hexens × LayerZero.

AUG.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 focused on the Wrapped Asset Bridge v2 (WABv2), which facilitates the provision of existing ERC-20 token liquidity to remote chains within the LayerZero protocol. The protocol utilizes the Omnichain Fungible Token (OFT), enabling users to transfer ERC-20 tokens across supported blockchains through wrapped tokens. The wrapped asset bridge serves as a token bridge, allowing tokens to move back and forth between endpoints. Adding ERC-20 tokens to the WAB is permissionless.

Our security assessment was a comprehensive review of the four smart contracts, conducted over one week.

During our audit, we did not find any major vulnerabilities. However, we did identify three low-severity issues and one informational issue.

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/wrapped-asset-bridge-v2/commit/eeffb429c6d63cbe4ad5300b90a4b9250aeffc39

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

https://github.com/LayerZero-Labs/wrapped-asset-bridge-v2/commit/d0a0b4f73ab4e50bef81ff8850d9adc44328915b

AUDITING DETAILS

STARTED
12.08.2024

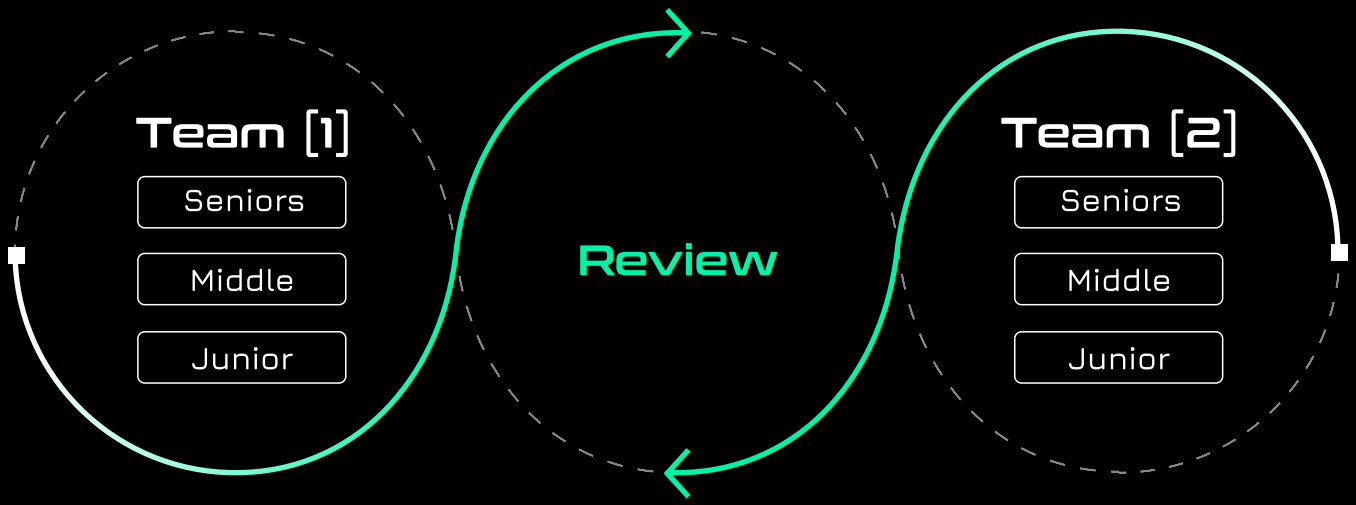
DELIVERED
20.08.2024

Review Led by

TRUNG DINH
Senior Security Researcher | 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.

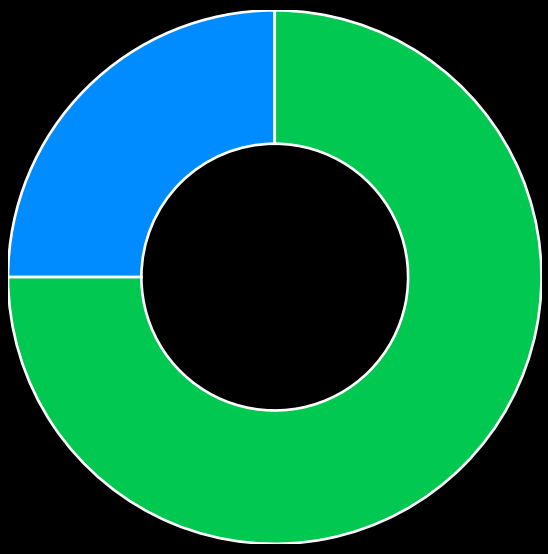
ISSUE SYMBOLIC CODES

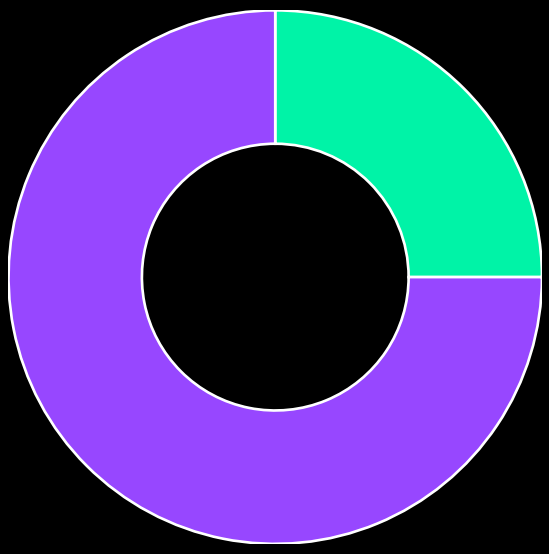
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

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

Total: 4





WEAKNESSES

This section contains the list of discovered weaknesses.

LZWAB-4

DUST REMOVAL IN FEE CALCULATION LEADS TO OVERCHARGING USERS

SEVERITY: Low

PATH:

contracts/WOFT.sol::_debitView()#L87-L100

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

In the **WOFT.sol** contract, the function **_debit()** calculates and deducts fees when burning tokens. This involves calling the **_debitView()** function, which determines the amount to be received after applying the fee and dust removal.

```
    function _debit(
        address _from,
        uint256 _amountLD,
        uint256 _minAmountLD,
        uint32 _dstEid
    ) internal virtual override returns (uint256 amountSentLD, uint256
amountReceivedLD) {
        (amountSentLD, amountReceivedLD) = _debitView(_amountLD,
_minAmountLD, _dstEid);

        uint256 fee = amountSentLD - amountReceivedLD;
        if (fee > 0) {
            _transfer(_from, address(WOFT_FACTORY), fee);
        }
        _burn(_from, amountReceivedLD);
    }
```

However, the current logic for fee calculation in **_debitView()** inadvertently causes the fee transferred to the WOFT factory to be larger than intended. This discrepancy arises due to the sequence of operations. The function first assigns **amountSentLD = _amountLD**, then calculates **amountReceivedLD = _removeDust(_amountLD - feeLD)**. This leads to the difference, **uint256 fee = amountSentLD - amountReceivedLD**, being greater than the initially calculated **feeLD**, resulting in users being overcharged.

Let's assume:

- LOCAL_DECIMALS = 18
- SHARED_DECIMALS = 6
- decimalConversionRate = 10^12 (10^(18-6))
- Fee rate = 0.1% (for simplicity)

Example transaction: User wants to send 1.2345678901234567 89 tokens (1234567890123456789 in wei)

Step by step:

1. Initial **_amountLD** = 1234567890123456789
2. Calculate fee: **feeLD** = 0.1% of 1234567890123456789 = 1234567890123456 (rounded down)

3. Subtract fee: 1234567890123456789 - 1234567890123456 = 1233333322233333333
4. Apply **_removeDust**: 1233333322233333333 / 10^12 * 10^12 = 1233333322233000000
5. Set **amountReceivedLD** = 1233333322233000000
6. In the current implementation: **amountSentLD** = 1234567890123456789 (original amount) Actual fee transferred = 1234567890123456789 - 1233333322233000000 = 1234567890456789

We can see that the actual fee transferred (1234567890456789) is larger than the calculated fee (1234567890123456) by 333333 wei.

```
    function _debitView(
        uint256 _amountLD,
        uint256 _minAmountLD,
        uint32 /*_dstEid*/
    ) internal view override returns (uint256 amountSentLD, uint256
amountReceivedLD) {
        amountSentLD = _amountLD;

        // @dev Check for fees on the factory
        uint256 feeLD = WOFT_FACTORY.getFee(address(this), _amountLD);
        amountReceivedLD = _removeDust(_amountLD - feeLD);

        // @dev Check for slippage
        if (amountReceivedLD < _minAmountLD) revert
SlippageExceeded(amountReceivedLD, _minAmountLD);
    }
```

The fee should be calculated after dust removal to ensure the correct fee amount is transferred:

```
    function _debitView(
        uint256 _amountLD,
        uint256 _minAmountLD,
        uint32 /*_dstEid*/
    ) internal view override returns (uint256 amountSentLD, uint256
amountReceivedLD) {
--      amountSentLD = _amountLD;

        // @dev Check for fees on the factory
        uint256 feeLD = WOFT_FACTORY.getFee(address(this), _amountLD);
        amountReceivedLD = _removeDust(_amountLD - feeLD);

++      amountSentLD = amountReceivedLD + feeLD;

        // @dev Check for slippage
        if (amountReceivedLD < _minAmountLD) revert
SlippageExceeded(amountReceivedLD, _minAmountLD);
    }
```

INCONSISTENT FEE BOUNDARY REQUIREMENT IN THE CONSTRUCTOR AND THE SETDEFAULTFEEBPS() FUNCTION

SEVERITY: Low

PATH:

contracts/WOFTFactory.sol#L56-L57

contracts/WOFTFactory.sol#L43-L48

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **WOFTFactory** contract's constructor, **_defaultFeeBps** must be less than **BPS_DENOMINATOR**. However, in the **WOFTFactory.setDefaultFeeBps()** function, the **feeBps** can be equal to **BPS_DENOMINATOR**, resulting in an inconsistency in the fee boundary requirements of the two functions.

```
function setDefaultFeeBps(uint16 _feeBps) external onlyOwner {
    if (_feeBps > BPS_DENOMINATOR) revert InvalidBps();
```

```
constructor(
    address _endpoint,
    address _delegate,
    uint16 _defaultFeeBps
) OApp(_endpoint, _delegate) Ownable(_delegate) {
    if (_defaultFeeBps >= BPS_DENOMINATOR) revert InvalidBps();
```

Consider modifying the **WOFTFactory's** constructor as follows:

```
    constructor(
        address _endpoint,
        address _delegate,
        uint16 _defaultFeeBps
    ) OApp(_endpoint, _delegate) Ownable(_delegate) {
-       if (_defaultFeeBps >= BPS_DENOMINATOR) revert InvalidBps();
+       if (_defaultFeeBps > BPS_DENOMINATOR) revert InvalidBps();
        defaultFeeBps = _defaultFeeBps;
    }
```

SINGLE-STEP OWNERSHIP CHANGE INTRODUCES RISKS

SEVERITY: Low

PATH:

WOFT.sol

WOFTAdapter.sol

WOFTAdapterFactory.sol

WOFTFactory.sol

REMEDIATION:

Use OpenZeppelin's Ownable2Step.sol.

STATUS: Acknowledged

DESCRIPTION:

Mentioned contracts import OZ's **Ownable.sol**, as the contracts are non-upgradeable and have some essential **onlyOwner** functionality, it is especially important that transfers of ownership should be handled with care.

Single-step ownership transfers add the risk of setting an unwanted owner by accident if the ownership transfer is not done with excessive care.

```
import { Ownable } from "@openzeppelin/contracts/access/Ownable.sol";
```

CONSTANT AND IMMUTABLE VARIABLES COULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

contracts/WOFTAdapterFactory.sol#L27, L28, L31

contracts/WOFTAdapter.sol#L20

contracts/WOFTFactory.sol#L24, L25, L26

contracts/WOFT.sol#L31

REMEDIATION:

Make the constants and the immutable variables private instead of public.

STATUS: Acknowledged

DESCRIPTION:

The project uses **public** visibility for the most constant parameters in various contracts.

Setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment **calldata**, won't need to store the bytes of the values outside of where it's used, and won't add another entry to the method ID table. The values can still be read from the verified contract source code.

```
uint16 public constant SEND = 1;
uint16 public constant SEND_AND_CALL = 2;
```

```
address public immutable FACTORY;
```

```
uint32 public constant ADAPTER_EID = 30101;
uint256 public constant BPS_DENOMINATOR = 10_000;
address public constant NATIVE_TOKEN_ADDRESS =
address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE);
```

```
IWOFTFactory public immutable WOFT_FACTORY;
```


hexens
×
LayerZero.