SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type: Private
Prepared for: Orderly Network
Prepared by: Sherlock
Lead Security Expert: 0xdeadbeef
Dates udited: June 20 - June 30, 2024
Prepared on: ugust 13, 2024

Introduction

Orderly Network is the permissionless liquidity layer for Web3 trading, featuring a shared orderbook across front-ends

Scope

Repository: OrderlyNetwork/vesting-contract

Branch: main

Commit: 0f7696185c3bd3135de7e1130e5ca7b9799769eb

Repository: OrderlyNetwork/omnichain-ledger

Branch: audit

Commit: fb0c093da7876968794f6b3a93adcc23b2f81077

Repository: OrderlyNetwork/oft-token

Branch: dev

Commit: 08dccdd381a5a922c39c7d3b8774b7dfd3142b0a

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. ll major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
4	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

KupiaSec
0xdeadbeef
cu5t0mPe0
ck
cocacola
aslanbek
Varun_05
0x_adi

Issue H-1: `Valor._doValor mission()` should not be a public function

Source: https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/30

Found by

0xdeadbeef, KupiaSec, ck, cocacola

Summary

The `Valor._doValorEmission()` function is public, meaning it can be called by anyone. If `_doValorEmission()` is invoked outside of the Staking.`updateValorVars()` function, it can lead to the `accValorPerShareScaled` value being incorrectly updated when updateValorVars() is subsequently called.

Vulnerability Detail

The `_doValorEmission()` function updates `totalValorEmitted` and `lastValorUpdateTimestamp`. However, since `_doValorEmission()` is a public function, it can be called by anyone, which could lead to unintended updates to these values.

https://github.com/OrderlyNetwork/omnichain-ledger/tree/fb0c093da7876968794f6b3a93adcc23b2f81077/omnichain-ledger/contracts/lib/Valor.sol#L127-L133

```
  function _doValorEmission() public whenNotPaused returns (uint256
↪  valorEmitted) {
    valorEmitted = _getValorPendingEmission();
    if (valorEmitted > 0) {
        totalValorEmitted += valorEmitted;
        lastValorUpdateTimestamp = block.timestamp;
    }
}
```

If `_doValorEmission()` is called outside of the `Staking.updateValorVars()` function, it can affect the calculation of `accValorPerShareScaled`, potentially leading to incorrect updates to this value.

https://github.com/OrderlyNetwork/omnichain-ledger/tree/fb0c093da7876968794f6b3a93adcc23b2f81077/omnichain-ledger/contracts/lib/Staking.sol#L114-L119

```
function updateValorVars() public whenNotPaused {
    uint256 valorEmission = _doValorEmission();
    if (valorEmission > 0) {
```

```
        accValorPerShareScaled =
↪   _getCurrent ccValorPerShareScaled(valorEmission);
    }
}
```

malicious user could exploit the interplay between the `_doValorEmission()` and `updateValorVars()` functions to indefinitely prevent the growth of `accValorPerShareScaled`. This could be done by calling `_doValorEmission()` immediately before `updateValorVars()` is invoked, which occurs every time staking balances change. Such an attack would lead to users receiving substantially less V LOR tokens than anticipated.

Impact

Users may receive significantly less V LOR tokens than they expect.

Code Snippet

https://github.com/OrderlyNetwork/omnichain-ledger/tree/fb0c093da7876968794f6b3a93adcc23b2f81077/omnichain-ledger/contracts/lib/Valor.sol#L127-L133

https://github.com/OrderlyNetwork/omnichain-ledger/tree/fb0c093da7876968794f6b3a93adcc23b2f81077/omnichain-ledger/contracts/lib/Staking.sol#L114-L119

Tool used

Manual Review

Recommendation

The `Valor._doValorEmission()` function should be a internal function.

```
-     function _doValorEmission() public whenNotPaused returns (uint256
↪   valorEmitted) {
+     function _doValorEmission() internal whenNotPaused returns (uint256
↪   valorEmitted) {
        valorEmitted = _getValorPendingEmission();
        if (valorEmitted > 0) {
            totalValorEmitted += valorEmitted;
            lastValorUpdateTimestamp = block.timestamp;
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/OrderlyNetwork/omnichain-ledger/commit/03c72de6d3e427a3da406e09b0e0fbf749ebc053

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: The most important invariant will be broken.

Source: https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/7

The protocol has acknowledged this issue.

Found by

cu5t0mPe0

Summary

Order tokens can violate key invariants through `transfer` or `transferFrom`.

Vulnerability Detail

Sherlock RE DME of Orderly Network : `The most important invariant: the sum of OFT version ORDER token on OFTs contracts (all L2s) == the ORDER balance of Order dapter on native OrderToken contract (Ethereum).`

When users hold order tokens on Ethereum, they can send order tokens to `Order dapter` via `transfer` or `transferFrom`. This increases the number of order tokens in the `Order dapter`, thus violating the aforementioned 'most important invariant'.

Impact

The most important invariant will be broken.

Code Snippet

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/oft-token/contracts/OrderToken.sol#L11-L18

Tool used

Manual Review

Recommendation

It is recommended to prohibit `transfer` and `transferFrom` to the `Order dapter` address or to restrict transfers to this address to whitelist users only.

Discussion

iamckn

Invalid as the reason for the invariant is to ensure there are sufficient locked tokens on Ethereum to match the L2s which is not broken.

cu5t0mPeo

escalate I disagree with the comment above.

1. The RE DME refers to the order token balance on the Order dapter, while the comment under this issue describes the order token balance for the entire Ethereum network
2. ccording to the Sherlock rules.
3. I also confirmed this again in the PT, and the sponsor's response was "yes".

sherlock-admin3

> escalate I disagree with the comment above.
>
> 1. The RE DME refers to the order token balance on the Order dapter, while the comment under this issue describes the order token balance for the entire Ethereum network
> 2. ccording to the Sherlock rules.
> 3. I also confirmed this again in the PT, and the sponsor's response was "yes".

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. fter that, the escalation becomes final.

cu5t0mPeo

To add, the lead judge has acknowledged this point. The discussion is in a private thread. If you need to see the specifics, I can invite you.

iamckn

If the statement of the invariant is followed exactly as it is stated, then the invariant can be broken and this qualifies as a medium.

On the other hand, the invariant as it is stated doesn't seem to have a medium, high or even low impact and may have been misquoted.

That said the rules state:

> The protocol team can use the RE DME (and only the RE DME) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity.

If there are any other factors to be considered, the head of judging can advice.

0xdeadbeef0x

There is absolutely no impact. Its not high, medium, low or unknown. Its None.

I agree with CK and believe the client wanted to make sure in this invariant that there are enough tokens locked in L1 to accommodate L2 tokens. Having more has no effect..

cu5t0mPeo

yes, it has no impact, but it break the invariant, which is mentioned in the RE DME. ccording to past judgments, such as https://github.com/sherlock-audit/2024-05-pooltogether-judging/issues/134 or https://github.com/sherlock-audit/2024-05-pooltogether-judging/issues/136, these examples also had no actual impact but clearly violated the invariant stated in the RE DME.

0xdeadbeef0x

Both items were not escalated and assessed by the HOJ.

I argue that the impact is known (None) and not unknown (complex, not verified/assessed).

lso notice that statement is in response to the following question:

```
Should potential issues, like broken assumptions about function behavior, be
↪  reported if they could pose risks in future integrations, even if they might
↪  not be an issue in the context of the scope? If yes, can you elaborate on
↪  properties/invariants that should hold?
```

Doesn't seem like this issue that `could pose risks in future integrations.`

Lets see HOJ interpretation. I would be surprised if such issue is rewarded with no impact at all.

cu5t0mPeo

No, it answers `be reported if they could pose risks in future integrations, even if they might not be an issue in the context of the scope,can you elaborate on properties/invariants that should hold?` nd intentionally mentions `The most important invariant` in the response.

0xdeadbeef0x

Yeah, we quoted this same piece :)

I think HOJ has all the context to determine.

cu5t0mPeo

Sorry, I didn't read it all, but according to Sherlock's rules, this qualifies as medium severity. If there are inaccuracies in the RE DME provided by the sponsor, they should be corrected during the competition. That's why I specifically confirmed this point again with the sponsor in the PT.

iamckn

I think this all comes down to an interpretation of the rule:

> The protocol team can use the RE DME (and only the RE DME) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity.

If the protocol specifies an invariant that logically doesn't have an impact or if they misquoted an invariant, does it still qualify as medium?

I initially judged this as invalid because the sponsor most likely didn't state the invariant accurately and it should have instead been: `the sum of OFT version ORDER token on OFTs contracts (all L2s) <= the ORDER balance of Order dapter on native OrderToken contract (Ethereum).`

We know what the impact would have been if the invariant had been stated correctly as above.

So does the unknown impact case still apply here if the consensus is that the invariant was probably misquoted. I believe if it doesn't matter then the issue is medium otherwise if that can be considered a mitigating factor then it would be invalid.

WangSecurity

Well, it looks like we're starting to speculate on what the sponsor intended to mean and if they misquoted anything, but we have to take that sentence as it is. Even though it may be the known risk, it wasn't submitted as such and the issue perfectly fits the given invariant. Hence, based on the following rule, it has to be valid:

> The protocol team can use the RE DME (and only the RE DME) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity

Planning to accept the escalation and validate the issue with medium severity. re the any duplicates.

P.S. I was waiting for the sponsor to clarify this part, that's why commenting here only now. But, I've got no answer.

cu5t0mPeo

> re the any duplicates.

no dup

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- cu5t0mPeo: accepted

Issue M-2: Backward fees are not sent to the ledger and there is no function to extract them from vault chain

Source: https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/17

Found by

0xdeadbeef, KupiaSec

Summary

s a fix to issue C-04 the protocol has added a feature to charge the user for the backwards fee: https://github.com/OrderlyNetwork/omnichain-ledger/commit/5f53faccc8be4e6254994cc9a6d4aa627d66e48f

However - the fee is not sent to the ledger and there is no function to extract them.

Vulnerability Detail

When sending a message from the vault chain to the ledger a backward fee is charged to facilitate the cost of sending a message back from the ledger chain to the vault chain: https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/omnichain-ledger/contracts/lib/OCCManager.sol#L105

```
    function vaultSendToLedger(OCCVaultMessage memory message) internal {
--------------------
        MessagingFee memory msgFee = MessagingFee(msg.value, 0);
--------------------

        uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType];

        (_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value:
↪   lzFee}(sendParam, msgFee, msg.sender);
--------------------
    }
```

By reducing the `payloadType2BackwardFee` from `msg.value`, `payloadType2BackwardFee` is kept in the contract instead of being sent to the ledger which needs it to perform the backwards message.

dditionally, there is no function that can use the fees or extract them.

Impact

1. Ledger would need to be funded by the protocol team in order to send messages.
2. No function to extract the fees to the protocol

I set this as a medium severity since `PoxyLedger` is an upgradable contract and the protocol could theoretically add a function to rescue the funds. However the fact remains that the ledger chain does not get these fees from the cross chain message and will not be able to send a backwards message if there are no funds present on the ledger contract.

If "bug fix due by contract upgrade" is not considered a valid approach - this should be considered ***HIGH***.

Code Snippet

Tool used

Manual Review

Recommendation

Send the backwards fee as part of the the LayerZero message using the `options`.

Discussion

iamckn

Probable high. Sponsor should confirm how they plan on accessing or using the fee

sherlock-admin3

> escalate This issue shouldn't be classified as high; it might be medium or invalid. The reason is based on the discussion in Discord's `Clarification of #35`.

You've deleted an escalation for this issue.

0x adi

> 2. No function to extract the fees to the protocol

This part of the issue is a known issue. Please see

Note: Link to the iscord thread mentioned in the escalation comment for the reference
https://discord.com/channels/812037309376495636/1258645927955271741

0xdeadbeef0x

This is not related to this - that is about ORDER/USDC that have not been claimed by users and in case the owner wants to migrate those funds.. not related at all to backward fees that are native and should be sent to ledger for transaction execution.

The two issues have no comparison. The mentioned issue is a user error in case of very rare migration. This issue is about the funds charged and **NEEDED** for the cross chain messages to be executed. It is not conditional and not user error.

0x adi

gree M-15 is related ERC20 withdrawal and this particular issue is related the withdrawal of native fee charged from users. Sorry for the mistake.

iamckn

From my perspective, the validity depends on whether an upgradable contract can be said to have funds permanently locked. The rules state:

1. The issue causes locking of funds for users for more than a week.
2. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity. dditional constraints related to the issue may decrease its severity accordingly.

Condition (1) will be met only if the upgradeablilty of the contract is not considerd. Condition (2) doesn't seem to be met.

I would therefore say medium validity if it is decided contract upgradeability is not a consideration. If it is a consideration, the issue could be low. The head of judging can advice on this.

0xdeadbeef0x

Hi,

Why are both conditions not met? It definitely impacts the availability of time-sensitive functions. Without the backwards fee the LZ messages would not be able to work. This means that users will not be able to withdraw ORDER, claim rewards, claim USDC and claim vesting.

iamckn

Correct me if I have this wrong, but isn't the gas (for use in the backward messages) set in the `buildOCCLedgerMsg` of `LedgerOCCManager` and forwarded for as part of the options?

```
function buildOCCLedgerMsg(OCCLedgerMessage memory message) internal view
↪  returns (SendParam memory sendParam) {
        /// build options
        uint8 _payloadType = message.payloadType;
        uint128 _dstGas = payloadType2DstGas[_payloadType];
        if (_dstGas == 0) {
            _dstGas = 2000000;
        }
        uint128 _oftGas = defaultOftGas;
        if (_oftGas == 0) {
            _oftGas = 2000000;
        }

        uint256 amount = message.token == LedgerToken.ORDER ?
↪  message.token mount : 0;

        bytes memory options =
↪  OptionsBuilder.newOptions().addExecutorLzReceiveOption(_oftGas,
↪  0).addExecutorLzComposeOption(0, _dstGas, 0);
        sendParam = SendParam({
            dstEid: chainId2Eid[message.dstChainId],
            to:
↪  bytes32(uint256(uint160(chainId2ProxyLedger ddr[message.dstChainId]))),
            amountLD: amount,
            min mountLD: amount,
            extraOptions: options,
            composeMsg: abi.encode(message),
            oftCmd: bytes("")
        });
    }
```

0xdeadbeef0x

Hi @iamckn

Thanks for digging into it. No, so the G S needed to execute the execution on LEDGER chain is defined here but no `value` is passed in the option

But the FEES needed to call back from the LEDGER chain to V ULT chain is charged on the V ULT chain but never passed to the LEDGER chain.

```
    function vaultSendToLedger(OCCVaultMessage memory message) internal {
--------------------
        MessagingFee memory msgFee = MessagingFee(msg.value, 0);
--------------------
```

```
        uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType];

        (_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value:
↪   lzFee}(sendParam, msgFee, msg.sender);
--------------------
    }
```

So the `payloadType2BackwardFee[message.payloadType];` should be added to the option `value` along side the gas estimate

cu5t0mPeo

hi,If @0xdeadbeef0x is correct, then the severity of issue #35 should be consistent with this issue, because both have mitigation methods that do not depend on contract upgrades. If this issue occurs, sponsors can send some native tokens to the target ledger to withdraw funds. (Please correct me if this mitigation method is incorrect.)

Issue #35 also has mitigation methods, but I canceled the escalation due to these mitigations. Therefore, I might cancel the escalation for this issue as well, upgrade #35 to high, or plan to upgrade this issue to medium instead of high.

I'm currently unsure whether setting the `value` parameter is necessary or if setting gas alone is sufficient, because I encountered this issue during the competition but didn't submit it due to this uncertainty.

0xdeadbeef0x

> then the severity of issue https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/35 should be consistent with this issue, because both have mitigation methods that do not depend on contract upgrades. If this issue occurs, sponsors can send some native tokens to the target ledger to withdraw funds. (Please correct me if this mitigation method is incorrect.)

No

1. This issue has funds at risk/frozen while 35 does not.
2. Sponsor can indeed send tokens to ledger chain at their own expense which is a loss of funds as it is calculated and charged from the user.

   Issue https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/35 also has mitigation methods, but I canceled the escalation due to these mitigations. Therefore, I might cancel the escalation for this issue as well, upgrade https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/35 to high, or plan to upgrade this issue to medium instead of high.

gain - the reason 35 is medium is because its a no funds lost/locked DOS while this one has funds at stake.

> I'm currently unsure whether setting the value parameter is necessary or if setting gas alone is sufficient, because I encountered this issue during the competition but didn't submit it due to this uncertainty.

Setting gas only determines the amount of fee needed to pay the executor to execute messages from V ULT -> LEDGER. It is paid through `lzFee`. However `payloadType2BackwardFee[message.payloadType]` is the amount thats needs to be paid at the LEDGER chain to execute messages from LEDGER -> V ULT which stays in the vault chain instead of being transferred to the ledger chain

cu5t0mPeo

Thank you for your explanation

> gain - the reason 35 is medium is because its a no funds lost/locked DOS while this one has funds at stake.

#35 may have a risk of trapped funds because if a user completes the staking operation using `OrderSafe.stakeOrder`, they will be unable to call `vaultSendToLedger` to send a message, causing the funds to be stuck.

cu5t0mPeo

nd the mitigation method is to set `payloadType2BackwardFee` to 0, but when I saw your issue, I found that even if `payloadType2BackwardFee` is set to 0, it still doesn't mitigate the problem. You also need to send native tokens to the target ledger to send the message correctly.

0xdeadbeef0x

Sure, thanks for digging into the issue.

> https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/35 may have a risk of trapped funds because if a user completes the staking operation using OrderSafe.stakeOrder, they will be unable to call vaultSendToLedger to send a message, causing the funds to be stuck.

Unfortunately `OrderSafe.stakeOrder` is out of scope of the contest and not part of the flow. It is used for tests as mentioned in the oft-token RE DME:

> Only OFT and dapter contracts are under mainly development, the OrderSafe(Relayer) and OrderBox(Relayer) contracts are used to test the Cross-Chain token transfer and message relay.

> nd the mitigation method is to set payloadType2BackwardFee to 0, but when I saw your issue, I found that even if payloadType2BackwardFee is

> set to 0, it still doesn't mitigate the problem. You also need to send native tokens to the target ledger to send the message correctly.

Yeah so in my issue I said that the mitigation to solve it should be to set the value as part of the options. Then the native token will be sent to the ledger so it can use it for paying the fee.

cu5t0mPeo

Thank you for your explanation yes, you are right. I am going to delete the escalation comment. If anyone else has doubts about this issue, please reconsider escalating it.

iamckn

> No, so the G S needed to execute the execution on LEDGER chain is defined here but no value is passed in the option

I am unsure whether that is correct.

From my understanding `addExecutorLzComposeOption` adds `_dstGas` to the message. This is the gas that should be used on the the destination chain and sent cross chain with the compose message. Therefore this is the gas that will be used on the Vault side for the backward messages.

If @0xdeadbeef0x is right and gas set in `addExecutorLzComposeOption` will not be used on the destination chain, then the sponsor should probably be informed of a similar issue in the V ULT -> LEDGER direction too. Let me investigate this further.

0xdeadbeef0x

Hi @iamckn!

So let me elaborate on the flow and flaw.

We are talking about the following flow `->B->` where is vault chain and `B` is ledger `->B`: The options are set as follows:

```
bytes memory options =
↪  OptionsBuilder.newOptions().addExecutorLzReceiveOption(_oftGas,
↪  0).addExecutorLzComposeOption(0, _dstGas, 0);
```

Notice that the value for the native token amount is `0` in the options. Therefore no native tokens will be sent to `B`.

Even though no native tokens are sent to `B` the user is charged with `payloadType2BackwardFee[message.payloadType]`. This value stays in and there is no withdraw function for it.

```
uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType];
(_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value: lzFee}(sendParam,
↪  msgFee, msg.sender);
```

`B->` : The options for the backwards message is:

```
bytes memory options =
↪  OptionsBuilder.newOptions().addExecutorLzReceiveOption(_oftGas,
↪  0).addExecutorLzComposeOption(0, _dstGas, 0);
```

nd the sending of the message is as follows:

```
function ledgerSendToVault(OCCLedgerMessage memory message) external payable
↪  onlyLedger {
    SendParam memory sendParam = buildOCCLedgerMsg(message);
    uint256 fee = estimateCCFeeFromLedgerToVault(sendParam);

    MessagingFee memory msgFee = MessagingFee(fee, 0);

    /// @dev test only
    _msgPayload = sendParam.composeMsg;
    _options = sendParam.extraOptions;

    (_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value: fee}(sendParam,
↪  msgFee, address(this));
}
```

Notice that `fee` is from the `B` contract balance.

Because the fee was not sent from `->B` there will be no funds available for `fee` therefore `B->` will revert and not work.

iamckn

Escalate

Having looked at this, I believe medium severity is more appropriate because of the following reasons:

The `payloadType2BackwardFee` is subtracted from the `msg.value` but not managed, which is true.

On the other hand, the `send` transaction will never succeed because there is a related issue that sets the fee as a value that doesn't have `payloadType2BackwardFee` subtracted #35. Both issues arise because of incorrectly managing the `payloadType2BackwardFee`.

- This issue talks about the fact that the `payloadType2BackwardFee` isn't sent along to the ledger but kept in the contract with no way to extract it.
- Issue #35 talks about the fact that sending the full fee without subracting the `payloadType2BackwardFee` causes a revert.

Both classes of issues result from an improper handling of how `payloadType2BackwardFee` is used. Iso to be noted is `payloadType2BackwardFee` will never be collected because of #35.

I believe that first the sponsor should clarify what their intended way of handling the `payloadType2BackwardFee` was.

In summary: the issue of funds being locked in the contract talked about in this issue will happen only if the recommendation of #35 is implemented (protocol could choose a different remediation path that manages the `payloadType2BackwardFee`). so note the rule on future issues which mentions that an issue that results from fixing another issue is not considered valid, which may be another consideration here.

This becomes an issue of whether the protocol needs to withdraw the `payloadType2BackwardFee` or do something else with it after fixing #35.

sherlock-admin3

> Escalate
>
> Having looked at this, I believe medium severity is more appropriate because of the following reasons:
>
> The `payloadType2BackwardFee` is subtracted from the `msg.value` but not managed, which is true.
>
> On the other hand, the `send` transaction will never succeed because there is a related issue that sets the fee as a value that doesn't have `payloadType2BackwardFee` subtracted #35. Both issues arise because of incorrectly managing the `payloadType2BackwardFee`.
>
> - This issue talks about the fact that the `payloadType2BackwardFee` isn't sent along to the ledger but kept in the contract with no way to extract it.
> - Issue #35 talks about the fact that sending the full fee without subracting the `payloadType2BackwardFee` causes a revert.
>
> Both classes of issues result from an improper handling of how `payloadType2BackwardFee` is used. Iso to be noted is `payloadType2BackwardFee` will never be collected because of #35.
>
> I believe that first the sponsor should clarify what their intended way of handling the `payloadType2BackwardFee` was.

> In summary: the issue of funds being locked in the contract talked about in this issue will happen only if the recommendation of #35 is implemented (protocol could choose a different remediation path that manages the `payloadType2BackwardFee`). so note the rule on future issues which mentions that an issue that results from fixing another issue is not considered valid, which may be another consideration here.
>
> This becomes an issue of whether the protocol needs to withdraw the `payloadType2BackwardFee` or do something else with it after fixing #35.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. fter that, the escalation becomes final.

0xdeadbeef0x

Hey,

Firstly - #35 and this issue are **NOT** the same class and do not share the same root cause/impact.

Issue #35:

- The root cause is an incorrect assignment of `msgFee` which should not include the backwards fee (since the fee is ONLY for the execution of `->B`.
- The impact of #35 is DOS and no funds at risk.

This issue:

- Root cause is that the backwards fee is not extractable/not sent to ledger.
- The impact of this issue is loss of funds as ledger messages need to covered by the protocol instead of the charged backwards fee.

Secondly - This issue does not depend on how a specific fix for #35 is implemented. ssuming #35 is fixed and the protocol behaves normally (`_lzSend` does not revert) this issue stays the same.

This issue is not a RESULT of a specific fix of another issue.

Think about it this way - #35 renders the contract useless. Following the claimed logic - no bugs in the protocol should be issued because #35 is already saying that every operations is halted and nothing will work in the first place.

I don't think thats the case. Lets leave it to HOJ

0x adi

Recommendation

Send the backwards fee as part of the the LayerZero message using the `options`.

IMO, the recommended mitigation will solve both the issues. (#35 , #17)

The contract should send the whole `msgFee` (without subtracting `payloadType2BackwardFee`) to the destination.

nd the root cause is also same, The `OCCManager` contract is not forwarding `payloadType2BackwardFee` to the destination.

0xdeadbeef0x

> IMO, the recommended mitigation will solve both the issues.

No - it will not solve issue 35 as the call will still revert due to an insufficient `lzFee`.

> The contract should send the whole msgFee (without subtracting payloadType2BackwardFee) to the destination.

That is one possibility, another is to include a "withdrawFees" function and send to ledger the fees. (which seems like the actual intent of the sponsor)

> nd the root cause is also same, The OCCManager contract is not forwarding payloadType2BackwardFee to the destination.

No - that is not the root cause of 35. The root cause of 35 is a mismatch between `msgFee` and `lzFee`

iamckn

In addition to my comments here https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/17#issuecomment-2212388882 I believe both classes of issues qualify as medium. They identified issues with the management of `payloadType2BackwardFee`.

s the contract is right now, the fund locking described in this issue does not occur because the issue described in #35 means that function always reverts. gain both are caused by not having proper management of the `payloadType2BackwardFee`. This is the reason why the sponsor's input may be important to explain how they planned to either manage/send/withdraw the fee.

Two scenarios:

- If the plan was to add withdraw functionality, #35 would still occur if no other changes are made.
- If issue #35 is fixed and no other change is done, the function would not revert and therefore `payloadType2BackwardFee` would be collected in the contract with no withdraw functionality.

I therefore believe both sets of issues should be classified as medium.

0xdeadbeef0x

Sorry I don't understand the argument in the comment and how this issue is related to #35 as mentioned here.

Is your claim for downgrading from High to Medium is that this issue could only cause impact if #35 is fixed?

iamckn

> Sorry I don't understand the argument in the comment and how this issue is related to #35 as mentioned here.
>
> Is your claim for downgrading from High to Medium is that this issue could only cause impact if #35 is fixed?

That is one consideration. Yes it can only cause a medium impact if #35 is resolved. If the future issues rule is considered, this could even be invalid.

Second reason is that even in isolation (i.e send operation doesn't revert), locking of fees qualifies it as a DOS therefore medium (this is also ignoring the fact that the specific contract can be upgraded with no side effects).

0xdeadbeef0x

I do not think the existence of one issue can block another - otherwise there would be no point in submitting any issues other then 35.

So maybe here is the misunderstanding regarding severity: There are funds at risk here from two scenarios:

1. In order to facilitate the ledger->vault messages - the protocol will need to add balance to the ledger contract **out of pocket**
2. If there is not enough balance in the ledger contract - users would not be able to withdraw ORDER, claim rewards, claim USDC and claim vesting.

I suggest to re-read all the comments. s seen - these fees are crucial for the protocol to facilitate ledger->chain messages. The user is charged with the fee and locking of the fees results in loss of funds. There are no pre-conditions, as such the severity should stay HIGH.

I think HOJ has enough info. Please let me know if more is needed.

iamckn

> In order to facilitate the ledger->vault messages - the protocol will need to add balance to the ledger contract out of pocket

This is the locking of funds issue. They are unable to claim the funds they use, making it a medium.

> If there is not enough balance in the ledger contract - users would not be able to withdraw ORDER, claim rewards, claim USDC and claim vesting.

There is a `receive() external payable {}` for the procotol to fund transactions. Therefore users would still be able to claim.

> I do not think the existence of one issue can block another

Those are the rules and despite that I say that it would be unfair to invalidate this and therefore believe medium is appropriate.

I also said even in isolation, the only issue is the locking funds DOS and medium.

0xdeadbeef0x

There are two cases of loss of funds

1. Someone will fund the ledger (extra out of pocket)
2. No one will fund the ledger and user cannot before critical operations that result in loss of funds (unable to unstakel, claim rewards, claim USDC and claim vesting)

   > This is the locking of funds issue. They are unable to claim the funds they use, making it a medium.

Not a medium - locking of funds that critical for the operation of the protocol should be high. dditionally, if the ledger is not funded - users will not receive rewards/his funds thus loss of funds

> There is a receive() external payable {} for the procotol to fund transactions. Therefore users would still be able to claim.

Yes, at the expense of funding the contract. While the fees are already charged.

> Those are the rules and despite that I say that it would be unfair to invalidate this and therefore believe medium is appropriate.

I do not see how those rules apply. This issue's root cause is not a result of a specific future fix implementation.

I think we are not reaching and agreement and would advise to wait for HOJ questions and settlement

iamckn

The ledger is fundable by the protocol through the `receive() external payable {}` which was specifically added for this. Issue C-04 of the known issues - https://chocolate-billboard-6fb.notion.site/Known-issues-and-fixes-e076499538974ab29a35760a605837b5?p=a861f11c1a9b4809ac4cbde7318e0ef9pm=s

This issue therefore doesn't not make unstaking broken. If the intention was to leave the fee in the contract, the only issue becomes that the fee is not correctly

subtracted and the function reverts due to #35.

If the head of judging needs any more clarification, I will provide it. I also suggest the sponsor specifically explains what the plan was with the fee.

0xdeadbeef0x

That is incorrect. The fix for C-04 was to add the charging mechanism on the vault chain: https://github.com/OrderlyNetwork/omnichain-ledger/commit/5f53faccc8be4e6254994cc9a6d4aa627d66e48f#diff-6e7cd07cac50bb67eceb9ad5d1327a35b76067152a142e1582c199beb89aeedc

```
fixed C-04 issue by charging user additional fee, for backward msg
```

The fee is charged but cannot be extracted to fund the ledger. I think its a good idea to get the sponsors view - however the intentions and impact are clear. Lets see if its clear for HOJ

WangSecurity

s I see from the report (not from the discussion), the impact is fees being stuck in the contract. But the contract is upgradeable so the fees can be extracted. The other part:

> Ledger would need to be funded by the protocol team in order to send messages

s I understand, due to fees not being forwarded, the functions requiring these fees will revert. Could you send me which functions will revert in that case?

The impact here would be just revert, correct? Could you clarify why in this case the fee wouldn't be returned to the caller, or the transaction will go through, fail, not revert, and fees will remain in the contract?

0xdeadbeef0x

Hi @WangSecurity

Sure: This function will revert on the ledger chain: https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/lib/LedgerOCCManager.sol#L119C1-L131C1

These are the function that call it: https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/OmnichainLedgerV1.sol#L168-L247

Essentially any operations to unstake, claim rewards, claim USDC and claim vesting will not work.

This means that users that stake, vest, etc.. (no need for backward fees to do that) will not be able to take their funds out of the ledger chain until its funded (need backward fees for that).

refunds are for excess fees sent and not for unsuccessful reverts.

iamckn

s per the current implementation, all user actions would revert and the issue of funds being locked wouldn't occur. They rely on this function succeeding:

```
(_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value: lzFee}(sendParam,
↪  msgFee, msg.sender);
```

The reason for the revert is that `msgFee` is used without subtracting the `payloadType2BackwardFee`.

The impact of revert is the basis for #35 which basically makes all user actions broken, which the protocol would have to fix. The transaction cannot go through until #35 is fixed.

0xdeadbeef0x

@WangSecurity maybe its important to clear out the rules regarding "this issue is not applicable because of issue 35".

Noting that every submitted issue in not applicable until issue 35 is fixed.

iamckn

I believe the distinction here is that `payloadType2BackwardFee` is the matter of contention for both issues. How `payloadType2BackwardFee` is managed determines whether it was meant to be sent as part of the transaction of be left in the contract. This issue claims that `payloadType2BackwardFee` should not be subtracted from `msgFee`(fixing locking of funds) while #35 claims it should be subtracted(preventing reverts).

The sponsor should therefore clarify what avenue they intended to use to manage `payloadType2BackwardFee` as it remains an issue of contention.

0xdeadbeef0x

> This issue claims that payloadType2BackwardFee should not be subtracted from msgFee

That is not what this issue claims. That is one possible mitigation. This issue simply claims that backward fees are not forwarded to the ledger and therefore backward operations from the ledger will not work unless someone pays out of pocket in additional to already charged fees. If the funds are sent via a withdraw function or through LZ protocol is irrelevant.

I also showed in this comment that its the clients intention to charge the user on the vault chain.

Issue 35 is an entirely different issue that the root cause is a mismatch of `msgFee` and `lzFee`.

iamckn

From this issue, this is the claim:

> By reducing the payloadType2BackwardFee from msg.value, payloadType2BackwardFee is kept in the contract instead of being sent to the ledger which needs it to perform the backwards message.

From issue #35, this is the claim:

> The vulnerability lies in the vaultSendToLedger function, which erroneously transfers the payloadType2BackwardFee[message.payloadType] to LayerZero along with the LayerZero fee.

These two issues claim the fee should be managed in a different way. The sponsor needs to clarify what they wanted to do with the fee.

From what I can see, the intention of the sponsor may have been to leave the fee in the contract as they explicitly do a subtraction:

```
uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType]
```

My argument is that both issues are medium.

This issue causes `payloadType2BackwardFee` to be left in the contract when #35 is fixed.

0xdeadbeef0x

Im not sure what we are arguing about.

Why does it matter what the "intent" is? (extract by withdraw function or send through LZ).

iamckn

Basically my claim is that we keep arguing about how `payloadType2BackwardFee` should be used. This issue claims that `payloadType2BackwardFee` should be sent while the protocol code indicates otherwise. If the fee was meant to be withdrawn, the impact is locking of fees because of lack of withdraw functionality.

Ignoring #35, what would the fixed code look like if the fees were to be sent in that function? How would the remediated code look like?. It doesn't make sense to send the fees crosschain and would be a wasteful design. There is no indication in the code that the fee was meant to be sent.

The lzcompose() options on both sides of the chain sets the correct amount of gas required for transactions to succeed which the sponsor already mentioned https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/20#issuecomment-2205570836.

> Why does it matter what the "intent" is? (extract by withdraw function or send through LZ).

- Basically I claim that the impact is medium if the protocols plan was to withdraw the fees because no other functionality would be affected.
- There is no indication anywhere that the intention was to send the fee.

0xdeadbeef0x

> This issue claims that payloadType2BackwardFee should be sent while the protocol code indicates otherwise. If the fee was meant to be withdrawn, the impact is locking of fees because of lack of withdraw functionality.

Indeed the protocol is missing the a withdraw function or ability to send through LZ. The impact however is not just locking of the fees but also the inability for ledger operations (unstaking, claim, etc..) to work (because of the lack of fees)

> what would the fixed code look like if the fees were to be sent in that function? How would the remediated code look like?. It doesn't make sense to send the fees crosschain and would be a wasteful design. There is no indication in the code that the fee was meant to be sent.

This is purely recommendation of how to fix the issue. IMO its not wasteful and would be more elegant/automatic/cheap to send it via LZ. However the protocol can decide just to add a withdrawFee function and send the funds to the ledger chain manually. The fix method does not impact the validity of this issue.

> The lzcompose() options on both sides of the chain sets the correct amount of gas required for transactions to succeed which the sponsor already mentioned https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/20#issuecomment-2205570836.

Yes, we assume the gas is correct. However thats not related to this issue. There will be no funds to pay for the gas on the ledger.

> Basically I claim that the impact is medium if the protocols plan was to withdraw the fees because no other functionality would be affected.

Why? If they plan and cannot withdraw - ledger operations (unstaking, claiming, etc..) will not be available.

> There is no indication anywhere that the intention was to send the fee.

s mentioned above - the method the fees arrive in the ledger chain is not important.

iamckn

> Indeed the protocol is missing the a withdraw function or ability to send through LZ. The impact however is not just locking of the fees but also the inability for ledger operations (unstaking, claim, etc..) to work (because of the lack of fees)

I claim that no other functions will be affected in a time sensitive manner. gas is funded through `lzcompose()` options and `payloadType2DstGas` which are not affected by this issue, again refer to https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/20#issuecomment-2205570836. Both sides of the chain have receive functionality for gas funding that were added after issue c-04 was raised here https://chocolate-billboard-6fb.notion.site/Known-issues-and-fixes-e076499538974ab29a35760a605837b5?p=a861f11c1a9b4809ac4cbde7318e0ef9pm=s

Basically all that is needed is that the contracts have funds and enough gas is allocated through the mapping `payloadType2DstGas` and `lzcompose()` options.

0xdeadbeef0x

> gas is funded through lzcompose() options and payloadType2DstGas which are not affected by this issue

Yes the amount of gas to successfully execute the call is assumed to be defined correctly. On the vault chain its paid through the `lzFee`. On the ledger chain its paid through ***contracts native TH balance***. The amount of native balance needed is defined in `payloadType2DstGas` (aka backwards fee). If its not sent to the ledger - the ledger will not have enough balance to pay for the ledger->chain vault message - please read my elaborated flor in this comment

> Basically all that is needed is that the contracts have funds and enough gas is allocated through the mapping payloadType2DstGas and lzcompose() options.

The problem here is that the contract will not have funds needed as defined in `payloadType2DstGas` unless paid out of pocket (instead of charged payloadType2DstGas) to the ledger contract. Therefore ledger unstake, claim, etc.. operations will fail.

iamckn

> The problem here is that the contract will not have funds needed as defined in payloadType2DstGas unless paid out of pocket (instead of charged payloadType2DstGas) to the ledger contract. Therefore ledger unstake, claim, etc.. operations will fail.

Indeed the contracts need to be funded by gas which the protocol will supply and the transactions will go through. This gas is equivalent to the amount they are unable to withdraw. So all in all the issue remains that they can't get their refund until the day they upgrade the contract therefore the medium validity.

0xdeadbeef0x

Ignoring upgradability - This issue has two possible impacts:

1. The protocol will need to add balance to the ledger contract ***out of pocket***
2. If there is not enough balance in the ledger contract - users would not be able to withdraw ORDER, claim rewards, claim USDC and claim vesting.

I think we should stop the discussion here and wait for HOJ because we added very minimal value in the last ±10 comments

iamckn

summary of my remarks:

I agree there needs to be enough balance in the contract. The contracts have receive functionality that was added specifically to be able to send native currency to the contracts. The funds locked now are owed to the protocol but everything works.

1. The issue causes locking of funds for users for more than a week.
2. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity. dditional constraints related to the issue may decrease its severity accordingly.

The second requirement is not met. The rule also allows for decrease of severity due to additional constraints (upgradability if considered a constraint would make this low) but I still believe medium is a fair assessment.

0xdeadbeef0x

summery of my defense:

The users are charged with the backward fee - however there is no way to extract them and send them to the ledger chain (without upgrading the implementation).

ssuming the protocol does not pay out of pocket instead of the charged fees - the two conditions are met - locking of funds and impact of time-sensitive functions.

If the protocol decides to pay out of pocket - the impact is direct loss of funds for the protocol.

This is not a conditional issue. The issue happens for every unstake and claiming operations

WangSecurity

Firstly, I'll clear the rule about future issues -- I believe it's not applicable here, cause this issue is not result of the future code change after fixing #35, the issue is still present in the current codebase.

Secondly, I believe the condition of lock of funds for a week is not met. The contract on the ledger chain can be funded with a fee and the users will get the funds, so the lock of funds can last even just for 2 blocks.

Thirdly, I believe the affected functions are not time-sensitive. Hence, I believe the second condition is not met.

Fourthly, we've got a problem of protocol fees being locked in the contract and cannot be withdrawn. But, the contracts are upgradeable, I disagree it makes the issue low severity, but I believe it indeed decreases the severity.

Planning to accept the escalation and decrease the severity of the issue to Medium.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- iamckn: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/OrderlyNetwork/omnichain-ledger/commit/03c72de6d3e427a3da406e09b0e0fbf749ebc053

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Mismatch between `lastValorUpdateTimestamp` and `valor missionStartTimestamp` during the `OmnichainLedgerV1.initialize()` function

Source: https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/34

Found by

KupiaSec, aslanbek

Summary

`lastValorUpdateTimestamp` is set to the current `block.timestamp`, but it is not set to the same value as `valorEmissionStartTimestamp`, which is set to `block.timestamp + 1` days. This mismatch leads to an incorrect calculation of the valor emission.

Vulnerability Detail

During the `OmnichainLedgerV1.initialize()` function, the `valorEmissionStartTimestamp` is set in the `valorInit()` function, while the `lastValorUpdateTimestamp` is set in the `stakingInit()` function. Specifically:

```
valorEmissionStartTimestamp = block.timestamp + 1 days;
```

```
lastValorUpdateTimestamp = block.timestamp;
```

s a result, `lastValorUpdateTimestamp` is 1 day earlier than `valorEmissionStartTimestamp`.

This mismatch leads to incorrect behavior during the first valor emission. In the `Valor._getValorPendingEmission()` function, the calculation of the newly emitted valor amount is based on the elapsed seconds since `lastValorUpdateTimestamp`. However, since `lastValorUpdateTimestamp` is 1 day earlier than `valorEmissionStartTimestamp`, the newly emitted valor amount is greater than expected, as if the emission had already started 1 day earlier.

```
uint256 secondsElapsed = block.timestamp - lastValorUpdateTimestamp;
```

The mismatch also occurs in the `Valor.setValorEmissionStartTimestamp()` function, as it only resets the `valorEmissionStartTimestamp`, but does not update the `lastValorUpdateTimestamp`.

Impact

Valor emission actually starts 1 days earlier than intended.

Code Snippet

https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/OmnichainLedgerV1.sol#L52-L61

https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/lib/Valor.sol#L80-L86

https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/lib/Valor.sol#L136-L144

https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/lib/Valor.sol#L108-L112

https://github.com/OrderlyNetwork/omnichain-ledger/blob/main/contracts/lib/Staking.sol#L80-L83

Tool used

Manual Review

Recommendation

`lastValorUpdateTimestamp` should be set to the same value as `valorEmissionStartTimestamp`.

```
    function stakingInit(address, uint256 _unstakeLockPeriod) internal
 ↪  onlyInitializing {
        unstakeLockPeriod = _unstakeLockPeriod;
-       lastValorUpdateTimestamp = block.timestamp;
+       lastValorUpdateTimestamp = block.timestamp + 1 days;
    }
```

```
    function setValorEmissionStartTimestamp(uint256
 ↪  _valorEmissionStartTimestamp) external whenNotPaused
 ↪  onlyRole(DEF ULT_ DMIN_ROLE) {
        if (block.timestamp > valorEmissionStartTimestamp) revert
 ↪  ValorEmission lreadyStarted();
        if (block.timestamp > _valorEmissionStartTimestamp) revert
 ↪  ValorEmissionCouldNotStartInThePast();
        valorEmissionStartTimestamp = _valorEmissionStartTimestamp;
+       lastValorUpdateTimestamp = _valorEmissionStartTimestamp;
    }
```

Discussion

iamckn

Sponsor to confirm whether this is intended

ElderJoy

I also found this issue when refactored tests. The fix is the same as proposed. But fix was made a bit later, than code freeze for audit and neither new. tests nor fix for this issue passed to the audit codebase. Thus the issue is valid and will have proposed fix.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/OrderlyNetwork/omnichain-ledger/commit/03c72de6d3e427a3da406e09b0e0fbf749ebc053

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: `ProxyLedger` operations will always revert due to mismatch of fee in the parameter and `ProxyLedger` provided to `_lzSend()`

Source: https://github.com/sherlock-audit/2024-05-orderly-network-judging/issues/35

Found by

0x adi, KupiaSec, Varun_05, ck, cu5t0mPe0

Summary

The `ProxyLedger` contract operations such as `claimReward`, `stakeOrder`, and `sendUserRequest` on the destination chain utilize the LayerZero protocol. However, discrepancies in the fee mentioned in `msg.value` and the `msgFee` parameter when calling `_lzSend()` will cause the transaction to revert.

Vulnerability Detail

The vulnerability lies in the `vaultSendToLedger` function, which erroneously transfers the `payloadType2BackwardFee[message.payloadType]` to LayerZero along with the LayerZero fee.

```
File: omnichain-ledger/contracts/lib/OCCManager.sol

    function vaultSendToLedger(OCCVaultMessage memory message) internal {
        if (message.token mount > 0) {
            address erc20Token ddr = IOFT(orderTokenOft).token();
            IERC20(erc20Token ddr).safeTransferFrom(message.sender,
↪   address(this), message.token mount);

            if (IOFT(orderTokenOft).approvalRequired()) {
                IERC20(erc20Token ddr).approve(address(orderTokenOft),
↪   message.token mount);
            }
        }

        SendParam memory sendParam = buildOCCVaultMsg(message);

@>>     MessagingFee memory msgFee = MessagingFee(msg.value, 0);

        /// @dev test only
        _msgPayload = sendParam.composeMsg;
```

```
        _options = sendParam.extraOptions;

@>>     uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType];

@>>     (_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value:
↪  lzFee}(sendParam, msgFee, msg.sender);
        chainedEventId += 1;
    }
```

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/omnichain-ledger/contracts/lib/OCCManager.sol#L87C5-L109C6

The `vaultSendToLedger` function transfers `lzFee` as `msg.value` but sets `msgFee` as the fee parameter to `IOFT(orderTokenOft).send()`.

```
File: oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oft/OFTCoreUpgr
↪  adeable.sol

      // @dev Sends the message to the LayerZero endpoint and returns the
↪  LayerZero msg receipt.
      msgReceipt = _lzSend(_sendParam.dstEid, message, options, _fee,
↪  _refund ddress);
```

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oft/OFTCoreUpgradeable.sol#L236C9-L237C89

Then `send()` calls `_lzSend()` to interact with the LayerZero `EndpointV2.send()`. Subsequently, `_lzSend()` calls `_payNative()` to pay the native fee associated with the message.

```
File: oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oapp/O ppSender
↪  Upgradeable.sol

        uint256 messageValue = _payNative(_fee.nativeFee);
```

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oapp/O_ppSenderUpgradeable.sol#L88

Here, the transaction reverts if `msg.value` is not equal to `_nativeFee`, meaning the `vaultSendToLedger` function transfers `lzFee` as `msg.value` and `msgFee` (sum of `lzFee + payloadType2BackwardFee[message.payloadType]`). Therefore, the transaction will revert due to this check.

```
    function _payNative(uint256 _nativeFee) internal virtual returns (uint256
↪  nativeFee) {
```

```
@>>     if (msg.value != _nativeFee) revert NotEnoughNative(msg.value);
        return _nativeFee;
    }
```

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OAppSenderUpgradeable.sol#L110C5-L113C6

Impact

`claimReward`, `stakeOrder`, and `sendUserRequest` transactions will always revert due to transferring a fee (as `msg.value`) that is greater than what LayerZero requires (mentioned in the parameters as `msgFee`) to process the transaction.

Code Snippet

https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/omnichain-ledger/contracts/lib/OCCManager.sol#L105C1-L107C108 https://github.com/sherlock-audit/2024-05-orderly-network/blob/main/oft-token/contracts/layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OAppSenderUpgradeable.sol#L110C5-L113C6

Tool used

Manual Review

Recommendation

Please update `vaultSendToLedger` as follows.

```
    function vaultSendToLedger(OCCVaultMessage memory message) internal {
        if (message.token mount > 0) {
            address erc20Token ddr = IOFT(orderTokenOft).token();
            IERC20(erc20Token ddr).safeTransferFrom(message.sender,
↪   address(this), message.token mount);

            if (IOFT(orderTokenOft).approvalRequired()) {
                IERC20(erc20Token ddr).approve(address(orderTokenOft),
↪   message.token mount);
            }
        }

        SendParam memory sendParam = buildOCCVaultMsg(message);

-       MessagingFee memory msgFee = MessagingFee(msg.value, 0);
```

```
        /// @dev test only
        _msgPayload = sendParam.composeMsg;
        _options = sendParam.extraOptions;

        uint256 lzFee = msg.value - payloadType2BackwardFee[message.payloadType];
+       MessagingFee memory msgFee = MessagingFee(lzFee, 0);

        (_msgReceipt, _oftReceipt) = IOFT(orderTokenOft).send{value:
↪   lzFee}(sendParam, msgFee, msg.sender);
        chainedEventId += 1;
    }
```

Discussion

iamckn

Setting this to medium because the impact is denial of service.

sherlock-admin3

> escalate s discussed in Discord's `Clarification of #35`, this issue should be classified as high if not considering an upgrade, while #17 should be considered medium or even invalid.

You've deleted an escalation for this issue.

0x adi

In addition to the escalation comment.

Since the DOS on `claimReward` and `sendUserRequest` operations like `CreateOrderUnstakeRequest`, `WithdrawOrder`, `EsOrderUnstake ndVest`, `ClaimVestingRequest`, `RedeemValor`, and `ClaimUsdcRevenue` can cause fund locks. lso, this make the contract is unusable.

Note: Link to the iscord thread mentioned in the escalation comment for the reference
https://discord.com/channels/812037309376495636/1258645927955271741

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/OrderlyNetwork/omnichain-ledger/commit/03c72de6d3e427a3da406e09b0e0fbf749ebc053

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.