MAY.24

SECURITY REVIEW REPORT FOR LAYERZERO

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered the newly developed OFT Rate Limiter smart contracts for LayerZero. These contracts serve as an example of how a rate limit between chains could be implemented.

Our security assessment was a full review of the 2 smart contract, spanning a total of 3 days.

During our audit, we have identified 1 medium severity vulnerability, various minor vulnerabilities and code optimisations.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/LayerZero-Labs/ondo-oft-rate-limiter/tree/fcf731a47bda74572ed0e54ea2c23cfeddaf458c

The issues described in this report were fixed in the following commit:

https://github.com/LayerZero-Labs/ondo-oft-rate-limiter/tree/d4a52a71b14c230c350e64cfdb5cc19c9d07a840

AUDITING DETAILS

STARTED

21.05.2024

DELIVERED

23.05.2024

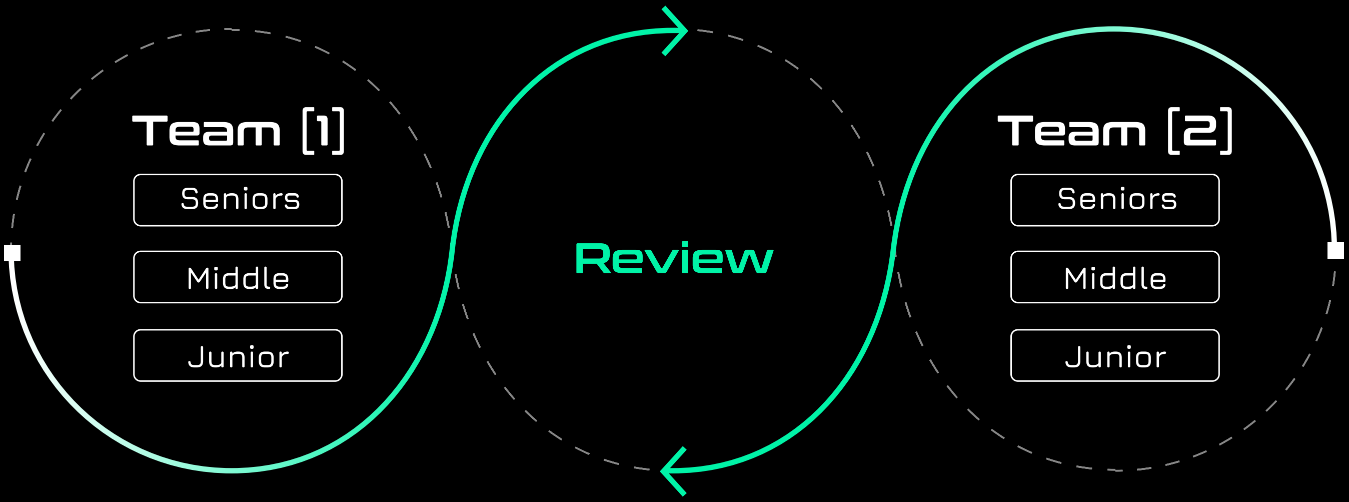
Review Led by

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

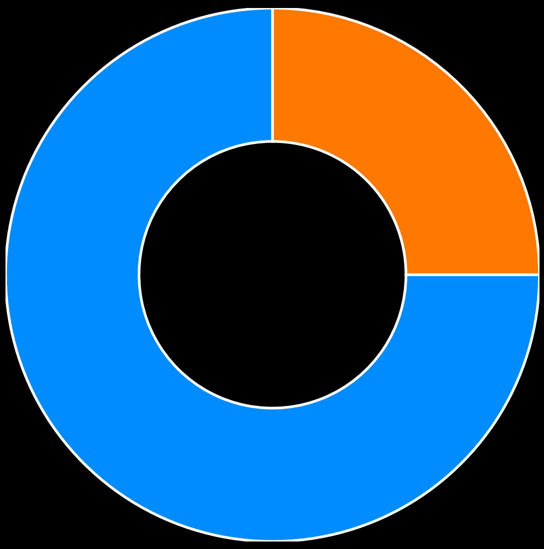
ISSUE SYMBOLIC CODES

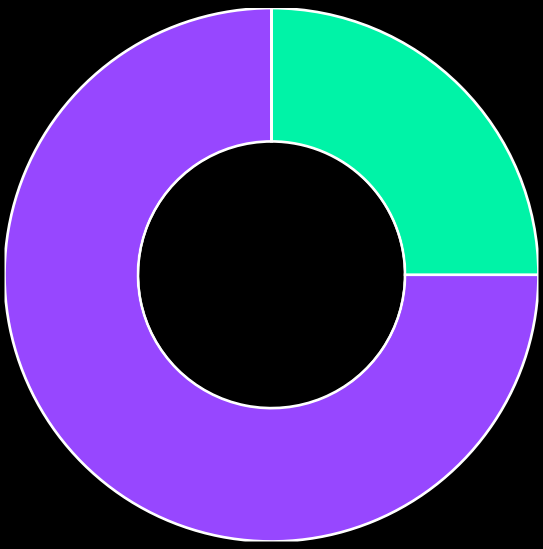
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	1
Low	0
Informational	3

Total: 4





WEAKNESSES

This section contains the list of discovered weaknesses.

LZ01-1

FUNDS CAN BE STUCK BECAUSE THE LIMITS ARE NOT SYNCHRONIZED AND CAN BE CHANGED AFTER A USER TX

SEVERITY: Medium

PATH:

contracts/OFTEfficientRateLimit.sol:L127-L137

REMEDIATION:

One solution is to implement a queue with a gradual release of funds in case the total amount exceeds the limit.

STATUS: Acknowledged

DESCRIPTION:

The limits can be different between chains where the OFT contracts are deployed.

If the source chain allows sending more than the destination allows receiving, transactions with an amount between these two limits can be sent but cannot be received on the destination chain because the corresponding transactions to **lzReceive()** will revert.

```
        // Check if the requested amount exceeds the available capacity
        if (_amount > availableCapacity) revert RateLimitExceeded();
```

The issue will only be triggered by a user error or if the limit is changed after a transaction is sent on the source chain but before its execution on the destination.

There's no mechanism to rescue the funds other than the owners manually setting the limit to the required amount, which can be impossible or harmful in some situations.

```
function _lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _extraData
) internal virtual override {
    // Check and update the rate limit based on the source endpoint ID
(srcEid) and the amount in local decimals from the message.
    _checkAndUpdateRateLimit(_origin.srcEid,
_toLD(OFTMsgCodec.amountSD(_message)), RateLimitDirection.Inbound);
    super._lzReceive(_origin, _guid, _message, _executor, _extraData);
}
```

LZ01-2

REFACTORING DECAY CALCULATION LOGIC IN EFFICIENTRATELIMITER

SEVERITY: Informational

PATH:

contracts/EfficientRateLimiter.sol

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The **EfficientRateLimiter.sol** contract implements rate limiting functionality to control how often specific actions can be executed. It contains logic to manage rate limits for both inbound and outbound transactions, ensuring that usage does not exceed predefined thresholds within specified time windows. However, the current implementation suffers from code duplication in the decay calculation logic across multiple functions.

The decay calculation logic is duplicated in the following functions:

- _amountCanBeSent()
- _amountCanBeReceived()
- _calculateLimits()

```
        uint256 timeSinceLastUpdate = block.timestamp - lastUpdated;
        if (timeSinceLastUpdate >= window) {
            return (0, limit);
        } else {
            uint256 decay = (limit * timeSinceLastUpdate) / window;
            currentAmountInFlight = amountInFlight > decay ? amountInFlight
- decay : 0;
            availableCapacity = limit > currentAmountInFlight ? limit -
currentAmountInFlight : 0;
            return (currentAmountInFlight, availableCapacity);
        }
```

We recommend adding a new helper function, e.g. called **_calculateDecay()** is created to handle the decay calculations. The function can take the current amount in flight, the last updated timestamp, the limit, and the window duration as parameters, and return the decayed amount in flight and the amount that can be processed (sent or received). This will improve the readability of the contract by reducing code duplication and clearly separating the decay logic.

```
    function _amountCanBeSent(
        uint256 _amountInFlight,
        uint128 _lastUpdated,
        uint64 _limit,
        uint48 _window
    ) internal view virtual returns (uint256 currentAmountInFlight, uint256
amountCanBeSent) {
        uint256 timeSinceLastDeposit = block.timestamp - _lastUpdated;
        if (timeSinceLastDeposit >= _window) {
            currentAmountInFlight = 0;
            amountCanBeSent = _limit;
        } else {
            // @dev Presumes linear decay.
            uint256 decay = (_limit * timeSinceLastDeposit) / _window;
            currentAmountInFlight = _amountInFlight <= decay ? 0 :
_amountInFlight - decay;
            // @dev In the event the _limit is lowered, and the 'in-flight'
amount is higher than the _limit, set to 0.
            amountCanBeSent = _limit <= currentAmountInFlight ? 0 : _limit -
currentAmountInFlight;
        }
    }
    function _amountCanBeReceived(
        uint256 _amountInFlight,
        uint128 _lastUpdated,
        uint64 _limit,
        uint48 _window
    ) internal view virtual returns (uint256 currentAmountInFlight, uint256
amountCanBeReceived) {
        uint256 timeSinceLastDeposit = block.timestamp - _lastUpdated;
        if (timeSinceLastDeposit >= _window) {
            currentAmountInFlight = 0;
            amountCanBeReceived = _limit;
        } else {
            // @dev Presumes linear decay.
            uint256 decay = (_limit * timeSinceLastDeposit) / _window;
            currentAmountInFlight = _amountInFlight <= decay ? 0 :
_amountInFlight - decay;
            // @dev In the event the _limit is lowered, and the 'in-flight'
amount is higher than the _limit, set to 0.
            amountCanBeReceived = _limit <= currentAmountInFlight ? 0 :
_limit - currentAmountInFlight;
        }
    }
    function _calculateLimits(
        uint256 amountInFlight,
        uint256 lastUpdated,
        uint64 limit,
        uint48 window
    ) internal view returns (uint256 currentAmountInFlight, uint256
availableCapacity) {
        uint256 timeSinceLastUpdate = block.timestamp - lastUpdated;
        if (timeSinceLastUpdate >= window) {
            return (0, limit);
        } else {
            uint256 decay = (limit * timeSinceLastUpdate) / window;
            currentAmountInFlight = amountInFlight > decay ? amountInFlight
- decay : 0;
            availableCapacity = limit > currentAmountInFlight ? limit -
currentAmountInFlight : 0;
            return (currentAmountInFlight, availableCapacity);
        }
    }
```

LZ01-3

MISSING ACCESS CONTROL IN MINT FUNCTION

SEVERITY: Informational

PATH:

contracts/mocks/OFTEfficientRateLimitMock.sol

REMEDIATION:

Restrict access to the mint function.

STATUS: Acknowledged

DESCRIPTION:

The **OFTEfficientRateLimitMock.sol** contract is a mock implementation of the **OFTEfficientRateLimit** contract, designed for testing purposes. This mock contract includes a mint function that allows for the creation of new tokens.

Although the **OFTEfficientRateLimitMock** contract is intended solely for testing purposes, having the **mint** function publicly accessible is a potential security risk, which in this case has no impact. Public access to minting functions can lead to the unauthorized creation of tokens, which could compromise the integrity of the token supply.

```
function mint(address _to, uint256 _amount) public {
    _mint(_to, _amount);
}
```

LZ01-4

ADDING BATCH RATE LIMIT SETTING FUNCTIONALITY TO OFTEFFICIENTRATELIMIT

SEVERITY: Informational

PATH:

contracts/OFTEfficientRateLimit.sol

REMEDIATION:

Introduce a batchSetRateLimits function that allows setting multiple rate limit configurations in a single transaction.

STATUS: Acknowledged

DESCRIPTION:

The **OFTEfficientRateLimit.sol** contract currently allows setting rate limits through the **setRateLimits()** function. However, this function processes rate limit configurations one by one, which can be inefficient and cumbersome for large-scale updates. Introducing a batch setting functionality can streamline this process, reducing gas costs and improving usability.

```
    function setRateLimits(RateLimitConfig[] calldata _rateLimitConfigs, 
RateLimitDirection direction) external onlyOwner {
        _setRateLimits(_rateLimitConfigs, direction);
    }
```

hexens × LayerZero.