NOKS AI Security Review

Duration: October 2, 2025 - October 5, 2025

December 14, 2025

Conducted by **KeySecurity**
Georgi Krastenov, Lead Security Researcher

Table of Contents

1 About KeySecurity

KeySecurity is an innovative Web3 security company that hires top talented security researchers for your project. We have conducted over 40 security reviews for different projects which hold over $500,000,000 in TVL. For security audit inquiries, you can reach out to us on Twitter/X or Telegram @gkrastenov, or check our previous work here.

2 About NOKS AI

NOKS AI is omnichain execution layer for AI agents (PERPs, DEXs, intelligent stablecoin management).

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	NOKS AI
Repository	https://github.com/noks-ai
Commit hash	1c538a88d0b888904479f196cd906f694e52aa6e
Review Commit hash	cf76465d69d860a5490894bad4d87c7cde680987
Documentation	N/A
Methods	Manual review

Scope

FeeTransferer.sol
DexRouter.sol

Timeline

October 2, 2025	Audit kick-off
October 5, 2025	Mitigation review
October 5, 2025	Final report

Issues Found

Severity	**Count**
Critical	0
High	0
Medium	0
Low	0
Information	3
Total	**3**

6 Findings

6.1 Information

6.1.1 Unnecessary external call when no fee remainder exists

Severity: *Information*

Context: FeeTransferer.sol#L164

Description: The `_fromSwapFee` and `_toSwapFee` functions calculate the total fee and a portion of this fee is distributed to the partners. Any remaining amount that is not distributed is sent to the destination defined in the first element of the `FeeReceiver` array.

When all swap fees are distributed to partners, the contract still executes `_sendETH` or `_claimToken` with a zero amount.

Although no funds are transferred, this results in an unnecessary external call, increasing gas usage and potentially exposing the system to avoidable risks.

Recommendation: Add a conditional check to skip `_sendETH` and `_claimToken` calls when `totalFee - totalPartnerFee == 0` to optimize gas and reduce unnecessary external interactions.

Resolution and Client comment: Resolved.

6.1.2 Unused token balance logic in _getBalanceOf

Severity: *Information*

Context: FeeTransferer.sol#L49-L65

Description: The `_getBalanceOf` function in the `FeeTransferer` contract can retrieve either the native currency balance of the user or their balance of a specific token. When the token is different from `_ETH` (0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE), it calls the token's `balanceOf` function to get the current balance. This functionality works properly but is never used. The `_getBalanceOf` function is used to get the contract's balance before execution when `inputFee == false`, meaning that `toToken` is native.

```
outputTokenBalanceBeforeOrInputFees = _getBalanceOf(
    baseRequest.toToken,
    address(this)
);
```

The second time `_getBalanceOf` is used is in the `_toSwapFee` function, where it retrieves `balanceNow`. In this case, the `dstToken` must again be equal to `_ETH`.

```
 uint256 balanceNow = _getBalanceOf(dstToken, address(this));
require(balanceNow > balanceBefore, "no tokens in contract");
require(dstToken == _ETH, "output fee can only be in ETH")
```

Recommendation: Remove the unused token balance logic or restrict`_getBalanceOf` to handle only the native currency. This reduces contract size, simplifies the codebase, and minimizes maintenance overhead.

Resolution and Client comment: Resolved.

6.1.3 Risk of reentrancy via external ETH transfers to partners

Severity: *Information*

Context: FeeTransferer.sol

Description: When distributing native token fees to partners, the contract makes an external call to the partner's address in order to transfer their share of the fee:

```
_sendETH(fees[i].destination, partnerFee);
```

The `_sendETH` function uses low-level `call` to transfer ETH.

```
function _sendETH(address to, uint256 amount) internal {
        assembly ("memory-safe") {
            function _revertWithReason(m, len) {
                mstore(
                    0,
                    0
                        x08c379a000000000000000000000000000000000000000000000000000000000

                )
                mstore(
                    0x20,
                    0
                        x0000002000000000000000000000000000000000000000000000000000000000

                )
                mstore(0x40, m)
                revert(0, len)
            }

            let success := call(gas(), to, amount, 0, 0, 0, 0)
            if eq(success, 0) {
                _revertWithReason(
                    0
                        x0000001c20636f6d6d697373696f6e207769746820657468657220657272726f72
                        , //commission with ether error
                    0x60
                )
            }
        }
    }
```

This design allows execution of arbitrary code at the recipient (fees[i].destination) address. If a malicious partner address is specified, it could exploit the external call to attempt reentrancy attacks (e.g., calling back into fee distribution or other state-mutating functions before completion). Although the assembly implementation reverts properly on failure, it does not prevent reentrancy attempts.

A malicious partner address can execute arbitrary fallback logic when receiving ETH. If the contract does not employ proper reentrancy protection, this can lead to reentrancy vulnerabilities or state inconsistencies.

Recommendation: Apply a `nonReentrant` modifier (e.g., from OpenZeppelin's ReentrancyGuard) to the `smartSwapByOrderId` and `smartSwapTo` functions.

Resolution and Client comment: Resolved.