xOracle

Smart Contract Audit Report
Prepared for Crown Labs

Date Issued:	Nov 10, 2023
Project ID:	AUDIT2023019
Version:	v1.0
Confidentiality Level:	Public

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2023019
Version	v1.0
Client	Crown Labs
Project	xOracle
Auditor(s)	Wachirawit Kanpanluk Phitchakorn Apiratisakul
Author(s)	Wachirawit Kanpanluk
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Nov 10, 2023	Full report	Wachirawit Kanpanluk

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Crown Labs, Inspex team conducted an audit to verify the security posture of the xOracle smart contracts between Oct 31, 2023 and Nov 1, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of xOracle smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 1 high, 1 medium, and 1 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved in the reassessment. Therefore, Inspex trusts that xOracle smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

The xOracle is a decentralized oracle price feed data solution for blockchain ecosystems.

An Oracle Price Feed is a mechanism for providing real-time, accurate pricing data within a blockchain ecosystem. By utilizing a network of independent data providers, known as "oracles", an Oracle Pricefeed ensures that the pricing data is accurate and tamper-proof. Smart contracts are used to define the rules and incentives for data providers, and handle the distribution of rewards for accurate and timely data submissions.

Scope Information:

Project Name	xOracle
Website	https://github.com/Crown-Labs
Smart Contract Type	Ethereum Smart Contract
Chain	Linea
Programming Language	Solidity
Category	Oracle

Audit Information:

Audit Method	Whitebox
Audit Date	Oct 31, 2023 - Nov 1, 2023
Reassessment Date	Nov 8, 2023

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: c8325819500196c4292b603981ae50f1582f7ca8)

Contract	Location (URL)
XOracle	https://github.com/Crown-Labs/xoracle-contracts/blob/c832581950/contracts/xOracle.sol
PriceFeedStore	https://github.com/Crown-Labs/xoracle-contracts/blob/c832581950/contracts/PriceFeedStore.sol

Reassessment: (Commit: 878e39fcbf6d32aab9b2274635027a658beb50bc)

Contract	Location (URL)
XOracle	https://github.com/Crown-Labs/xoracle-contracts/blob/878e39fcbf/contracts/xOracle.sol
PriceFeedStore	https://github.com/Crown-Labs/xoracle-contracts/blob/878e39fcbf/contracts/PriceFeedStore.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

2.3. Security Model

2.3.1 Trust Modules

The xOracle has privileged roles with the authority to mutate the critical state variables of the contract. Changes to these state variables significantly impact the contract's functionality. The privileged roles and their corresponding privileged functions are enumerated as follows:

- The proxy admin address can upgrade the contract implementation; this privileged operation could be used to drain all users' approved funds from the users' contract.
- The owner address, which is a multisig contract, can perform the following actions:
  - Pause/unpause the contract functionality; when paused `requestPrices()` and `cancelRequestPrice()` functions in the contract will be unusable.
  - Setting important states, such as the controller and signer addresses who provide the price feed data to the contract, the whitelist flag, and the platform fee, is a privileged operation that could be used to include a malicious signer to manipulate prices in a favorable manner.
- The controller address can update the price to fulfill the user's request via the `fulfillRequest()` and `refundRequest()` functions; this privileged operation could be used to drain the user's approved fund.

The xOracle several functionalities have relied on the external components, which may significantly impact the contract if they malfunction. The external components are listed as follows:

- The xOracle chain, which provides asset prices to the contract, consists of the following components:
  - The price feed node, also known as the signer, is the node responsible for updating the price in the xOracle chain, which is used to fulfill user requests. An incorrect asset price can result in financial losses for platform users and undermine the platform's authority.
  - The xOracle API, which is the node that aggregates the signed price feed data and forwards it to fulfill user requests. Improper behavior can lead to the failure of the request fulfillment flow.

2.3.2 Trust Assumptions

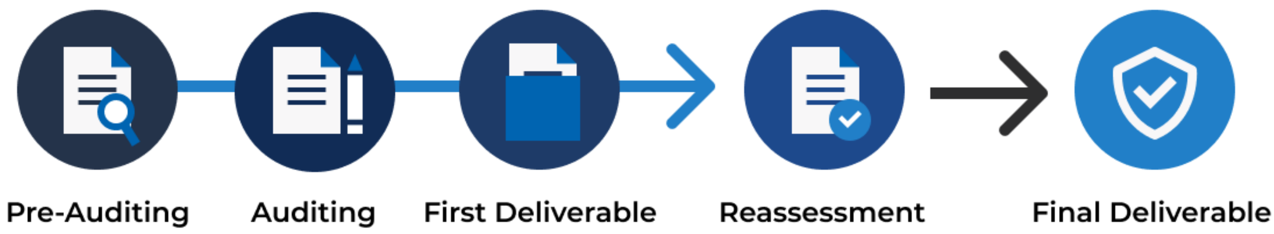
In the xOracle, the protocol's privileged roles, include the ability to change the critical state variables of the contract, also external components are assumed to be trusted. Acknowledging these trust assumptions is important, as it introduces substantial risks to the platform. Trust assumptions include, but are not limited to:

- The following privileged roles perform the privileged function with good will:
  - The proxy admin address is trusted to upgrade the contract implementation.
  - The owner address is entrusted with the responsibility of consistently configuring the contract's states correctly.
- It is assumed that the external module, specifically the xOracle chain, consistently provides accurate pricing and prepares data as part of its regular operations at all times.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at (https://docs.inspex.co/smart-contract-security-testing-guide/).

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

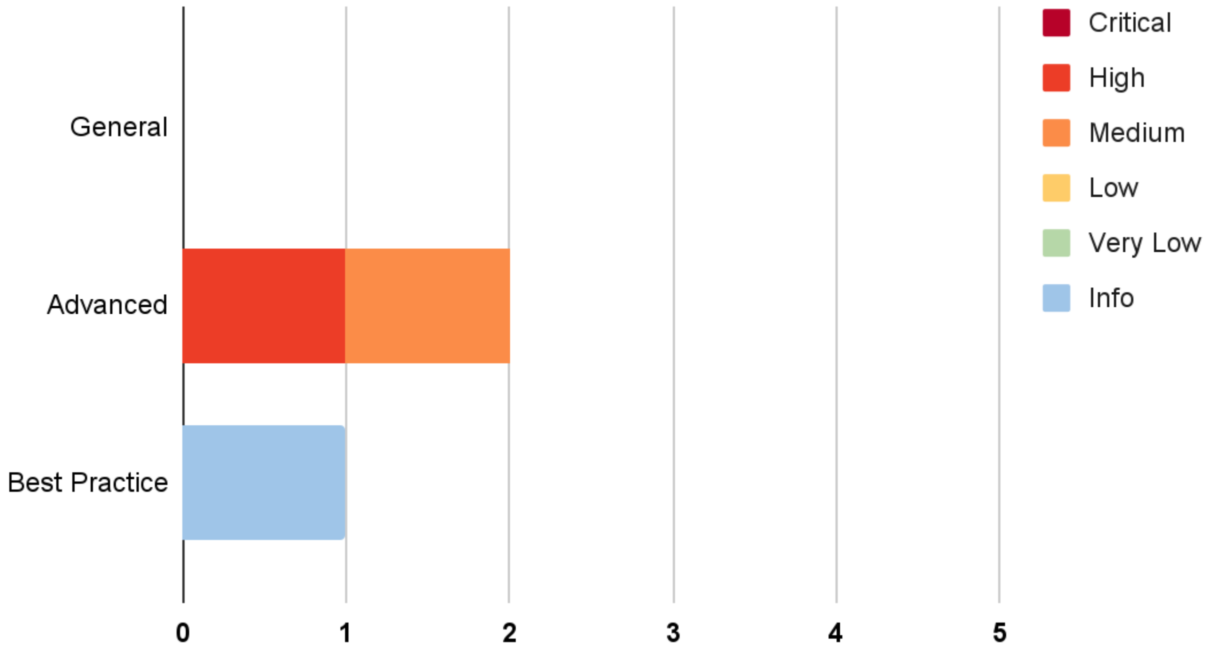
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

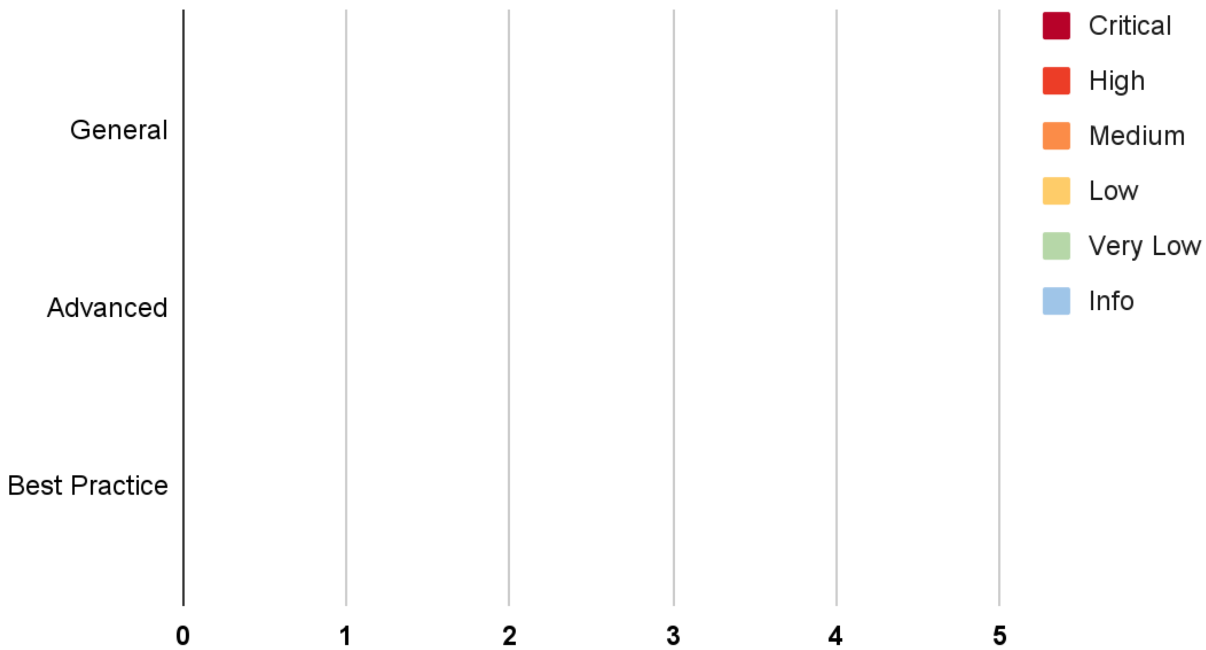
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Draining of User's Approved Funds	Advanced	High	Resolved
IDX-002	Gas Griefing in xOracle Contract	Advanced	Medium	Resolved
IDX-003	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved

* The mitigations or clarifications by Crown Labs can be found in Chapter 5.

5. Detailed Findings Information

5.1. Draining of User's Approved Funds

ID	IDX-001
Target	XOracle
Category	Advanced Smart Contract Vulnerability,
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** **High** **Impact:** **High** The controller address can drain the user's approved amount by increasing the `tx.gasPrice` value in a transaction and calling `fulfillRequest()` with the victim's request ID. **Likelihood:** **Medium** Only the controller address that was set by the owner, can take advantage by setting `tx.gasPrice` in the transaction.
Status	**Resolved** The Crown Labs team has resolved this issue by modifying the request price flow to let the user deposit the gas usage to the contract instead of approve it to the contract and define the acceptable gas price by the user as suggested in commit `878e39fcbf6d32aab9b2274635027a658beb50bc`.

5.1.1. Description

In the contract `XOracle`, the controller address, which is set by the contract owner, can call the `fulfillRequest()` function to fulfill the request with the price data and then collect a fee from the requester.

xOracle.sol

```
function fulfillRequest(Data[] memory _data, uint256 _reqId) external onlyController {
    Request storage request = requests[_reqId];
    if (request.status != 0) {
        return;
    }
    // set status executed
    request.status = 1;

    require(request.owner != address(0), "request not found");
    require(request.expiration > block.timestamp, "request is expired");
```

```
    // capture gas used
    uint256 gasStart = gasleft();

    // set price
    (bool priceUpdate, string memory message) = setPrices(request.timestamp, _data);

    // callback
    xOracleCallback(request.owner, _reqId, priceUpdate, request.payload);

    // payment request fee
    transferRequestFee(_reqId, request.owner, gasStart - gasleft());

    emit FulfillRequest(_reqId, priceUpdate, message);
}
```

However, the `transferRequestFee()` function uses the `transferFrom()` function to directly transfer the `reqFee` amount from the user to the controller address. The `reqFee` value is calculated from `tx.gasprice * _gasUsed` then added by the fulfill fee, depending on `fullfillFee`, as shown in line 392.

Since the calculation relies on the `tx.gasPrice` of the transaction, which could be controlled by the controller, this could be profitable for the controller. Therefore, the controller could possibly act maliciously to drain the user's approved amount by setting `tx.gasPrice` as they wish.

xOracle.sol

```
386 function transferRequestFee(uint256 _reqId, address _from, uint256 _gasUsed) private {
387     if (fulfillFee == 0) {
388         return;
389     }
390
391     // calculate req fee
392     uint256 reqFee = (tx.gasprice * _gasUsed * (FULFILL_FEE_PRECISION + fulfillFee)) / FULFILL_FEE_PRECISION;
393     IERC20(weth).transferFrom(_from, msg.sender, reqFee);
394
395     emit TransferRequestFee(_reqId, _from, msg.sender, reqFee);
396 }
```

5.1.2. Remediation

Inspex suggests allowing users to define a gas price limit by introducing the `maxGasPrice` parameter when creating a request as follows:

In line 97, add the `maxGasPrice` to the `Request` struct.

xOracle.sol

```
struct Request {
    uint256 timestamp;
    address owner;
    bytes payload;
    uint256 status; // 0 = request,  1 = fulfilled, 2 = cancel, 3 = refund
    uint256 expiration;
    uint256 maxGasPrice;
}
```

In the `requestPrices()` function, receive the new parameter `_maxGasPrice` and add it to the requests state, and validate it with the new constant `MIN_GAS_PRICE` that could be defined by platform, as you can see in lines 98 and 114.

xOracle.sol

```
91  function requestPrices(bytes memory _payload, uint256 _expiration, uint256
    _maxGasPrice) external onlyContract whenNotPaused returns (uint256) {
92      // check allow all or only whitelist
93      require(!onlyWhitelist || whitelists[msg.sender], "whitelist: forbidden");
94
95      // check request fee balance
96      require(paymentAvailable(msg.sender), "insufficient request fee");
97
98      require(_maxGasPrice >= MIN_GAS_PRICE, "Gas price is too low");
99
100     reqId++;
101
102     // default expire time
103     if (_expiration < block.timestamp + MINIMUM_EXPIRE_TIME) {
104         _expiration = block.timestamp + DEFAULT_EXPIRE_TIME;
105     }
106
107     // add request
108     requests[reqId] = Request({
109         timestamp: markdown(block.timestamp),
110         owner: msg.sender,
111         payload: _payload,
112         status: 0, // set status request
113         expiration: _expiration,
114         maxGasPrice: _maxGasPrice
115     });
116
117     emit RequestPrices(reqId);
118     return reqId;
119 }
```

When the controller calls the `fulfillRequest()` function, the `request.maxGasprice` must be added to the `transferRequestFee()` function parameter, as shown in line 153.

xOracle.sol

```
132 function fulfillRequest(Data[] memory _data, uint256 _reqId) external onlyController {
133     Request storage request = requests[_reqId];
134     if (request.status != 0) {
135         return;
136     }
137     // set status executed
138     request.status = 1;
139
140     require(request.owner != address(0), "request not found");
141     require(request.expiration > block.timestamp, "request is expired");
142
143     // capture gas used
144     uint256 gasStart = gasleft();
145
146     // set price
147     (bool priceUpdate, string memory message) = setPrices(request.timestamp, _data);
148
149     // callback
150     xOracleCallback(request.owner, _reqId, priceUpdate, request.payload);
151
152     // payment request fee
153     transferRequestFee(_reqId, request.owner, gasStart - gasleft(), request.maxGasprice);
154
155     emit FulfillRequest(_reqId, priceUpdate, message);
156 }
```

Finally, add the `require` statement to check `tx.gasprice` and `_maxGasPrice` in line 391.

xOracle.sol

```
386 function transferRequestFee(uint256 _reqId, address _from, uint256 _gasUsed, uint256 _maxGasPrice) private {
387     if (fulfillFee == 0) {
388         return;
389     }
390
391     require(tx.gasprice <= _maxGasPrice, "tx.gasprice exceeds maxgasprice");
392
393     // calculate req fee
394     uint256 reqFee = (tx.gasprice * _gasUsed * (FULFILL_FEE_PRECISION +
```

```
fulfillFee)) / FULFILL_FEE_PRECISION;
    IERC20(weth).transferFrom(_from, msg.sender, reqFee);

    emit TransferRequestFee(_reqId, _from, msg.sender, reqFee);
}
```

Please note that the remediations for other issues are not yet applied in the examples above.

5.2. Gas Griefing in XOracle Contract

ID	IDX-002
Target	XOracle
Category	Advanced Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Medium **Impact:** Medium The waste of execution gas costs results from the reverting of the `fulfillRequest()` function. **Likelihood:** Medium Anyone can call the `requestPrices()` function to abuse the `fulfillRequest()` function, if the whitelisted flag is set to false.
Status	Resolved The Crown Labs team has resolved this issue by modifying the request price and fulfillment flow as suggested in commit `878e39fcbf6d32aab9b2274635027a658beb50bc`.

5.2.1. Description

Users can request the latest price from the platform by providing a callback contract that contains the amount of gas cost via the `requestPrices()` function.

xOracle.sol

```
function requestPrices(bytes memory _payload, uint256 _expiration) external
onlyContract whenNotPaused returns (uint256) {
    // check allow all or only whitelist
    require(!onlyWhitelist || whitelists[msg.sender], "whitelist: forbidden");

    // check request fee balance
    require(paymentAvailable(msg.sender), "insufficient request fee");

    reqId++;

    // default expire time
    if (_expiration < block.timestamp + MINIMUM_EXPIRE_TIME) {
        _expiration = block.timestamp + DEFAULT_EXPIRE_TIME;
    }

    // add request
    requests[reqId] = Request({
        timestamp: markdown(block.timestamp),
```

```
        owner: msg.sender,
        payload: _payload,
        status: 0, // set status request
        expiration: _expiration
    });

    emit RequestPrices(reqId);
    return reqId;
}
```

The platform then updates the latest price and calls back to the user's contract via the `fulfillRequest()` function.

xOracle.sol

```
132 function fulfillRequest(Data[] memory _data, uint256 _reqId) external
onlyController {
133     Request storage request = requests[_reqId];
134     if (request.status != 0) {
135         return;
136     }
137     // set status executed
138     request.status = 1;
139
140     require(request.owner != address(0), "request not found");
141     require(request.expiration > block.timestamp, "request is expired");
142
143     // capture gas used
144     uint256 gasStart = gasleft();
145
146     // set price
147     (bool priceUpdate, string memory message) = setPrices(request.timestamp,
_data);
148
149     // callback
150     xOracleCallback(request.owner, _reqId, priceUpdate, request.payload);
151
152     // payment request fee
153     transferRequestFee(_reqId, request.owner, gasStart - gasleft());
154
155     emit FulfillRequest(_reqId, priceUpdate, message);
156 }
```

However, the user can implement the callback function that uses a high gas cost and transfers tokens out of the contract to trigger a revert in the `fulfillRequest()` function at line 153. This allows users to abuse the fulfillment process and drain the execution gas cost from the executor.

5.2.2. Remediation

Inspex suggests transferring the fulfillment fee, which is calculated by the user's supply parameters named `_callbackGasLimit` and `_maxGasPrice` (from the IDX-001 remediation), when the `requestPrices()` function is called and adding the `fulfillFee` value to the struct for the calculation when the `fulfillRequest()` function is called, as shown below in lines 91, 96, 99, 104 and 118 - 121.

xOracle.sol

```
91  function requestPrices(bytes memory _payload, uint256 _expiration, uint256 _maxGasPrice, uint256 _callbackGasLimit) external onlyContract whenNotPaused returns (uint256) {
92      // check allow all or only whitelist
93      require(!onlyWhitelist || whitelists[msg.sender], "whitelist: forbidden");
94  
95      // check gas price
96      require(_maxGasPrice >= minGasPrice, "gas price is too low");
97  
98      // check mininum gas limit
99      require(_callbackGasLimit > minGasLimit, "gas limit is too low");
100 
101     reqId++;
102 
103     // deposit request fee
104     uint256 reqFee = transferRequestFee(reqId, msg.sender, address(this), _callbackGasLimit, _maxGasPrice, fulfillFee);
105 
106     // default expire time
107     if (_expiration < block.timestamp + MINIMUM_EXPIRE_TIME) {
108         _expiration = block.timestamp + DEFAULT_EXPIRE_TIME;
109     }
110 
111     // add request
112     requests[reqId] = Request({
113         timestamp: markdown(block.timestamp),
114         owner: msg.sender,
115         payload: _payload,
116         status: 0, // set status request
117         expiration: _expiration,
118         maxGasPrice: _maxGasPrice,
119         callbackGasLimit: _callbackGasLimit,
120         depositReqFee: reqFee,
121         fulfillFee: fulfillFee
122     });
123 
124     emit RequestPrices(reqId);
125     return reqId;
126 }
```

And modifying the `transferRequestFee()` function to handle the transferring of the fund and using the `_fulfillFee` param instead of the `fulfillFee` state as shown in lines 386, 392 and 394 - 401.

xOracle.sol

```
386 function transferRequestFee(uint256 _reqId, address _from, address _to, uint256 _gasUsed, uint256 _gasPrice, uint256 _fulfillFee) private returns(uint256) {
387     if (_fulfillFee == 0) {
388         return 0;
389     }
390
391     // calculate req fee
392     uint256 reqFee = (_gasPrice * _gasUsed * (FULFILL_FEE_PRECISION + _fulfillFee)) / FULFILL_FEE_PRECISION;
393
394     if(_from != address(this)){
395         IERC20(weth).transferFrom(_from, _to, reqFee);
396     } else {
397         IERC20(weth).transfer(_to, reqFee);
398     }
399
400     emit TransferRequestFee(_reqId, _from, msg.sender, reqFee);
401     return reqFee;
402 }
```

Then, using the leftover gas after calling the `setPrices()` function as gas usage in when call `xOracleCallback()` function in lines 150 and 153 and payout the fulfillment fee to the controller who calls the `fulfillRequest()` function for that user's request and transfer back the leftover fee to the request's owner, as shown in lines 159 - 166.

xOracle.sol

```
132 function fulfillRequest(Data[] memory _data, uint256 _reqId) external onlyController {
133     Request storage request = requests[_reqId];
134     if (request.status != 0) {
135         return;
136     }
137     // set status executed
138     request.status = 1;
139
140     require(request.owner != address(0), "request not found");
141     require(request.expiration > block.timestamp, "request is expired");
142
143     // capture gas used
144     uint256 gasStart = gasleft();
145
146     // set price
```

```
147     (bool priceUpdate, string memory message) = setPrices(request.timestamp, _data);
148
149     // check gas used
150     uint256 gasUsedSetprice = gasStart - gasleft();
151
152     // callback
153     xOracleCallback(request.owner, _reqId, priceUpdate, request.payload, request.callbackGasLimit - gasUsedSetprice);
154
155     // check gas used
156     uint256 gasUsed = gasStart - gasleft();
157
158     // payment request fee
159     uint256 reqFee = transferRequestFee(_reqId, address(this), msg.sender, gasUsed, tx.gasprice, request.fulfillFee);
160
161     require(request.depositReqFee >= reqFee, "reqFee exceed depositReqFee");
162
163     // refund request fee
164     if (request.depositReqFee > reqFee) {
165         IERC20(weth).transfer(request.owner, request.depositReqFee - reqFee);
166     }
167
168     emit FulfillRequest(_reqId, priceUpdate, message);
169 }
```

Additionally, modifying the `xOracleCallback()` function to ensure that the gas usage does not exceed the `_callbackGasLimit`, as shown in line 204 - 206.

xOracle.sol

```
203 function xOracleCallback(address _to, uint256 _reqId, bool _priceUpdate, bytes memory _payload, uint256 _callbackGasLimit) private {
204     (bool success, bytes memory data) = _to.call{gas: _callbackGasLimit}(
205         abi.encodeWithSignature("xOracleCall(uint256,bool,bytes)", _reqId, _priceUpdate, _payload)
206     );
207     emit XOracleCall(_reqId, success, string(data));
208 }
```

Finally, modifying the refund process to return the fulfillment fee to the users, for example, in lines 127, 170, and 416.

xOracle.sol

```
function cancelRequestPrice(uint256 _reqId) external whenNotPaused {
    Request storage request = requests[_reqId];
    require(request.owner == msg.sender, "not owner request");
    require(request.status == 0, "status is not request");

    // set status cancel
    request.status = 2;

    // refund request fee
    IERC20(weth).transfer(request.owner, request.depositReqFee);

    emit CancelRequestPrices(_reqId);
}
```

```
function refundRequest(uint256 _reqId) external onlyController {
    Request storage request = requests[_reqId];
    if (request.status != 0) {
        return;
    }
    // set status refund
    request.status = 3;

    // callback
    xOracleCallback(request.owner, _reqId, false, request.payload,
request.callbackGasLimit);

    // refund request fee
    IERC20(weth).transfer(request.owner, request.depositReqFee);

    emit RefundRequest(_reqId);
}
```

```
function adminRefundRequest(uint256 _reqId) external onlyOwner {
    Request storage request = requests[_reqId];
    require(request.status == 0, "status is not request");

    // set status refund
    request.status = 3;

    // refund request fee
    IERC20(weth).transfer(request.owner, request.depositReqFee);

    emit RefundRequest(_reqId);
}
```

5.3. Inexplicit Solidity Compiler Version

ID	IDX-003
Target	XOracle PriceFeedStore
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	**Resolved** The Crown Labs team has resolved this issue by changing the Solidity compiler to 0.8.19, which can be supported by the Linea chain in commit `878e39fcbf6d32aab9b2274635027a658beb50bc`.

5.3.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

XOracle.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.13;
```

The following table contains all targets which the Inexplicit compiler version is declared.

File	Version
XOracle.sol (L: 3)	^0.8.13
PriceFeedStore.sol (L: 3)	^0.8.18

5.3.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.21. (https://github.com/ethereum/solidity/releases)

XOracle.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.22;
```

For chains that may not be compatible with Solidity compiler version 0.8.22, Inspex suggests using Solidity compiler version 0.8.19 instead, as Solidity compiler version 0.8.20 or later introduces the PUSH0 (0x5f) opcode, which some chains have not yet included. (https://github.com/ethereum/solidity/releases/tag/v0.8.20)

XOracle.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspex

CYBERSECURITY PROFESSIONAL SERVICE