 stack slots, which is the maximum limit. This issue is prevalent in large contracts with complex functions and nested calls. The issue may be resolved by refactoring the code to reduce the complexity of functions and the number of local variables, which will help to keep the stack depth within the limit set by the Solidity compiler.

Stack depth issues are generally addressed through refactoring the code to reduce the complexity of functions and the number of local variables. This is achieved by extracting parts of the code into separate functions, reducing the number of variables in the function scope, and optimizing the code to use fewer stack slots. These changes help to keep the stack depth within the limit set by the Solidity compiler.

In order to help address this issue, Symbolic Software provided Native DEX with a thorough refactoring spanning three smart contracts. These changes are aimed at reducing the complexity of the contracts and thus mitigating the risk of "stack depth exceeded" errors during compilation.

Here's a breakdown of the changes in each contract:

- FILE `ExternalSwapRouterUpgradeable`:
  - This contract has undergone significant refactoring. The changes are primarily focused on reducing the complexity of the functions `swapPancake`, `swapUniswapV3`, and `swap1inch`.
  - The main change is the extraction of a common piece of code into a new function `handleEthCase`. This function handles the case where the user calls with ETH, reducing code duplication and making the code more readable.
  - Additionally, the code has been restructured to use fewer local variables, which can help avoid stack depth issues. For example, the `buyerTokenAmount` and `sellerTokenAmount` variables are now declared and assigned in a more compact way.
  - The `emitSwap1Inch` function has been extracted from `swap1inch` to further reduce the complexity of the latter.
- FILE `NativePool`:
  - The `initialize` function of this contract has been simplified by replacing multiple parameters with a single `NewPoolConfig` struct. This reduces the number of local variables and arguments, which can help avoid stack depth issues.

  - The `NewPoolConfig` struct includes all the necessary parameters for initialization, making the function call more straightforward and reducing the risk of errors.

- `NativePoolFactory`: Similar to `NativePool`, the `createNewPool` function has been simplified by replacing multiple parameters with a single `NewPoolConfig` struct. This reduces the number of local variables and arguments, which can help avoid stack depth issues.

In summary, these changes aim to reduce the complexity of the contracts and the number of local variables used, which can help avoid "stack depth exceeded" errors during Solidity compilation.

> **Recommendation: integrate changes provided by Symbolic Software**
>
> Simply integrate our changes documented above, and you're good to go!

4.2 | NAT-001-004 Outdated Dependencies with Patched Vulnerabilities

Severity: *Low*
The Native DEX smart contracts have been identified to utilize outdated versions of the `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable` dependencies, which have documented critical vulnerabilities impacting specific versions. These vulnerabilities could lead to a range of issues including incorrect computations, and are rectifiable via an update to the latest versions, ensuring patched vulnerabilities.

A routine check of smart contract dependencies has revealed that the Native DEX contracts pin outdated dependency versions of the @openzeppelin/contracts and @openzeppelin/contracts-upgradeable packages, some of which have since had critical vulnerability disclosures. These vulnerabilities affect versions 3.2.0 to 4.9.1 of `@openzeppelin/contracts` and all versions up to 4.9.1 of `@openzeppelin/contracts-upgradeable`. The identified issues are as follows:

- **Incorrect Calculation**: This issue could lead to incorrect computations in the contracts.

- **`TransparentUpgradeableProxy` Clashing Selector Calls**: This issue could prevent certain function calls from being correctly delegated in the context of upgradeable contracts.
- **`GovernorCompatibilityBravo` Trimming Proposal Calldata**: This issue could lead to the trimming of proposal calldata in the `GovernorCompatibilityBravo` contract.
- **Governor Proposal Creation Frontrunning**: This issue could allow an attacker to block the creation of new proposals in the governor contract by frontrunning.
- **Merkle Proof Multiproofs Arbitrary Leaf Proving**: This issue could allow an attacker to prove arbitrary leaves in specific Merkle trees.

Recommendation: update smart contract dependencies

For both packages, a fix is available and can be applied by running the `npm audit fix` command. This command automatically installs any compatible updates to vulnerable dependencies. Update the smart contract dependencies to the latest versions, ensuring that all known vulnerabilities are patched. This will help to ensure the security and reliability of the smart contracts.

4.3 | NAT-001-005 Unncessary Gas Cost from On-Chain Hash Operation

Severity: *Informational*

The FILE `NativePool`, FILE `NativePoolFactory`, and FILE `NativeRouter` contracts contain unnecessary on-chain keccak hashing for certain constants, which was inefficient due to the computational overhead and gas costs. The issue may be resolved by replacing these on-chain computations with precomputed hash values, improving contract efficiency and readability.

During our review of the smart contracts, we identified an issue concerning unnecessary on-chain Keccak hashing in the FILE `NativePool`, FILE `NativePoolFactory`, and FILE `NativeRouter` contracts. This issue was found in the computation of `ORDER_SIGNATURE_HASH`, `INIT_SELECTOR`, `UPGRADE_SELECTOR`, and `EXACT_INPUT_SIGNATURE_HASH`:

```
// NativePool.sol:
bytes32 private constant ORDER_SIGNATURE_HASH =
  keccak256(
    "Order(uint256 id,address signer,address buyer,address seller,address
  ↪ buyerToken,address sellerToken,uint256 buyerTokenAmount,uint256
  ↪ sellerTokenAmount,uint256 deadlineTimestamp,address caller,bytes16
  ↪ quoteId)"
  );

// NativePoolFactory.sol:
bytes4 public constant INIT_SELECTOR =
  bytes4(
    keccak256(
      bytes(
        "initialize((address,address,address,address,bool,bool,uint256[],
  ↪ address[],address[],uint256[]),address)"
      )
    )
  );
bytes4 public constant UPGRADE_SELECTOR = bytes4(keccak256(bytes("upgradeTo
  ↪ (address)")));

// NativeRouter.sol:
bytes32 private constant EXACT_INPUT_SIGNATURE_HASH =
  keccak256(
    "NativeSwapCalldata(bytes32 orders,address recipient,address signer,
  ↪ address feeRecipient,uint256 feeRate)"
  );
```

These constants are computed on-chain using the `keccak256` function. However, this computation is unnecessary and inefficient as these constants can be pre-computed off-chain and hard-coded into the contract. This is because the values being hashed are constant and do not change during the contract's execution.

Recommendation: Use precomputed hash values

Use precomputed hash values for constants in the contract. This practice improves the contract's efficiency by reducing the computational overhead and gas costs associated with on-chain hashing. It also simplifies the contract's code, making it easier to read and understand.

5

Additional Analysis & Conclusion

This concluding section of the report presents a detailed analysis of various aspects of the Native DEX smart contracts. The evaluation focuses on operational efficiency, code quality, interoperability, handling of on-chain and off-chain transactions, liquidity models, and user experience. Each section provides a comprehensive review and evaluation of the respective aspect, highlighting the strengths and areas for improvement. The aim is to provide a holistic understanding of the functionality, efficiency, and usability of the Native DEX smart contracts. The evaluations are based on rigorous testing and analysis, ensuring an accurate and objective assessment of the contracts. In this conluding section, we consider the following additional aspects:

1. **Performance Evaluation**: Assessment of the operational efficiency of the smart contracts, focusing on aspects such as gas usage, scalability, and transaction speed.
2. **Code Quality Inspection**: Review of the code quality in terms of readability, maintainability, and adherence to established solidity and smart contract best practices.
3. **Interoperability Review**: Examination of the interaction and integration capabilities of the smart contracts with both internal (other contracts in the system) and external (third-party contracts and services) entities.
4. **On-chain and Off-chain Analysis**: Investigation of the handling of both on-chain and off-chain transactions, verifying the accuracy, security, and effectiveness of these operations.
5. **Evaluation of Liquidity Models**: Evaluation of the different liquidity models provided by Native, focusing on their effectiveness, usability, and potential risks.
6. **User Experience Audit**: Analysis of how the smart contracts impact the user experience in terms of ease of use, accessibility, and transparency.

5.1 | Performance Evaluation

Evaluation: ●●●●○

The performance evaluation of the Native DEX smart contracts was conducted with a focus on operational efficiency, particularly in terms of gas usage, scalability, and transaction speed. The contracts were tested under various conditions to assess their performance in different scenarios. The gas usage of the contracts was analyzed to identify any inefficiencies or areas for optimization. The scalability of the contracts was evaluated by testing their performance under high transaction volumes. The transaction speed was assessed by measuring the time taken to execute various functions in the contracts.

The results of the performance evaluation indicated that the Native DEX contracts are sufficiently efficient in terms of gas usage. While the findings discussed in **NAT-01-003** led to possible gas usage improvements, the contracts were found to have already been optimized to minimize gas consumption, which contributes to lower transaction costs for users. The contracts also demonstrated scalability, maintaining consistent performance even under high transaction volumes. The transaction speed was found to be satisfactory, with most functions executing swiftly.

Recommendation: While the performance of the Native DEX smart contracts is within reasonable best-effort range given the use case scenario, it is recommended that the team continues to monitor and optimize gas usage, especially in light of the evolving Ethereum gas price landscape. Regular performance evaluations should be conducted to ensure that the contracts remain efficient and cost-effective for users. Additionally, as the platform grows and transaction volumes increase, scalability should remain a key focus to ensure consistent performance.

5.2 | Code Quality Inspection

Evaluation: ●●●○○

The code quality inspection involved a comprehensive review of the Native DEX smart contracts. The contracts were evaluated in terms of readability, maintainability, and adherence to established Solidity and smart contract best practices. The code was examined for clarity and simplicity, with a focus on the use of clear variable names, concise functions, and comprehensive comments. The main-

tainability of the code was assessed by considering factors such as modularity, reusability, and the ease of adding new features or making changes.

Upon inspection, it was observed that the functions within the contracts were sometimes defined in a sprawling manner, and there was no single clear coding style maintained throughout the codebase. This lack of consistent style and sprawling function definitions could potentially make the code harder to read and maintain. However, it's important to note that despite these inconsistencies, the provided documentation was comprehensive and well-structured, which greatly facilitated understanding of the codebase.

The contracts, while not strictly adhering to a single coding style, were still found to be reasonably well-structured and organized. The use of meaningful variable names and the presence of comprehensive comments throughout the codebase contributed to its readability. Despite the sprawling nature of some functions, the code was modular and reusable, which enhances maintainability.

In conclusion, while there is room for improvement in terms of coding style consistency and function organization, the overall quality of the code in the Native DEX contracts is satisfactory. The comprehensive documentation provided by the team significantly aids in understanding the codebase, mitigating some of the potential readability issues caused by the lack of a consistent coding style and sprawling function definitions.

Recommendation: It is recommended that the team adopts a more consistent coding style across the codebase to improve readability and maintainability. This could involve the use of a style guide or linter to enforce consistency. Additionally, refactoring some of the sprawling functions could make the code easier to read and maintain. Despite these areas for improvement, the comprehensive documentation is highly beneficial and should continue to be updated and expanded as the codebase evolves.

5.3 | Interoperability Review

Evaluation: ●●●●●

The interoperability review involved examining the interaction and integration capabilities of the Native DEX smart contracts with both internal and external entities, namely PancakeSwap, Uniswap v3 and 1inch. The contracts were tested for their ability to interact seamlessly with other contracts in the system, as well as with third-party contracts and services. The review also considered the ease of integrating the contracts into different blockchain environments and platforms.

The review found that the Native DEX contract achieve interoperability. The contracts interact as expected with other contracts in the system, facilitating efficient and secure transactions. The contracts are also capable of integrating smoothly with third-party contracts and services, which enhances their versatility and usability.

Recommendation: The Native DEX smart contracts have demonstrated robust interoperability capabilities. It is recommended that the team continues to maintain and enhance this interoperability, particularly as the DeFi ecosystem continues to grow and evolve. This could involve integrating with additional third-party contracts and services, or adapting to new standards and protocols as they emerge.

5.4 | On-chain and Off-chain Analysis

Evaluation: ●●●●●

The on-chain and off-chain analysis focused on the handling of both types of transactions within the Native DEX smart contracts. The aim was to verify the accuracy, security, and effectiveness of these operations. Native supports both on-chain pricing (via AMMs) and off-chain pricing (via PMMs). The workflow of Native involves a series of on-chain and off-chain operations, starting from pool setup to the execution of swaps.

During the pool setup, the project owner deploys a `NativePool`, sets the pricing model, and provides liquidity support by granting allowance to the NativePool contract. This process is conducted on-chain. The project owner also sets up a signer, which is used to sign orders. This can be done either on-chain or off-chain, depending on whether the project owner chooses to operate their own signer or use the default signer provided by Native.

The swap process involves both on-chain and off-chain operations. The trader interacts with the Native off-chain API to receive a signed order. The off-chain router reviews various liquidity sources, selects the route based on price, and returns the signed order. Once the trader has the signed order, they can execute the transaction on-chain by invoking the NativeRouter functions.

The core contracts, such as `NativeRouter.sol` and `NativePool.sol`, handle the on-chain operations, while the off-chain operations are handled by the off-chain API and router. The analysis of these contracts and their operations revealed a well-structured system that effectively handles both on-chain and off-chain transactions.

In conclusion, the on-chain and off-chain analysis of the Native DEX smart contracts demonstrated a robust system that effectively handles both types of transactions. The system's design ensures the accuracy and security of these operations, contributing to the overall effectiveness of the Native DEX platform.

Recommendation: The on-chain and off-chain operations of the Native DEX smart contracts are well-structured and effective. It is recommended that the team continues to monitor and optimize these operations, particularly in response to changes in the Ethereum network or the broader blockchain environment. Regular audits and reviews should be conducted to ensure the accuracy, security, and effectiveness of both on-chain and off-chain transactions.

5.5 | Evaluation of Liquidity Models

Evaluation: ●●●●○

The evaluation of liquidity models involved assessing the different liquidity models provided by Native. The models were evaluated for their effectiveness, usability, and potential risks. The evaluation considered factors such as the ease of providing liquidity, the return on liquidity provision, and the security of the liquidity pools.

The evaluation found that the liquidity models provided by Native are effective and user-friendly. The models make it easy for users to provide liquidity and earn returns. The liquidity pools are secure, with robust mechanisms in place to protect users' funds. However, the evaluation also identified potential risks associated with the liquidity models, such as the risk of impermanent loss. Users are advised to understand these risks before providing liquidity.

Recommendation: While the liquidity models provided by Native are effective and user-friendly, it is recommended that the team continues to educate users about the potential risks associated with providing liquidity, such as impermanent loss. This could involve creating educational content or tools to help users understand these risks. Additionally, the team should continue to monitor and optimize the liquidity models to ensure they remain effective and competitive in the evolving DeFi landscape.

5.6 | User Experience Audit

Evaluation: ●●●●◐

The user experience audit involved analyzing how the Native DEX smart contracts impact the user experience. The audit focused on the ease of use, accessibility, and transparency of the contracts. The ease of use was assessed by considering the simplicity and intuitiveness of the contract functions. The accessibility was evaluated by examining the availability and responsiveness of the contracts. The transparency was assessed by considering the clarity and comprehensiveness of the contract documentation and user guides.

The audit found that the Native DEX contracts provide a positive user experience, especially when paired with the available developer documentation [4]. The contract structure is fairly accessible, with intuitive functions that make it simple for users to perform transactions. The contracts are accessible and responsive, ensuring that users can interact with them smoothly and efficiently. The contracts are also transparent, with clear and comprehensive documentation and user guides that help users understand how to use them effectively.

Recommendation: The user experience provided by the Native DEX smart contracts is positive, but there is always room for improvement. It is recommended that the team continues to focus on improving the ease of use, accessibility, and transparency of the contracts. This could involve refining the user interface, improving error messages, or enhancing the documentation and user guides. Regular user feedback should be sought to identify areas for improvement and ensure that the contracts continue to meet the needs of users.

5.7 | Conclusion

The conducted evaluations and reviews of the Native DEX smart contracts encompassed performance measurement, code scrutiny, interoperability assessment, on-chain and off-chain transaction analysis, liquidity model examination, and user experience auditing. This broad analysis has led to a comprehensive understanding of the operational effectiveness, code standard, interoperability, transaction processing, liquidity models, and user interaction of these contracts.

In the aspect of performance, the smart contracts presented acceptable gas usage, scalability, and transaction velocity metrics. The code analysis identified a number of inconsistencies in coding style and excessive complexity in function definitions, but the provided documentation was found to sufficiently compensate for these issues, leading to a positive assessment of the code quality. The

smart contracts have proven to be robust in terms of interoperability, with successful interaction with both internal and external components.

The analysis of on-chain and off-chain transactions indicated a well-organized system that manages both transaction types, underpinning both the accuracy and security of transactions. The liquidity models offered by Native were found to be efficient and user-oriented. However, it is recommended that users fully understand the potential risks before participating in liquidity provision. The user experience audit indicates that the smart contracts have been designed with an emphasis on user accessibility, and the resulting transparency and intuitive functionality contribute to the overall user experience.

The results of this audit suggest that the Native DEX smart contracts have been developed with a significant degree of consideration and exactness. This study's findings should contribute to a better understanding of the integrity and reliability of the Native DEX platform for users and other relevant parties.

6

About Symbolic Software

Symbolic Software[1], established in Paris, France in 2017, is a software consultancy specializing in applied cryptography and software security. The firm has executed over 300 cryptographic software audits within the European information security sector and has made significant contributions to the field by publishing peer-reviewed cryptographic research software. Demonstrating their creativity and passion, Symbolic Software also develops captivating puzzle games that have been critically acclaimed.

Offering wide-ranging expertise in cryptographic software audits, Symbolic Software has audited critical cryptographic components of global platforms, ranging from password managers to cryptocurrencies. The company has developed Verifpal® and Noise Explorer, innovative research software for cryptographic engineering, which have contributed to peer-reviewed scientific publications. Symbolic Software's portfolio is marked by collaboration with leading entities such as Cure53 and the Linux Foundation, and they have successfully audited critical technologies like MetaMask and key COVID-19 contact tracing applications in Europe.

In addition to their software security expertise, Symbolic Software creates engaging puzzle games. Their game, Dr. Kobushi's Labyrinthine Laboratory®, available on Nintendo Switch™ and Steam, has been highly praised by reviewers.

[1] Stay updated on Symbolic Software's latest work by visiting `https://symbolic.sofware`.

References

[1] Native Labs. *Native Whitepaper*. 2023. URL: https://docs.native.org/native-whitepaper/ (visited on 07/18/2023).

[2] Native Labs. *Native Contracts*. 2023. URL: https://github.com/Native-org/native-contracts (visited on 07/18/2023).

[3] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: a static analysis framework for smart contracts." In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. 2019, pp. 8–15.

[4] Native Labs. *Native V1 Developer Documentation*. 2023. URL: https://native-1.gitbook.io/native-dev/native-v1/overview (visited on 07/31/2023).