SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	GFX
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	July 4 - July 8, 2023
Prepared on:	October 6, 2023

Introduction

Software for the next generation of finance GFX Labs is a multi-faceted blockchain research and development company.

Scope

Repository: crispymangoes/uniswap-v3-limit-orders

Branch: main

Commit: 753974a4ba5b07ec076c5e5bcbe7277e5921be4b

For the detailed scope, see the contest details.

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

Medium	High
3	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90	kutugu	ginlee
mstpr-brainbot	trachev	MohammedRizwan
Dug	mahyar	Breeje
p0wd3r	Avci	XDZIBEC

Issue H-1: Lack of segregation between users' assets and collected fees resulting in loss of funds for the users

Source: https://github.com/sherlock-audit/2023-06-gfx-judging/issues/48

Found by

Dug, kutugu, mahyar, mstpr-brainbot, trachev, xiaoming90

Summary

The users' assets are wrongly sent to the owner due to a lack of segregation between users' assets and collected fees, which might result in an irreversible loss of assets for the victims.

Vulnerability Detail

GLX uses the Chainlink Automation to execute the `LimitOrderRegistry.performUpkeep` function when there are orders that need to be fulfilled. The `LimitOrderRegistry` contract must be funded with LINK tokens to keep the operation running.

To ensure the LINK tokens are continuously replenished and funded, users must pay a fee denominated in Native ETH or ERC20 WETH tokens on orders claiming as shown below. The collected ETH fee will be stored within the `LimitOrderRegistry` contract.

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L696

```
File: LimitOrderRegistry.sol
696:     function claimOrder(uint128 batchId, address user) external payable
 ↪  returns (ERC20, uint256) {
..SNIP..
723:         // Transfer tokens owed to user.
724:         tokenOut.safeTransfer(user, owed);
725:
726:         // Transfer fee in.
727:         address sender = _msgSender();
728:         if (msg.value >= userClaim.feePerUser) {
729:             // refund if necessary.
730:             uint256 refund = msg.value - userClaim.feePerUser;
731:             if (refund > 0) sender.safeTransferETH(refund);
732:         } else {
733:             WRAPPED_NATIVE.safeTransferFrom(sender, address(this),
 ↪  userClaim.feePerUser);
734:            // If value is non zero send it back to caller.
735:            if (msg.value > 0) sender.safeTransferETH(msg.value);
736:        }
..SNIP..
```

To retrieve the ETH fee collected, the owner will call the `LimitOrderRegistry.withdrawNative` function that will send all the Native ETH and ERC20 WETH tokens within the `LimitOrderRegistry` contract to the owner's address. After executing this function, the Native ETH and ERC20 WETH tokens on this contract will be zero and wiped out.

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L505

```
File: LimitOrderRegistry.sol
505:     function withdrawNative() external onlyOwner {
506:         uint256 wrappedNativeBalance =
 ↪  WRAPPED_NATIVE.balanceOf(address(this));
507:         uint256 nativeBalance = address(this).balance;
508:         // Make sure there is something to withdraw.
509:         if (wrappedNativeBalance == 0 && nativeBalance == 0) revert
 ↪  LimitOrderRegistry__ZeroNativeBalance();
510:
511:         // transfer wrappedNativeBalance if it exists
512:         if (wrappedNativeBalance > 0) WRAPPED_NATIVE.safeTransfer(owner,
 ↪  wrappedNativeBalance);
513:         // transfer nativeBalance if it exists
514:         if (nativeBalance > 0) owner.safeTransferETH(nativeBalance);
515:     }
```

Most owners will automate replenishing the `LimitOrderRegistry` contract with LINK tokens to ensure its balance does not fall below zero and for ease of maintenance. For instance, a certain percentage of the collected ETH fee (e.g., 50%) will be swapped immediately to LINK tokens on a DEX upon collection and transferred the swapped LINK tokens back to the `LimitOrderRegistry` contract. The remaining will be spent to cover operation and maintenance costs.

However, the issue is that there are many Uniswap V3 pools where their token pair consists of ETH/WETH. In fact, most large pools in Uniswap V3 will consist of ETH/WETH. For instance, the following Uniswap pools consist of ETH/WETH as one of the pool tokens:

- Assume that the owner has configured and setup the `LimitOrderRegistry` contract to work with the Uniswap DAI/ETH pool, and the current price of the DAI/ETH pool is 1,500 DAI/ETH.

  Bob submit a new Buy Limit Order swapping DAI to ETH at the price of 1,000 DAI/ETH. Bob would deposit 1,000,000 DAI to the `LimitOrderRegistry` contract.

When Bob's Buy Limit Order is ITM and fulfilled, 1000 ETH/WETH will be sent to and stored within the `LimitOrderRegistry` contract.

The next step that Bob must do to claim the swapped 1000 ETH/WETH is to call the `LimitOrderRegistry.claimOrder` function, which will collect the fee and transfer the swapped 1000 ETH/WETH to Bob.

Unfortunately, before Bob could claim his swapped ETH/WETH, the `LimitOrderRegistry.withdrawNative` function is triggered by the owner or the owner's bots. As noted earlier, when the `LimitOrderRegistry.withdrawNative` function is triggered, all the Native ETH and ERC20 WETH tokens on this contract will be transferred to the owner's address. As a result, Bob's 1000 swapped ETH/WETH stored within the `LimitOrderRegistry` contract are sent to the owner's address, and the balance of ETH/WETH in the `LimitOrderRegistry` contract is zero.

When Bob calls the `LimitOrderRegistry.claimOrder` function, the transaction will revert because insufficient ETH/WETH is left in the `LimitOrderRegistry` contract.

Unfortunately for Bob, there is no way to recover back his ETH/WETH that is sent to the owner's address. Following outline some of the possible scenarios where this could happen:

- The owners set up their infrastructure to automatically swap a portion or all the ETH/WETH received to LINK tokens and transfer them to the `LimitOrderRegistry` contract, and there is no way to retrieve the deposited LINK tokens from the `LimitOrderRegistry` contract even if the owner wishes to do so as there is no function within the contract to allow this action.
- The owners set up their infrastructure to automatically swap a small portion of ETH/WETH received to LINK tokens and send the rest of the ETH/WETH to 100 investors/DAO members' addresses. So, it is no guarantee that the investors/DAO members will return the ETH/WETH to Bob.

Impact

Loss of assets for the users

Code Snippet

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L505

Tool used

Manual Review

Recommendation

Consider implementing one of the following solutions to mitigate the issue:

Solution 1 - Only accept Native ETH as fee Uniswap V3 pool stored ETH as Wrapped ETH (WETH) ERC20 token internally. When the `collect` function is called against the pool, WETH ERC20 tokens are returned to the caller. Thus, the most straightforward way to mitigate this issue is to update the contract to collect the fee in Native ETH only.

In this case, there will be a clear segregation between users' assets (WETH) and owner's fee (Native ETH)

```
function withdrawNative() external onlyOwner {
-    uint256 wrappedNativeBalance = WRAPPED_NATIVE.balanceOf(address(this));
    uint256 nativeBalance = address(this).balance;
    // Make sure there is something to withdraw.
-    if (wrappedNativeBalance == 0 && nativeBalance == 0) revert
 ↪  LimitOrderRegistry__ZeroNativeBalance();
+    if (nativeBalance == 0) revert LimitOrderRegistry__ZeroNativeBalance();

-    // transfer wrappedNativeBalance if it exists
-    if (wrappedNativeBalance > 0) WRAPPED_NATIVE.safeTransfer(owner,
 ↪  wrappedNativeBalance);
    // transfer nativeBalance if it exists
    if (nativeBalance > 0) owner.safeTransferETH(nativeBalance);
}
```

```
function claimOrder(uint128 batchId, address user) external payable returns
 ↪  (ERC20, uint256) {
..SNIP..
    // Transfer tokens owed to user.
    tokenOut.safeTransfer(user, owed);

    // Transfer fee in.
    address sender = _msgSender();
    if (msg.value >= userClaim.feePerUser) {
        // refund if necessary.
        uint256 refund = msg.value - userClaim.feePerUser;
        if (refund > 0) sender.safeTransferETH(refund);
    } else {
-        WRAPPED_NATIVE.safeTransferFrom(sender, address(this),
 ↪  userClaim.feePerUser);
-       // If value is non zero send it back to caller.
-       if (msg.value > 0) sender.safeTransferETH(msg.value);
+       revert LimitOrderRegistry__InsufficientFee;
    }
..SNIP..
```

Solution 2 - Define state variables to keep track of the collected fee
Consider defining state variables to keep track of the collected fee so that the fee will not mix up with users' assets.

```
function claimOrder(uint128 batchId, address user) external payable returns
↪   (ERC20, uint256) {
..SNIP..
    // Transfer fee in.
    address sender = _msgSender();
    if (msg.value >= userClaim.feePerUser) {
+       collectedNativeETHFee += userClaim.feePerUser
        // refund if necessary.
        uint256 refund = msg.value - userClaim.feePerUser;
        if (refund > 0) sender.safeTransferETH(refund);
    } else {
+       collectedWETHFee += userClaim.feePerUser
        WRAPPED_NATIVE.safeTransferFrom(sender, address(this),
↪   userClaim.feePerUser);
        // If value is non zero send it back to caller.
        if (msg.value > 0) sender.safeTransferETH(msg.value);
    }
..SNIP..
```

```
function withdrawNative() external onlyOwner {
-   uint256 wrappedNativeBalance = WRAPPED_NATIVE.balanceOf(address(this));
-   uint256 nativeBalance = address(this).balance;
+   uint256 wrappedNativeBalance = collectedWETHFee;
+   uint256 nativeBalance = collectedNativeETHFee;
+   collectedWETHFee = 0; // clear the fee
+   collectedNativeETHFee = 0; // clear the fee
    // Make sure there is something to withdraw.
    if (wrappedNativeBalance == 0 && nativeBalance == 0) revert
↪   LimitOrderRegistry__ZeroNativeBalance();

    // transfer wrappedNativeBalance if it exists
    if (wrappedNativeBalance > 0) WRAPPED_NATIVE.safeTransfer(owner,
↪   wrappedNativeBalance);
    // transfer nativeBalance if it exists
    if (nativeBalance > 0) owner.safeTransferETH(nativeBalance);
}
```

Discussion

elee1766

I believe solution 2 is the better option here, but will have to consider more before implementation

elee1766

https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1

elee1766

fix

xiaoming9090

Verified. Fixed in https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1

Issue H-2: Users' funds could be stolen or locked by malicious or rouge owners

Source: https://github.com/sherlock-audit/2023-06-gfx-judging/issues/54

Found by

Dug, mstpr-brainbot, p0wd3r, xiaoming90

Summary

Users' funds could be stolen or locked by malicious or rouge owners.

Vulnerability Detail

In the contest's README, the following was mentioned.

> **Q: Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?**
>
> restricted. the owner should not be able to steal funds.

It was understood that the owner is not "trusted" and should not be able to steal funds. Thus, it is fair to assume that the sponsor is keen to know if there are vulnerabilities that could allow the owner to steal funds or, to a lesser extent, lock the user's funds.

Many control measures are implemented within the protocol to prevent the owner from stealing or locking the user's funds.

However, based on the review of the codebase, there are still some "loopholes" that the owner can exploit to steal funds or indirectly cause losses to the users. Following is a list of methods/tricks to do so.

Method 1 - Use the vulnerable `withdrawNative` function Once the user's order is fulfilled, the swapped ETH/WETH will be sent to the contract awaiting the user's claim. However, the owner can call the `withdrawNative` function, which will forward all the Native ETH and Wrapped ETH in the contract to the owner's address due to another bug ("Lack of segregation between users' assets and collected fees resulting in loss of funds for the users") that I highlighted in another of my report.

Method 2 - Add a malicious custom price feed
https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L482

```
File: LimitOrderRegistry.sol
482:     function setFastGasFeed(address feed) external onlyOwner {
483:         fastGasFeed = feed;
484:     }
```

The owner can create a malicious price feed contract and configure the `LimitOrderRegistry` to use it by calling the `setFastGasFeed` function.

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L914

```
File: LimitOrderRegistry.sol
914:     function performUpkeep(bytes calldata performData) external {
915:         (UniswapV3Pool pool, bool walkDirection, uint256 deadline) =
 ↪   abi.decode(
916:             performData,
917:             (UniswapV3Pool, bool, uint256)
918:         );
919:
920:         if (address(poolToData[pool].token0) == address(0)) revert
 ↪   LimitOrderRegistry__PoolNotSetup(address(pool));
921:
922:         PoolData storage data = poolToData[pool];
923:
924:         // Estimate gas cost.
925:         uint256 estimatedFee = uint256(upkeepGasLimit * getGasPrice());
```

When fulfilling an order, the `getGasPrice()` function will fetch the gas price from the malicious price feed that will report an extremely high price (e.g., 100000 ETH), causing the `estimatedFee` to be extremely high. When users attempt to claim the order, they will be forced to pay an outrageous fee, which the users cannot afford to do so. Thus, the users have to forfeit their orders, and they will lose their swapped tokens.

Impact

Users' funds could be stolen or locked by malicious or rouge owners.

Code Snippet

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L505

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L482

Tool used

Manual Review

Recommendation

Consider implementing the following measures to reduce the risk of malicious/rouge owners from stealing or locking the user's funds.

1) To mitigate the issue caused by the vulnerable `withdrawNative` function. Refer to my recommendation in my report titled "Lack of segregation between users' assets and collected fees resulting in loss of funds for the users".
2) To mitigate the issue of the owner adding a malicious custom price feed, consider performing some sanity checks against the value returned from the price feed. For instance, it should not be larger than the `MAX_GAS_PRICE` constant. If it is larger than `MAX_GAS_PRICE` constant, fallback to the user-defined gas feed, which is constrained to be less than `MAX_GAS_PRICE`.

Discussion

elee1766

this is a valid concern.

I think a sanity check on the gas variable makes a lot of sense.

Issue M-1: Owners will incur loss and bad debt if the value of a token crashes

Source: https://github.com/sherlock-audit/2023-06-gfx-judging/issues/51

Found by

xiaoming90

Summary

If the value of the swapped tokens crash, many users will choose not to claim the orders, which result in the owner being unable to recoup back the gas fee the owner has already paid for automating the fulfillment of the orders, incurring loss and bad debt.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L696

```
File: LimitOrderRegistry.sol
696:     function claimOrder(uint128 batchId, address user) external
 ↪  payable returns (ERC20, uint256) {
..SNIP..
726:         // Transfer fee in.
727:         address sender = _msgSender();
728:         if (msg.value >= userClaim.feePerUser) {
729:             // refund if necessary.
730:             uint256 refund = msg.value - userClaim.feePerUser;
731:             if (refund > 0) sender.safeTransferETH(refund);
732:         } else {
733:             WRAPPED_NATIVE.safeTransferFrom(sender, address(this),
 ↪  userClaim.feePerUser);
734:             // If value is non zero send it back to caller.
735:             if (msg.value > 0) sender.safeTransferETH(msg.value);
736:         }
```

Users only need to pay for the gas cost for fulfilling the order when they claim the order to retrieve the swapped tokens. When the order is fulfilled, the swapped tokens will be sent to and stored in the `LimitOrderRegistry` contract.

However, in the event that the value of the swapped tokens crash (e.g., Terra's LUNA crash), it makes more economic sense for the users to abandon (similar to defaulting in traditional finance) the orders without claiming the worthless tokens to avoid paying the more expensive fee to the owner.

As a result, many users will choose not to claim the orders, which result in the owner being unable to recoup back the gas fee the owner has already paid for automating the fulfillment of the orders, incurring loss and bad debt.

Impact

Owners might be unable to recoup back the gas fee the owner has already paid for automating the fulfillment of the orders, incurring loss and bad debt.

Code Snippet

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L696

Tool used

Manual Review

Recommendation

Consider collecting the fee in advance based on a rough estimation of the expected gas fee. When the users claim the order, any excess fee will be refunded, or any deficit will be collected from the users.

In this case, if many users choose to abandon the orders, the owner will not incur any significant losses.

Discussion

elee1766

this is a known issue, but i don't think we will fix it.

owners recognize that gas can possibly be "wasted" under bad network conditions

Issue M-2: Owner unable to collect fulfillment fee from certain users due to revert error

Source: https://github.com/sherlock-audit/2023-06-gfx-judging/issues/55

Found by

XDZIBEC, kutugu, mstpr-brainbot, xiaoming90

Summary

Certain users might not be able to call the `claimOrder` function under certain conditions, resulting in the owner being unable to collect fulfillment fees from the users.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L721

```
File: LimitOrderRegistry.sol
696:     function claimOrder(uint128 batchId, address user) external payable returns (ERC20, uint256) {
697:         Claim storage userClaim = claim[batchId];
698:         if (!userClaim.isReadyForClaim) revert LimitOrderRegistry__OrderNotReadyToClaim(batchId);
699:         uint256 depositAmount = batchIdToUserDepositAmount[batchId][user];
700:         if (depositAmount == 0) revert LimitOrderRegistry__UserNotFound(user, batchId);
701:
702:         // Zero out user balance.
703:         delete batchIdToUserDepositAmount[batchId][user];
704:
705:         // Calculate owed amount.
706:         uint256 totalTokenDeposited;
707:         uint256 totalTokenOut;
708:         ERC20 tokenOut;
709:
710:         // again, remembering that direction == true means that the input token is token0.
711:         if (userClaim.direction) {
712:             totalTokenDeposited = userClaim.token0Amount;
713:             totalTokenOut = userClaim.token1Amount;
714:             tokenOut = poolToData[userClaim.pool].token1;
715:         } else {
```

```
            totalTokenDeposited = userClaim.token1Amount;
            totalTokenOut = userClaim.token0Amount;
            tokenOut = poolToData[userClaim.pool].token0;
        }

        uint256 owed = (totalTokenOut * depositAmount) /
    ↪ totalTokenDeposited;

        // Transfer tokens owed to user.
        tokenOut.safeTransfer(user, owed);
```

Assume the following:

- SHIB has 18 decimals of precision, while USDC has 6.
- Alice (Small Trader) deposited 10 SHIB while Bob (Big Whale) deposited 100000000 SHIB.
- The batch order was fulfilled, and it claimed 9 USDC (`totalTokenOut`)

The following formula and code compute the number of swapped/claimed USDC tokens a user is entitled to.

```
owed = (totalTokenOut * depositAmount) / totalTokenDeposited
owed = (9 USDC * 10 SHIB) / 100000000 SHIB
owed = (9 * 10^6 * 10 * 10^18) / (100000000 * 10^18)
owed = (9 * 10^6 * 10) / (100000000)
owed = 90000000 / 100000000
owed = 0 USDC (Round down)
```

Based on the above assumptions and computation, Alice will receive zero tokens in return due to a rounding error in Solidity.

The issue will be aggravated under the following conditions:

- If the difference in the precision between `token0` and `token1` in the pool is larger
- The token is a stablecoin, which will attract a lot of liquidity within a small price range (e.g. `$0.95 ~ $1.05`)

Note: Some tokens have a low decimal of 2 (e.g., Gemini USD), while others have a high decimal of 24 (e.g., `YAM-V2` has 24). Refer to https://github.com/d-xo/weird-erc20#low-decimals

The rounding down to zero is unavoidable in this scenario due to how values are represented. It is not possible to send Alice 0.9 WEI of USDC. The smallest possible amount is 1 WEI.

In this case, it will attempt to transfer a zero amount of `tokenOut`, which might result in a revert as some tokens disallow the transfer of zero value. As a result, when users call the `claimOrder` function, it will revert, and the owner will not be able to collect the fulfillment fee from the users.

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L724

```
723:         // Transfer tokens owed to user.
724:         tokenOut.safeTransfer(user, owed);
```

Impact

When a user cannot call the `claimOrder` function due to the revert error, the owner will not be able to collect the fulfillment fee from the user, resulting in a loss of fee for the owner.

Code Snippet

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L724

Tool used

Manual Review

Recommendation

Consider only transferring the assets if the amount is more than zero.

```
uint256 owed = (totalTokenOut * depositAmount) / totalTokenDeposited;

// Transfer tokens owed to user.
- tokenOut.safeTransfer(user, owed);
+ if (owed > 0) tokenOut.safeTransfer(user, owed);
```

Discussion

elee1766

i agree - this change should be made.

elee1766

https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1

elee1766

fix

xiaoming9090

Verified. Fixed in https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1

Issue M-3: getGasPrice() doesn't check Arbitrum l2 chainlink feed is active

Source: https://github.com/sherlock-audit/2023-06-gfx-judging/issues/65

Found by

Avci, Breeje, MohammedRizwan, ginlee, kutugu

Summary

getGasPrice() doesn't check Arbitrum l2 chainlink feed is active

Vulnerability Detail

When utilizing Chainlink in L2 chains like Arbitrum, it's important to ensure that the prices provided are not falsely perceived as fresh, even when the sequencer is down. This vulnerability could potentially be exploited by malicious actors to gain an unfair advantage.

If the sequencer is down, messages cannot be transmitted from L1 to L2, and no L2 transactions are executed. Instead, messages are enqueued in the CanonicalTransactionChain on L1

On the L1 network:

1.A network of node operators runs the external adapter to post the latest sequencer status to the AggregatorProxy contract and relays the status to the Aggregator contract. The Aggregator contract calls the validate function in the OptimismValidator contract.

2.The OptimismValidator contract calls the sendMessage function in the L1CrossDomainMessenger contract. This message contains instructions to call the updateStatus(bool status, uint64 timestamp) function in the sequencer uptime feed deployed on the L2 network.

3.The L1CrossDomainMessenger contract calls the enqueue function to enqueue a new message to the CanonicalTransactionChain.

4.The Sequencer processes the transaction enqueued in the CanonicalTransactionChain contract to send it to the L2 contract.

On the L2 network:

1.The Sequencer posts the message to the L2CrossDomainMessenger contract.

2.The L2CrossDomainMessenger contract relays the message to the OptimismSequencerUptimeFeed contract.

3.The message relayed by the L2CrossDomainMessenger contains instructions to call updateStatus in the OptimismSequencerUptimeFeed contract.

4.Consumers can then read from the AggregatorProxy contract, which fetches the latest round data from the OptimismSequencerUptimeFeed contract.

Impact

could potentially be exploited by malicious actors to gain an unfair advantage.

Code Snippet

```
function getGasPrice() public view returns (uint256) {
       // If gas feed is set use it.
       if (fastGasFeed != address(0)) {
           (, int256 _answer, , uint256 _timestamp, ) =
 ↪  IChainlinkAggregator(fastGasFeed).latestRoundData();
           uint256 timeSinceLastUpdate = block.timestamp - _timestamp;
           // Check answer is not stale.
           if (timeSinceLastUpdate > FAST_GAS_HEARTBEAT) {
               // If answer is stale use owner set value.
               // Multiply by 1e9 to convert gas price to gwei
               return uint256(upkeepGasPrice) * 1e9;
           } else {
               // Else use the datafeed value.
               uint256 answer = uint256(_answer);
               return answer;
           }
       }
       // Else use owner set value.
       return uint256(upkeepGasPrice) * 1e9; // Multiply by 1e9 to convert
 ↪  gas price to gwei
   }
```

https://github.com/sherlock-audit/2023-06-gfx/blob/main/uniswap-v3-limit-orders/src/LimitOrderRegistry.sol#L1447-L1465

Tool used

Manual Review

Recommendation

The recommendation is to implement a check for the sequencer in the L2 version of the contract, and a code example of Chainlink can be found at https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code.

CHAINLINK DOC :
https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

Same issue in other contests

https://github.com/sherlock-audit/2023-02-bond-judging/issues/1

https://github.com/sherlock-audit/2022-11-sentiment-judging/issues/3

https://github.com/sherlock-audit/2023-01-sentiment-judging/issues/16

Discussion

elee1766

https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1

elee1766

fix

xiaoming9090

Verified. Fixed in https://github.com/gfx-labs/uniswap-v3-limit-orders/pull/1