hexens × LayerZero.

SEPT.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.

SCOPE

The analyzed resources are located on:

https://github.com/LayerZero-Labs/orbit-adapter/tree/a79be47f2076a3e3579399dd34df9d2ef68a73de

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

https://github.com/LayerZero-Labs/orbit-adapter/tree/fed4db4c16dcc09981c947c570fbdcf7b2fbd0fc

AUDITING DETAILS

STARTED
11.09.2024

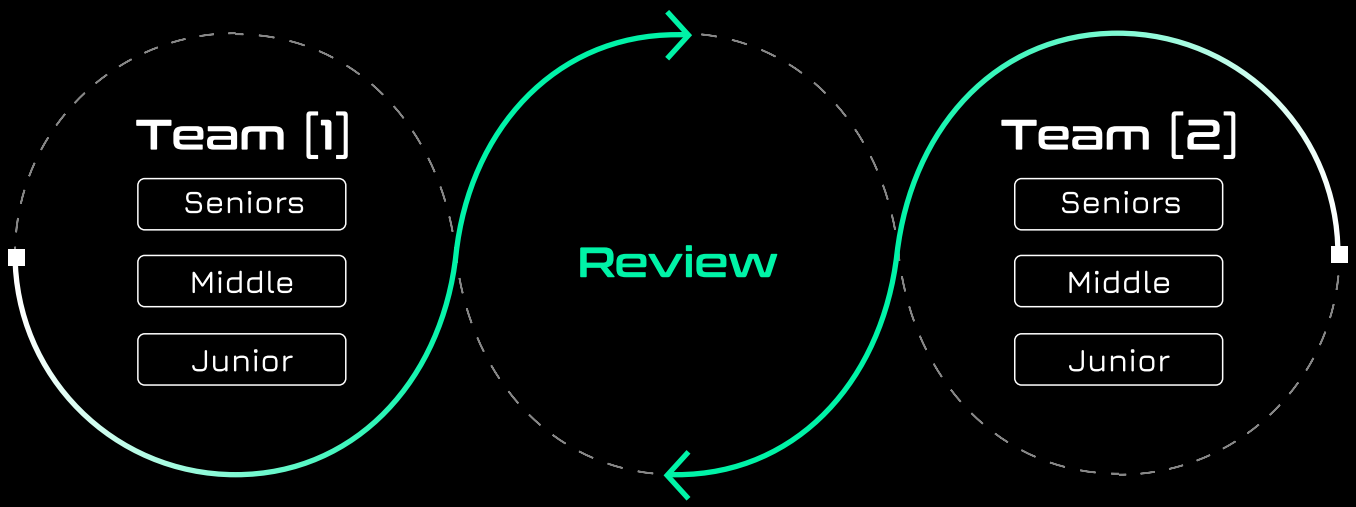
DELIVERED
17.09.2024

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.

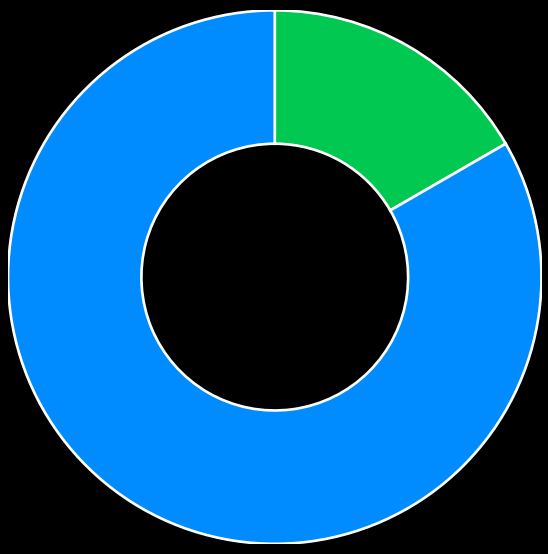
ISSUE SYMBOLIC CODES

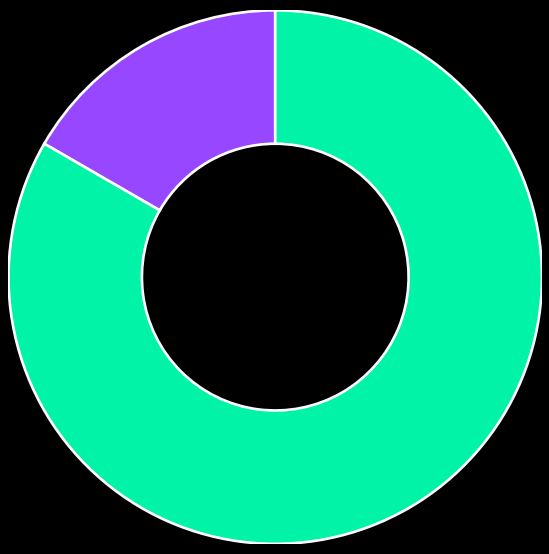
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	0
Medium	0
Low	1
Informational	5

Total: 6





WEAKNESSES

This section contains the list of discovered weaknesses.

LZOARB-6

INCONSISTENT TRANSFER OF FUNDS TO RECEIVER

SEVERITY: Low

PATH:

OrbitNativeOFT1_2.sol:_creditTo#L176-L186

REMEDIATION:

If it is not intended for the receiver to receive tokens instead of native assets on a sendAndCall, then we would recommend to remove the _creditTo to the contract itself (because no mint is required with native assets) and send native assets in callOnOFTReceived instead.

Otherwise we would just recommend to remove the _creditTo in _sendAndCallAck and replace it with a direct _mint instead.

STATUS: Fixed

DESCRIPTION:

The L3 OrbitNativeOFT overrides the **_creditTo** function to send the funds using a native transfer instead. This creates some inconsistency and redundancy when using **sendAndCall** from L2 and receiving it on this contract on L3.

This is because the internal **_sendAndCallAck** function first credits the contract itself with the tokens, in case the receiving call reverts and needs to be retried:

```
if (!credited) {
    amount = _creditTo(_srcChainId, address(this), amount);
    creditedPackets[_srcChainId][_srcAddress][_nonce] = true;
}
```

It will do a native token transfer to itself (which is a redundant transfer) but this also triggers the **receive** function of the contract:

```
receive() external payable {
    deposit();
}
function deposit() public payable {
    _mint(msg.sender, msg.value);
    emit Deposit(msg.sender, msg.value);
}
```

And so it will mint the tokens to the contract itself.

It does not create an impactful issue because in **callOnOFTReceived** where it makes the call to the receiver, it uses **_transferFrom** to send the tokens to the receiver:

```
_amount = _transferFrom(address(this), _to, _amount);
```

However, it does create inconsistency, because the receiver will receive native assets if **send** is used, while they will receive tokens if **sendAndCall** is used. The native token transfer to the contract itself that happens in **_creditTo** is also redundant as the contract could've just called **_deposit** directly (or send native tokens in **callOnOFTReceived**).

```
function _creditTo(
    uint16,
    address _toAddress,
    uint256 _amount
) internal override returns (uint) {
    (bool success, ) = _toAddress.call{value: _amount}("");
    if (!success) {
        revert CreditToFailed();
    }
    return _amount;
}
```

LZOARB-5

REDUNDANT MINT IN _DEBITMSGSENDER

SEVERITY: Informational

PATH:

OrbitNativeOFT1_2.sol:_debitMsgSender#L129-L149

REMEDIATION:

Remove the _mint and adjust the final to-be-burned amount that goes into _burn to not include the mintAmount.

STATUS: Fixed

DESCRIPTION:

This function is responsible for taking funds from the **msg.sender** for the cross-chain transfer. If the user has insufficient balance, the amount is taken from **msg.value**.

In this function, the extra value that was sent using **msg.value** gets minted directly to the **msg.sender** using **_mint** only to be burned with **_burn** a few lines later.

This is in contract to **_debitMsgFrom** where the **_burn** amount is adjusted to not include the value provided in **msg.value**.

The extra **_mint** is a waste of gas as it updates the user's balance only for it to be removed again later in **_burn**, while also emitting events. This is also inconsistent with **_debitMsgFrom**.

```
function _debitMsgSender(uint256 _amount) internal returns (uint256
messageFee) {
    uint256 msgSenderBalance = balanceOf(msg.sender);


    if (msgSenderBalance < _amount) {
        if (_amount > msgSenderBalance + msg.value) {
            revert InsufficientMessageValue();
        }


        // user can cover difference with additional msg.value ie. wrapping
        uint256 mintAmount = _amount - msgSenderBalance;
        _mint(address(msg.sender), mintAmount);


        // update the messageFee to take out mintAmount
        messageFee = msg.value - mintAmount;
    } else {
        messageFee = msg.value;
    }


    _burn(msg.sender, _amount);
    return messageFee;
}
```

CAP CHECK OPTIMIZATION FOR GAS EFFICIENCY IN _DEBITFROM FUNCTION

SEVERITY: Informational

PATH:

src/L2/OrbitProxyOFT1_2.sol::_debitFrom()#L57-L79

REMEDIATION:

Move the cap check to the beginning of the function, before the token transfer logic. This "fail fast" approach will save gas in cases where the cap is exceeded.

STATUS: Acknowledged

DESCRIPTION:

In the **OrbitProxyOFT1_2.sol** contract, the **_debitFrom()** function includes a cap check to prevent amounts exceeding the maximum allowed (**type(uint64).max**). Currently, the cap check is performed after the token transfer logic, which results in unnecessary gas consumption if the transfer amount exceeds the cap.

```
    function _debitFrom(
        address _from,
        uint16,
        bytes32,
        uint256 _amount
    ) internal virtual override returns (uint) {
        if (_from != _msgSender()) {
            revert OwnerNotSendCaller();
        }

        _amount = _transferFrom(_from, address(bridge), _amount);

        // _amount still may have dust if the token has transfer fee, then
give the dust back to the sender
        (uint256 amount, uint256 dust) = _removeDust(_amount);
        if (dust > 0) bridge.executeCall(_from, dust, "");

        uint256 cap = _sd2ld(type(uint64).max);
        if (amount > cap) {
            revert AmountOverflow();
        }

        return amount;
    }
```

USE UNCHECKED WHEN IT IS SAFE

SEVERITY: Informational

PATH:

src/L3/OrbitNativeOFT1_2.sol::_debitMsgSender()#L129-L149

src/L3/OrbitNativeOFT1_2.sol::_debitMsgFrom()#L151-L174

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In Solidity (^0.8.0), adding an unchecked keyword for arithmetical operations can decrease gas usage on contracts where underflow/underflow is unrealistic.

The **_debitMsgSender()** and **_debitMsgFrom()** functions in the **OrbitNativeOFT1_2.sol** contract can be optimized for better performance and clarity by using **unchecked** arithmetic for safe performance gains.

```
    function _debitMsgSender(uint256 _amount) internal returns (uint256
messageFee) {
        uint256 msgSenderBalance = balanceOf(msg.sender);

        if (msgSenderBalance < _amount) {
            if (_amount > msgSenderBalance + msg.value) {
                revert InsufficientMessageValue();
            }

            // user can cover difference with additional msg.value ie.
wrapping
            uint256 mintAmount = _amount - msgSenderBalance;
            _mint(address(msg.sender), mintAmount);

            // update the messageFee to take out mintAmount
            messageFee = msg.value - mintAmount;
        } else {
            messageFee = msg.value;
        }

        _burn(msg.sender, _amount);
        return messageFee;
    }
```

```
    function _debitMsgFrom(address _from, uint256 _amount) internal returns
(uint256 messageFee) {
        uint256 msgFromBalance = balanceOf(_from);

        if (msgFromBalance < _amount) {
            if (_amount > msgFromBalance + msg.value) {
                revert InsufficientMessageValue();
            }

            // user can cover difference with additional msg.value ie.
wrapping
            uint256 amountFromValue = _amount - msgFromBalance;

            // overwrite the _amount to take the rest of the balance from
the _from address
            _amount = msgFromBalance;

            // update the messageFee to take out amountFromValue
            messageFee = msg.value - amountFromValue;
        } else {
            messageFee = msg.value;
        }

        _spendAllowance(_from, msg.sender, _amount);
        _burn(_from, _amount);
        return messageFee;
    }
```

As there are already checks in place, using **unchecked** will make the functions more gas efficient, without compromising the security:

```
       function _debitMsgSender(uint256 _amount) internal returns (uint256
messageFee) {
        uint256 msgSenderBalance = balanceOf(msg.sender);

        if (msgSenderBalance < _amount) {
            if (_amount > msgSenderBalance + msg.value) {
                revert InsufficientMessageValue();
            }
+           unchecked {
                // user can cover difference with additional msg.value ie.
wrapping
            uint256 mintAmount = _amount - msgSenderBalance;
            _mint(address(msg.sender), mintAmount);

            // update the messageFee to take out mintAmount
            messageFee = msg.value - mintAmount;
+           }
        } else {
            messageFee = msg.value;
        }

        _burn(msg.sender, _amount);
        return messageFee;
    }
```

```
    function _debitMsgFrom(address _from, uint256 _amount) internal returns
(uint256 messageFee) {
        uint256 msgFromBalance = balanceOf(_from);

        if (msgFromBalance < _amount) {
            if (_amount > msgFromBalance + msg.value) {
                revert InsufficientMessageValue();
            }
+           unchecked {
                // user can cover difference with additional msg.value ie.
wrapping
            uint256 amountFromValue = _amount - msgFromBalance;

              // overwrite the _amount to take the rest of the balance from
the _from address
            _amount = msgFromBalance;

            // update the messageFee to take out amountFromValue
            messageFee = msg.value - amountFromValue;
+           }
        } else {
            messageFee = msg.value;
        }

        _spendAllowance(_from, msg.sender, _amount);
        _burn(_from, _amount);
        return messageFee;
    }
```

REDUNDANT BALANCE CHECK IN WITHDRAW

SEVERITY: Informational

PATH:

OrbitNativeOFT1_2.sol:withdraw#L64-L74

REMEDIATION:

If the custom error is not a UI requirement, then we would recommend to consider removing the check and relying on the check in _burn instead.

STATUS: Fixed

DESCRIPTION:

In the withdraw function of the L3 OrbitNativeOFT contract, the withdraw function allows the user to burn their tokens into native assets.

The function starts with a balance check for the amount, but the following function call **_burn** also already checks this balance and will revert if it is insufficient.

```
function withdraw(uint256 _amount) external nonReentrant {
    if (_amount > balanceOf(msg.sender)) {
        revert InsufficientBalance();
    }
    _burn(msg.sender, _amount);
    (bool success, ) = msg.sender.call{value: _amount}("");
    if (!success) {
        revert WithdrawalFailed();
    }
    emit Withdrawal(msg.sender, _amount);
}
```

INCONSISTENT USE OF _MSGSENDER() AND MSG.SENDER

SEVERITY: Informational

PATH:

OrbitNativeOFT1_2.sol

REMEDIATION:

We recommend to replace instances of msg.sender with _msgSender().

STATUS: Fixed

DESCRIPTION:

The L3 OrbitNativeOFT contract implements the LayerZero OFTV2 contract, which inherits from OpenZeppelin's Context and it uses **_msgSender()** everywhere to get the sender in the current context.

However this contract uses **msg.sender** in multiple places, which is inconsistent with the use of Context. If the _msgSender() function would ever be overridden in the future, it could lead to critical bugs.

```
        function _debitMsgSender(uint256 _amount) internal returns (uint256
messageFee) {
            uint256 msgSenderBalance = balanceOf(msg.sender);
            [..]
    }
```


hexens
×
LayerZero.