Findings for Oku Finance Audit Contest

https://audits.sherlock.xyz/contests/641

High: Reentrancy in `fillOrder` Function

https://github.com/sherlock-audit/2024-11-oku-judging/issues/222

Summary

The `fillOrder` function in `OracleLess.sol` is vulnerable to a reentrancy attack, which may result in the loss of user funds. An attacker can exploit the contract's token transfer mechanism due to the absence of proper reentrancy protections, risking significant financial losses for protocol users.

Root Cause

The root cause of the vulnerability lies in the following:

1. The `fillOrder` function lacks a `nonReentrant` modifier, despite the contract inheriting from `ReentrancyGuard`.
2. The function does not follow the Checks-Effects-Interactions (CEI) pattern, making external calls (e.g., token transfers via `execute`) before updating the internal state (e.g., `pendingOrderIds`).
3. The external calls in `execute` allow attackers to re-enter the function before state changes are finalized, creating opportunities for exploitation.

Internal Pre-conditions

1. An existing order must be present in the `pendingOrderIds` array.
2. The attacker must prepare a malicious contract capable of executing a reentrancy attack.
3. The contract's state must permit token transfer manipulation.
4. The order's tokens must have sufficient balance to make the attack economically viable.

External Pre-conditions

1. Network gas prices must support complex transaction execution.
2. The attacker must have sufficient cryptocurrency to initiate the attack.
3. The target blockchain must support the necessary contract interactions.

Attack Path

1. The attacker deploys a malicious contract with a carefully crafted fallback function.
2. The attacker initiates a legitimate order fill through the `fillOrder` function.
3. During the external call in the `execute` method, the attacker's contract triggers its fallback function.
4. The fallback function recursively calls `fillOrder` before the original function completes its state changes.
5. By re-entering the function, the attacker manipulates token accounting and extracts additional funds.
6. The attack exploits the fact that token transfers occur after the external call, creating a window for manipulation.

Impact

The protocol risks significant financial loss. An attacker could potentially:

1. Drain multiple orders simultaneously.
2. Create inconsistent states in token accounting.
3. Manipulate the contract's balance and transfer mechanisms.
4. Steal funds from other users' pending orders.

This vulnerability is critical as it exists in a core function responsible for order execution and token management.

Mitigation

To address this vulnerability, the following measures are recommended:

1. **Apply the `nonReentrant` modifier** to the `fillOrder` function.
2. **Reorder the function logic** to:
    - Perform all state changes before external calls.
    - Ensure the contract's state is updated before token transfers.
3. **Follow the Checks-Effects-Interactions (CEI) pattern** to prevent reentrancy.
4. Implement additional safeguards, such as:
    - Tracking and limiting reentrancy attempts.
    - Adding explicit reentrancy guards beyond the inherited `ReentrancyGuard`.
5. Conduct thorough testing to verify the effectiveness of reentrancy protections.

Suggested Implementation

Below is a recommended implementation that incorporates reentrancy protection and the CEI pattern:

```
function fillOrder(uint96 pendingOrderIdx, uint96 orderId, address target, bytes calldata txData)
    external
    override
    nonReentrant  // Add this modifier
{
    // Perform all state changes first
    // Remove order from pending orders
    pendingOrderIds = ArrayMutation.removeFromArray(pendingOrderIdx, pendingOrderIds);

    // Execute order with external call
    (uint256 amountOut, uint256 tokenInRefund) = execute(target, txData, order);

    // Perform token transfers after all state changes
    // ... (rest of the existing token transfer logic)
}
```

High: Predictable `orderId` Generation in `generateOrderId` Function

https://github.com/sherlock-audit/2024-11-oku-judging/issues/61

Summary

The use of `block.timestamp` in the `generateOrderId` function makes order IDs predictable. Attackers can exploit this predictability to calculate upcoming `orderId` values and disrupt or front-run legitimate order placements, potentially undermining user trust in the protocol.

Root Cause

The root cause of this vulnerability is the reliance on `block.timestamp` as the sole entropy source in the `generateOrderId` function:

- In `AutomationMaster.sol:92`, `generateOrderId` uses `block.timestamp` to create unique identifiers.
- `block.timestamp` is vulnerable to miner manipulation, as miners can adjust it within a ±15-second range, making the generated `orderId` predictable.

Internal Pre-conditions

1. A user interacts with the `generateOrderId` function to receive a new `orderId`.
2. The current `block.timestamp` must be accessible to the attacker.

External Pre-conditions

1. Miners can adjust `block.timestamp` within a ±15-second range.

Attack Path

1. The attacker calls `generateOrderId()` and observes the generated order ID.
2. The attacker manipulates `block.timestamp` by a few seconds and calls `generateOrderId()` again.
3. The attacker predicts future order IDs and potentially front-runs legitimate orders or places malicious orders to disrupt order placement.

Impact

The predictable nature of `orderId` generation results in:

1. A loss of trust from protocol users.
2. Potential exploitation of timing-sensitive order processing by attackers.

While the attacker may not gain direct financial benefits, the predictability undermines the integrity of the protocol.

Mitigation

To enhance randomness and prevent timestamp manipulation, implement one of the following strategies:

Option 1: Enhanced Entropy

Introduce additional sources of entropy in `generateOrderId`:

```
function generateOrderId(address sender) external view returns (uint96) {
    uint256 hashedValue = uint256(
        keccak256(
            abi.encodePacked(
                sender,
                block.timestamp,
                block.prevrandao,  // Additional entropy source
                address(this)      // Contract address for further randomness
            )
        )
    );
    return uint96(hashedValue);
}
```

Option 2: Nonce-based Approach

Use a nonce to ensure unique `orderId` values:

```
mapping(address => uint256) private _nonces;

function generateOrderId(address sender) external returns (uint96) {
    uint256 nonce = _nonces[sender]++;
    uint256 hashedValue = uint256(
        keccak256(
            abi.encodePacked(
                sender,
                nonce,
                block.timestamp,
                block.prevrandao
            )
        )
    );
    return uint96(hashedValue);
}
```

Alternative Recommendations

1. **Integrate a Verifiable Random Function (VRF):** Use Chainlink VRF or similar solutions to introduce secure randomness.
2. **Use external randomness oracles:** Leverage trusted third-party randomness oracles.
3. **Implement a commit-reveal scheme:** Require users to commit to a value and later reveal it to complete the order ID generation process.
4. **Add additional entropy sources:** Incorporate elements such as `block.number` or a unique identifier associated with the contract's state.
5. **Conduct thorough testing:** Ensure the new randomness mechanism is robust against potential attacks.

Medium: Incorrect Stale Price Check in `currentValue` Function

https://github.com/sherlock-audit/2024-11-oku-judging/issues/224

Summary

An incorrect stale price check in `PythOracle.sol` causes improper price validation. The contract may reject valid prices and accept stale prices, resulting in potential financial losses for users.

Root Cause

The vulnerability lies in the incorrect condition for stale price validation in the `currentValue` function:

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

- The condition incorrectly checks if the price is older than the allowed age (`noOlderThan`).
- The correct logic should ensure that the price is **newer** than the allowed age.

Internal Pre-conditions

1. The contract must have a valid price feed from the Pyth oracle.
2. The `currentValue()` function must be invoked to retrieve the price.

External Pre-conditions

1. The price feed from the Pyth oracle must be updated within the allowed age (`noOlderThan`).

Attack Path

1. The `currentValue()` function is called to retrieve the current price.
2. The contract checks the price using the incorrect condition:

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

3. As a result:
    - The contract incorrectly rejects valid prices that are newer than the allowed age.
    - The contract incorrectly accepts stale prices that are older than the allowed age.

Impact

1. **Rejection of Valid Prices:** Users may face rejected transactions due to improperly flagged prices.
2. **Acceptance of Stale Prices:** Stale prices may be treated as valid, leading to incorrect price calculations and potential financial losses.

Mitigation

To resolve this issue, update the condition in the `currentValue` function to properly validate stale prices:

Corrected Code:

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

- **Logic Update:** Ensure that the price is newer than the allowed age by checking that `block.timestamp - price.publishTime` is less than or equal to `noOlderThan` .