hexens × LayerZero.

JUNE.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 Layer Zero Network's Donate and Claim contracts, which handle the distribution of the ZRO token using Merkle proofs and a donation mechanism.

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

During our audit, we identified two high-severity vulnerabilities that could cause users to lose their donation tokens without being able to claim their ZRO tokens in return. Additionally, we found four medium-severity issues, one low-severity issue, 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/ZROClaim/commit/c1101c12c8336e9f743dcbd86ece1bcb999a3686

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

https://github.com/LayerZero-Labs/ZROClaim/commit/bf965039f2793cfcc91d265a87c6288302ed4b96

AUDITING DETAILS

STARTED
17.06.2024

DELIVERED
18.06.2024

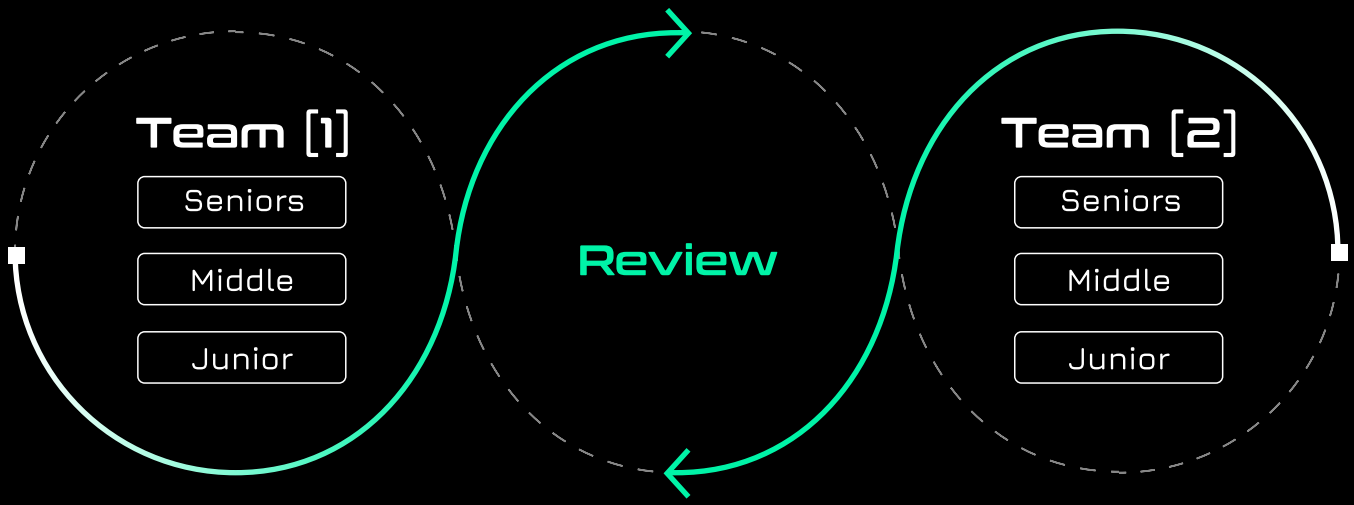
Review Led by

TRUNG DINH

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.

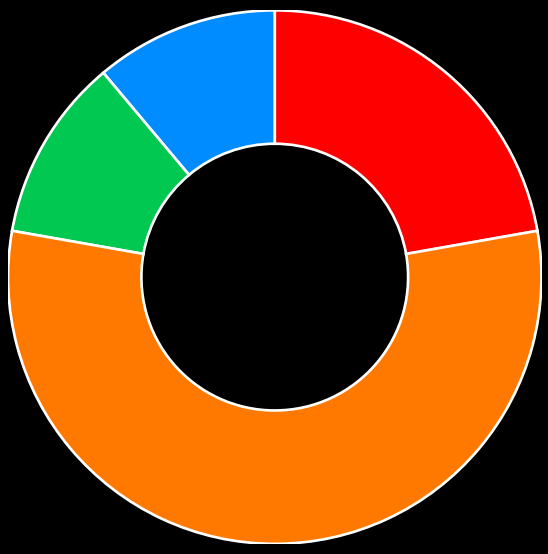
ISSUE SYMBOLIC CODES

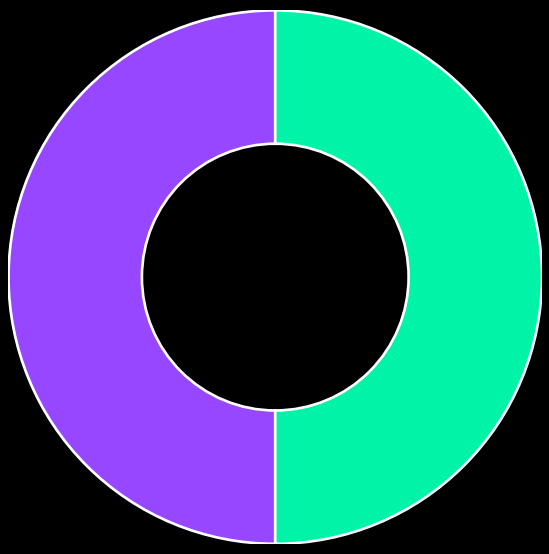
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	2
Medium	4
Low	1
Informational	1

Total: 8





WEAKNESSES

This section contains the list of discovered weaknesses.

LZ02-12

LACK OF CONSIDERATION OF DECIMALCONVERSIONRATE WHEN SETTING MINAMOUNTLD IN CALLBACK FUNCTION OF CLAIMLOCAL

SEVERITY: High

PATH:

ClaimLocal.sol:_lzReceive

REMEDIATION:

Consider adjusting the value of minAmountLD to the _zroAmount after removing dust.

STATUS: Fixed

DESCRIPTION:

The function **ClaimRemote.claim()** is used to claim ZRO drops from another chain. Through a LayerZero message, the callback function **_lzReceive** in ClaimLocal is called.

In this function, the claimed **ZRO** tokens amount **_zroAmount** is sent **to** the specified to address on the source chain. To accomplish this, a **SendParam** object is constructed with both **amountLD** and **minAmountLD** set to **_zroAmount**, which is then used in the **OFT.send(zroToken)** invocation.

The problem arises with the **SlippageExceeded** requirement in the flow **OFTCore.send() -> OFT._debit() -> OFTCore._debitView()**.

LayerZero-v2/packages/layerzero-v2/evm/oapp/contracts/oft/OFTCore.sol at 417cbb9eb68a4f678490d18728973c8c99f3f017 · LayerZero-Labs/LayerZero-v2

```
function _removeDust(uint256 _amountLD) internal view virtual returns
(uint256 amountLD) {
    return (_amountLD / decimalConversionRate) * decimalConversionRate;
}

function _debitView(
    uint256 _amountLD,
    uint256 _minAmountLD,
    uint32 /*_dstEid*/
) internal view virtual returns (uint256 amountSentLD, uint256
amountReceivedLD) {
    // @dev Remove the dust so nothing is lost on the conversion between
chains with different decimals for the token.
    amountSentLD = _removeDust(_amountLD);
    // @dev The amount to send is the same as amount received in the default
implementation.
    amountReceivedLD = amountSentLD;

    // @dev Check for slippage.
    if (amountReceivedLD < _minAmountLD) {
        /// $audit could revert here
        revert SlippageExceeded(amountReceivedLD, _minAmountLD);
    }
}
```

As illustrated in the **_debitView()** function snippet, the amount of tokens for cross-chain transfer undergoes a dust removal process to prevent loss during chain conversion. The **_removeDust()** function can result in **amountReceivedLD** being less than **amountSentLD** if **_amountLD** isn't divisible by **decimalConversionRate**.

Since both **amountLD** and **minAmountLD** are set to **_zroAmount** when creating the **SendParam** object, if **amountLD** isn't divisible by **decimalConversionRate**, the actual sent amount will be less than **minAmountLD**, triggering the **SlippageExceeded** revert. This leads to users losing funds when they donate to the **DonateRemote** contract but cannot claim their **ZRO** tokens through the **ClaimRemote** contract.

```
    function _lzReceive(
        Origin calldata _origin,
        bytes32 /*_guid*/,
        bytes calldata _payload,
        address /*_executor*/,
        bytes calldata /*_extraData*/
    ) internal override {
        (address user, uint256 zroAmount, address to) = abi.decode(_payload,
(address, uint256, address));

        // msg.sender in this case is the endpoint
        uint256 availableToClaim = _availableToClaim(user, zroAmount);

        bytes memory emptyBytes = new bytes(0);
        SendParam memory sendParams = SendParam({
            dstEid: _origin.srcEid,
            to: addressToBytes32(to),
            amountLD: availableToClaim,
            minAmountLD: availableToClaim,
            extraOptions: emptyBytes,
            composeMsg: emptyBytes,
            oftCmd: emptyBytes
        });

        // solhint-disable-next-line check-send-result
        IOFT(zroToken).send{ value: msg.value }(sendParams,
MessagingFee(msg.value, 0), address(this));

        emit ClaimRemote(user, availableToClaim, _origin.srcEid, to);
    }
```

LZ02-19

UNABLE TO CLAIM REMOTELY DUE TO LACK OF LAYER ZERO FEE

SEVERITY: High

PATH:

contracts/DonateAndClaim.sol:L87-L88

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Within the function **ClaimRemote._claim()**, the **msg.value** is used as the fee for sending a cross-chain message via Layer Zero. If **msg.value** is **0**, the message cannot be sent.

```
// step 4: send the message cross chain
// solhint-disable-next-line check-send-result
receipt = endpoint.send{ value: msg.value }(
    MessagingParams(remoteEid, _getPeerOrRevert(remoteEid), payload,
options, false),
    payable(_user)
);
```

The issue arises in the **DonateAndClaim.donateAndClaim()** function, where the **claim()** function of the **claimContract** is called without any attached native tokens. If the **claimContract** is an instance of **ClaimRemote**, the cross-chain message cannot be sent due to the lack of a Layer Zero fee, preventing the user from claiming their tokens.

```
// claim
return IClaim(claimContract).claim(msg.sender, currency, _zroAmount, _proof,
_to, _extraBytes);
```

Consider modifying the code from line 84 to line 88 to:

```
// donate
- IDonate(donateContract).donate{ value: msg.value }(currency,
amountToDonate, msg.sender);
+ uint donationAmountNative;
+ if (currency == Currency.Native && stargateNative != address(0)) {
+   donationAmountNative = amountToDonate;
+ }
+ IDonate(donateContract).donate{ value: donationAmountNative }(currency,
amountToDonate, msg.sender);

// claim
- return IClaim(claimContract).claim(msg.sender, currency, _zroAmount,
_proof, _to, _extraBytes);
+ return IClaim(claimContract).claim{ value: msg.value -
donationAmountNative }(msg.sender, currency, _zroAmount, _proof, _to,
_extraBytes);
```

LZ02-17

DONATEREMOTE CANNOT BE DEPLOYED TO SKIP TOKEN DONATIONS

SEVERITY: Medium

PATH:

contracts/donate/DonateRemote.sol:L33-L35

REMEDIATION:

The constructor of DonateRemote should check that the token Stargate addresses are not address(0) before calling the token() function, similar to the pattern in DonateCore.

STATUS: Fixed

DESCRIPTION:

DonateCore deployments have options where the stargate addresses of tokens can be set to address(0), which will skip the donations of those tokens.
However, the DonateRemote contract doesn't allow this because it attempts to call **IOFT(_stargateUsdc).token()**, **IOFT(_stargateUsdt).token()** and **IOFT(_stargateNative).token()** in the constructor function. If any of these stargate addresses is address(0), the deployment will revert.
Therefore, all of these tokens must be used for donations in DonateRemote, and the protocol can't skip donations for any of them.

```
constructor(
    uint32 _remoteEid,
    address _donationReceiver,
    address _stargateUsdc,
    address _stargateUsdt,
    address _stargateNative
) DonateCore(_donationReceiver, _stargateUsdc, _stargateUsdt,
_stargateNative) {
    remoteEid = _remoteEid;

    IERC20(IOFT(_stargateUsdc).token()).forceApprove(_stargateUsdc,
type(uint256).max);
    IERC20(IOFT(_stargateUsdt).token()).forceApprove(_stargateUsdt,
type(uint256).max);

    // Only accept stargate if it is the native token
    if (IOFT(_stargateNative).token() != address(0)) {
        revert InvalidNativeStargate();
    }
}
```

LZ02-20

NATIVE CURRENCY PRICES WILL BECOME STALE

SEVERITY: Medium

PATH:

contracts/claim/ClaimCore.sol:L56-L62

REMEDIATION:

We would recommend to use an oracle or at least add a manual function to update the price numerator.

STATUS: Acknowledged

DESCRIPTION:

The **ClaimCore** contract responsible for checking if a user deposited enough assets and doesn't have a function to set **numeratorNative** (the price of the native currency). It'll be set when the contract is created, and the chosen price will be used even if there's a huge fluctuation in the spot price, which will make claiming cheaper in one native currency versus another one.

For example, if claiming ZRO on Ethereum has a numerator of 1% ETH, while on BSC it is 5% BNB which at the start has a similar value. If the price of BNB plummets, it would become much cheaper to donate on BSC and claim the ZRO there remotely.

```
        if (_stargateNative != address(0) && _nativePrice > 0) {
            numeratorNative = _nativePrice * 10 ** (18 - 1);
            // Validate this is an actual native pool, eg. NOT WETH
            if (IOFT(_stargateNative).token() != address(0)) {
                revert InvalidNativeStargate();
            }
        }
```

INCORRECT USE OF _NATIVEPRICE IN CONTRACT CLAIMCORE

SEVERITY: Medium

PATH:

contracts/claim/ClaimCore.sol:L57

contracts/claim/ClaimCore.sol:L80

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

In the implementation of the **ClaimCore.requiredDonation()** function, the amount of native wei tokens needed to claim **_zroAmount** ZRO tokens is calculated as follows:

```
native = (_zroAmount * numeratorNative) / DENOMINATOR;
       = (_zroAmount * _nativePrice * 10^(18 - 1)) / 10^18
       = (_zroAmount / 10^18) * 1/10
           * _nativePrice
           * 10^18
```

The native amount calculation can be divided into three parts:

1. **(_zroAmount / 10^18) * 1/10**: This part calculates the USD value of **_zroAmount** wei ZRO tokens.
2. **_nativePrice**: Introduced in the **ClaimCore** contract's constructor, this parameter represents the price of the native token and is used to convert the USD value to the amount of native tokens.

3. **10^18**: This constant value represents the decimals of the native token, helping to determine the required amount in wei.

The issue arises with the second part, **_nativePrice**. In the **native** calculation, **_nativePrice** is multiplied by the first part to determine the required amount of native tokens. This implies that **_nativePrice** should be defined as:

- **_nativePrice**: "Amount of native tokens that worth 1 USD"

This definition leads to a problem when the native token has a high value (e.g., ETH). In such cases, **_nativePrice** will be **0** because 1 ETH is worth more than 1 USD.

```
numeratorNative = _nativePrice * 10 ** (18 - 1);
```

```
uint256 native = (_zroAmount * numeratorNative) / DENOMINATOR;
```

Consider passing the _nativePrice with precision to the constructor.

```
constructor(
    bytes32 _merkleRoot,
    address _donateContract,
    address _stargateUsdc,
    address _stargateUsdt,
    address _stargateNative,
    uint256 _nativePrice,
+    uint256 _nativePricePrecision,
    address _owner
) Ownable(_owner) {
    merkleRoot = _merkleRoot;
    donateContract = IDonate(_donateContract);

    // TODO needs tests
    if (_stargateUsdc != address(0)) {
        numeratorUsdc = 1 * 10 **
(IERC20Metadata(IOFT(_stargateUsdc).token()).decimals() - 1);
    }

    if (_stargateUsdt != address(0)) {
        numeratorUsdt = 1 * 10 **
(IERC20Metadata(IOFT(_stargateUsdt).token()).decimals() - 1);
    }

    // native is always denominated in 18 decimals
    if (_stargateNative != address(0) && _nativePrice > 0) {
-        numeratorNative = _nativePrice * 10 ** (18 - 1);
+        numeratorNative = _nativePrice * 10 ** (18 - 1) /
_nativePricePrecision;
        // Validate this is an actual native pool, eg. NOT WETH
        if (IOFT(_stargateNative).token() != address(0)) {
            revert InvalidNativeStargate();
        }
    }
}
```

Another approach is to pre-calculate the value of `_nativePrice * 10 ** (18 - 1) / _nativePricePrecision` and then pass the result to the constructor, assigning it to `numeratorNative`. This method can save a small amount of gas by avoiding the need for computation at runtime.

Nice to have: Provide an explanation in the constructor indicating that `_nativePrice` represents the amount of native tokens needed to purchase 1 USD.

LZ02-22

INACCURATE USD VALUATION IN CLAIMCORE CONTRACT

SEVERITY: Medium

PATH:

contracts/claim/ClaimCore.sol

REMEDIATION:

We recommend integrating Chainlink oracles to obtain accurate real-time price feeds for USDC, USDT.

STATUS: Acknowledged

DESCRIPTION:

The **ClaimCore.sol** contract currently operates under the assumption that 1 USDC and **1 USDT** are exactly equivalent to **1 USD**. However, this assumption does not accurately reflect the real-world market value of these stablecoins relative to **USD**. In reality, there's no hard peg, the price can go both above or below **1 USD** (USDC). This discrepancy in valuation poses a potential risk of exploitation through arbitrage opportunities, where users could exploit the price difference between stablecoins to profit unfairly from the system.

If the actual price of **USDC** is lower than **1 USD**, users could potentially exploit the price difference by donating **USDC** to claim ZRO and then swapping the ZRO for another asset such as **USDT** that has a higher value relative to **USD**. The contract does not account for the actual market value of **USDC** and **USDT**, leading to an incorrect valuation of donations and potential exploitation of price differentials between stablecoins.

```
        // TODO needs tests
        if (_stargateUsdc != address(0)) {
            numeratorUsdc = 1 * 10 **
(IERC20Metadata(IOFT(_stargateUsdc).token()).decimals() - 1);
        }

        if (_stargateUsdt != address(0)) {
            numeratorUsdt = 1 * 10 **
(IERC20Metadata(IOFT(_stargateUsdt).token()).decimals() - 1);
        }
```

```
    function requiredDonation(uint256 _zroAmount) public view returns
(Donation memory) {
        uint256 usdc = (_zroAmount * numeratorUsdc) / DENOMINATOR;
        uint256 usdt = (_zroAmount * numeratorUsdt) / DENOMINATOR;
        uint256 native = (_zroAmount * numeratorNative) / DENOMINATOR;
        return Donation(usdc, usdt, native);
    }
```

LZ02-16

PARTIAL CLAIMS ARE UNSUPPORTED

SEVERITY: Low

PATH:

ClaimLocal.sol:_claim, ClaimRemote.sol:_claim

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

Both the ClaimLocal and ClaimRemote contracts only allow the user to claim the exact amount that is inside of the leaf in the Merkle tree. Because a donation of assets with monetary value is required to claim the LZO tokens, users that receive larger airdrops but don't are not able to or don't want to pay for all of those tokens could be barred from claiming.

The functionality to allow for partial claims is already in place because a Merkle root could also be updated to increase a user's claim amount. If the **_claim** function in both ClaimLocal and ClaimRemote add a parameter **_claimAmount** that is checked to be smaller than **_zroAmount**, then the user would be able to perform a partial claim without any overhead.

```
function _claim(
    address _user,
    Currency _currency,
    uint256 _zroAmount,
    bytes32[] calldata _proof,
    address _to
) internal returns (MessagingReceipt memory receipt) {
    // step 1: assert user has donated sufficient amount
    assertDonation(_currency, _user, _zroAmount);

    // step 2: assert the proof
    if (!_verifyProof(_user, _zroAmount, _proof)) revert InvalidProof();

    // step 3: get availableToClaim
    uint256 availableToClaim = _availableToClaim(_user, _zroAmount);

    // step 4: transfer the ZRO tokens
    IERC20(zroToken).safeTransfer(_to, availableToClaim);

    emit Claim(_user, availableToClaim, _to);
}
```

LZ02-18

RISK OF USER ERROR WHEN STARGATENATIVE IS NOT SET

SEVERITY: Informational

PATH:

contracts/donate/DonateCore.sol:L63-L79

REMEDIATION:

The donate() function should revert whenever it's not a donation of native tokens but msg.value is still greater than 0, as shown in the following check:

```
if (_currency != Currency.Native || stargateNative == address(0)) {
  require(msg.value == 0);
}
```

STATUS: Fixed

DESCRIPTION:

In the **DonateCore::donate()** function, if the stargate address of a token is address(0), it will skip donations from that token and will not pull funds from users.
However, when **stargateNative** is address(0), this function does not return the native tokens (msg.value) back to the user. This poses a risk of losing funds for users if they mistakenly donate native tokens when **stargateNative** is not set. Additionally, the router contract (**DonateAndClaim**) also doesn't prevent this scenario.

```
function donate(Currency _currency, uint256 _amount, address _beneficiary)
external payable {
    if (_currency == Currency.USDC && stargateUsdc != address(0)) {
        tokenUsdc.safeTransferFrom(msg.sender, address(this), _amount);
        donations[stargateUsdc][_beneficiary] += _amount;
    } else if (_currency == Currency.USDT && stargateUsdt != address(0)) {
        tokenUsdt.safeTransferFrom(msg.sender, address(this), _amount);
        donations[stargateUsdt][_beneficiary] += _amount;
    } else if (_currency == Currency.Native && stargateNative != address(0))
{
        if (msg.value != _amount) revert InsufficientMsgValue();
        donations[stargateNative][_beneficiary] += _amount;
    } else {
        // sanity just in case somehow a different currency is somehow
passed
        revert UnsupportedCurrency(_currency);
    }

    emit Donated(_currency, msg.sender, _beneficiary, _amount);
}
```


hexens
×
LayerZero.