10AP17 Audits

Oku Audit

Prepared by: 10ap17

Contents

1 Executive Summary

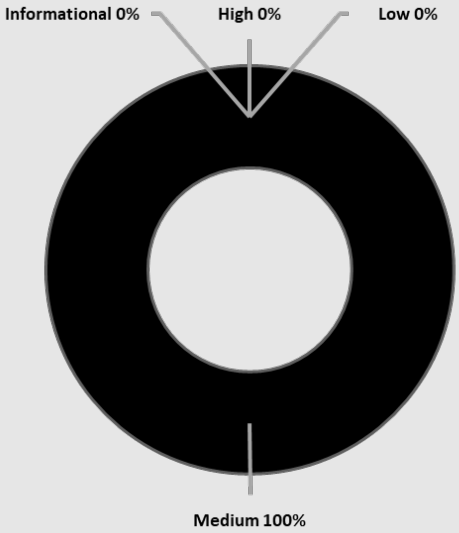
1.1 Overview

Oku is a non-custodial DeFi meta-aggregator deployed on Optimism, designed to provide seamless trading and bridging functionality while ensuring compatibility with a wide range of tokens. The platform supports advanced features, including market and limit orders, dynamic token whitelisting to restrict non-standard or incompatible tokens, and an off-chain mechanism for automated transaction execution. This off-chain system operates without privileged access, allowing any user to fulfill its duties if needed.The protocol's only privileged role, the owner, has the authority to manage crucial functionalities such as withdrawing fees, registering oracles, setting minimum order sizes, defining maximum lending orders, and introducing new sub-keeper implementations. The codebase leverages a dynamic array for tracking pending orders by their IDs, prioritizing readability over gas optimization, which is feasible given its deployment on Layer 2 networks where gas costs are lower.With its thoughtful design choices, robust feature set, and focus on usability, Oku aims to provide a secure and user-friendly platform for decentralized trading and bridging, catering to both individual users and professional traders.

1.2 Audit Scope

- contracts/automatedTrigger/AutomationMaster.sol
- contracts/automatedTrigger/Bracket.sol
- contracts/automatedTrigger/IAutomation.sol
- contracts/automatedTrigger/OracleLess.sol
- contracts/automatedTrigger/StopLimit.sol
- contracts/libraries/ArrayMutation.sol
- contracts/oracle/External/OracleRelay.sol
- contracts/oracle/External/PythOracle.sol

1.3 Summary of Findings



	Issue Title	Severity
1	Incorrect staleness check in `currentValue` function leading to use of stale prices	Medium
2	Unspent token allowance blocks reuse of swap targets	Medium

2 Detailed Findings

2.1 [Medium] Incorrect staleness check in `currentValue` function leading to use of stale prices

2.1.1 Vulnerability Detail

The `currentValue` function in `PythOracle.sol` has an incorrect implementation of the staleness check for prices retrieved from the oracle. The condition used to verify that the price is not stale is implemented incorrectly, resulting in only stale prices being accepted by the protocol, while up-to-date prices are rejected. This flaw will lead to incorrect calculations and the potential for users to lose funds.

```
function currentValue() external view override returns (
   uint256) {
        IPyth.Price memory price = pythOracle.
           getPriceUnsafe(tokenId);
        require(
            price.publishTime < block.timestamp -
               noOlderThan,
            "Stale Price"
        );
        return uint256(uint64(price.price));
    }
```

This condition checks whether the `publishTime` of the price is less than the current block timestamp minus the `noOlderThan` threshold. However, this logic is incorrect, because it allows stale prices to pass the check while rejecting fresh prices.

2.1.2 Proof of Concept

Consider the following scenario where:

1. The `block.timestamp` is set to 10,000.
2. The `noOlderThan` parameter is set to 100, specifying that the price's publish time must not be older than 100 seconds before the current `block timestamp`.
3. A price is retrieved with a `publishTime` of 9,950.
4. However, the require statement incorrectly checks: `price.publishTime < block.timestamp - noOlderThan`. Substituting the values: 9,950 < 9,900, which evaluates to false.

Due to this incorrect logic, the require statement rejects the price, even though it meets the intended validity criteria.

2.1.3 Impact

The incorrect implementation of the staleness check in the `currentValue` function allows only stale prices to be used by the protocol. This leads to inaccurate calculations of exchange rates, which in turn will result in users potentially losing their funds due to incorrect valuation of assets.

2.1.4 Recommendations

Consider updating the staleness check in the `currentValue` function to correctly validate that the price data is recent. Replace the existing condition with the following:

```
require(
    price.publishTime >= block.timestamp - noOlderThan,
    "Stale Price"
);
```

This ensures that only prices published within the acceptable timeframe (`noOlderThan`) are considered valid.

2.2 [Medium] Unspent token allowance blocks reuse of swap targets

2.2.1 Vulnerability Detail

The use of `safeApprove` in the execute function will cause a revert for the protocol when attempting token swaps via the target contract if a non-zero allowance already exists. This occurs because `safeApprove` reverts when trying to set an allowance on a token that already has a non-zero approval amount.

```
function execute(
        address target,
        bytes calldata txData,
        Order memory order
    ) internal returns (uint256 amountOut, uint256
       tokenInRefund) {
        //update accounting
        uint256 initialTokenIn = order.tokenIn.balanceOf(
           address(this));
        uint256 initialTokenOut = order.tokenOut.balanceOf(
           address(this));

        //approve
        order.tokenIn.safeApprove(target, order.amountIn);

        //perform the call
        (bool success, bytes memory reason) = target.call(
           txData);

        if (!success) {
            revert TransactionFailed(reason);
        }

        uint256 finalTokenIn = order.tokenIn.balanceOf(
           address(this));
        require(finalTokenIn >= initialTokenIn - order.
           amountIn, "over spend");
        uint256 finalTokenOut = order.tokenOut.balanceOf(
           address(this));

        require(
            finalTokenOut - initialTokenOut > order.
               minAmountOut,
```

```
            "Too Little Received"
        );

        amountOut = finalTokenOut - initialTokenOut;
        tokenInRefund = order.amountIn - (initialTokenIn -
            finalTokenIn);
    }
```

In the `execute` function of `OracleLess.sol`, the following code creates the root cause:

```
order.tokenIn.safeApprove(target, order.amountIn);
```

The `safeApprove` function from the OpenZeppelin library requires that the allowance for target on `order.tokenIn` must first be set to zero before being re-approved. However, in certain scenarios, leftover approvals from partial token usage during previous swaps (e.g., unspent tokenIn is refunded but allowance remains) will result in the next call to `safeApprove` reverting.

```
//refund any unspent tokenIn
//this should generally be 0 when using exact input for
    swaps, which is recommended
```

Any malicious user could perform `fillOrder` without fully utilizing the amount, managing to DoS that target, preventing it from being used for that token.

2.2.2 Proof of Concept

Consider the following scenario where:

1. A malicious user initiates a `fillOrder` to swap the tokens, which triggers the `execute` function. During this process, the function sets the token allowances for the target address.
2. The swap executes but does not utilize the entire allowance.
3. Next user attempts to fill an order using the same swap target.
4. Since the leftover allowance is non-zero, the `safeApprove` function call in the execute function will revert.

This vulnerability prevents the protocol from using the same target for any future swaps involving the same token.

2.2.3 Impact

A malicious user (or even a non-malicious user who is not aware of the issue) could execute a swap without fully utilizing the approved amount of tokens. This leftover allowance for the target address and token pair will prevent any future `fillOrder` attempts using the same target and token pair from succeeding, as the `safeApprove` function will fail due to the pre-existing non-zero allowance. This could disrupt the expected behavior of the protocol and prevent users from executing swaps correctly. Furthermore, a malicious user could exploit this by targeting any swap target they choose, making it possible to disrupt the protocol's operations across multiple targets and token pairs, leading to failed transactions and potential losses of funds.

2.2.4 Recommendations

Consider updating the code to reset the allowance to zero before setting a new allowance.

```
order.tokenIn.safeApprove(target, 0);
order.tokenIn.safeApprove(target, order.amountIn);
```

This ensures compatibility with the `safeApprove` function's requirements and prevents issues caused by leftover allowances.

3 Conclusion

The audit of the Oku protocol revealed two medium-severity vulnerabilities that could hinder its functionality and user experience. The first issue lies in the `currentValue` function of the `PythOracle` contract, where an incorrect staleness check causes stale prices to be accepted while rejecting fresh ones, leading to inaccurate calculations and potential user losses. The second vulnerability occurs in the `execute` function of the `OracleLess` contract, where leftover token allowances from previous swaps prevent reuse of the same swap target, enabling denial-of-service attacks on critical functionalities.

By addressing the identified vulnerabilities through the implementation of correct staleness validation and proper management of token allowances, the Oku protocol can significantly enhance its security and operational reliability. These improvements will ensure accurate price calculations, prevent disruptions in swap functionality, and maintain a seamless trading experience for its users. With a continued commitment to diligence and best practices, Oku can strengthen its position as a secure and efficient DeFi platform.