Dola Protocol ETH
Audit Report

Wed Feb 07 2024

contact@scalebit.xyz https://twitter.com/scalebit_

ScaleBit

Dola Protocol ETH Audit Report

1 Executive Summary

1.1 Project Information

Description	A cross-chain liquidity aggregation protocol
Type	Crosschain Liquidity
Auditors	ScaleBit
Timeline	Mon Jan 15 2024 - Wed Feb 07 2024
Languages	Solidity
Platform	Ethereum
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/OmniBTC/DolaProtocolDev
Commits	b9a31d2ca73a51eb560af8160c3102ff4620ed89 6d7b43cbb5f85c67d3d0ebed56c23f5f6b489dea c9d577ecbcaa8b126760cb2cb6b1eeac16cd8620 93c2475d5a4cd4dd6255dc7d96ad2533ac9ad7cd

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
IERC2	ethereum/interfaces/IERC20.sol	3628b689e13321c5195555f64fdde551d9b4ad6e
LBY	ethereum/contracts/libraries/LibBytes.sol	1f98cb3573244587d42c94e14a04766aca4743b5
LPC	ethereum/contracts/libraries/LibPoolCodec.sol	c34cecb14940dda08ca2e97bfbcbb4e9c4abaa9c
LSC	ethereum/contracts/libraries/LibSystemCodec.sol	10d531672b65413509766bc9cfb3614b2efa6b85
LDT	ethereum/contracts/libraries/LibDolaTypes.sol	b68630c16e377b7a12fb9c5b346f93b95c88c3c6
LGC	ethereum/contracts/libraries/LibGovCodec.sol	481c203a71e89871fac03d2f807cc740c268328d
LDE	ethereum/contracts/libraries/LibDecimals.sol	b5700cb2eb02390ef8087872cb234043603338b1
DPO2	ethereum/contracts/omnipool/DolaPool.sol	088ab3a969a6a7e1baee549df124176248fa2a15
LAS	ethereum/contracts/libraries/LibAsset.sol	0dda6dfc927f4c1c793d5cef4740f785cfda28a2
IWAP	ethereum/interfaces/IWormholeAdapterPool.sol	b134f42c2967d8c6e5cf25ede11b1b6a002aac6a
IWO	ethereum/interfaces/IWormhole.sol	0ab0029ee77bdf73c4b7936cb520484b3282d8e3

LLC	ethereum/contracts/libraries/LibLendingCodec.sol	51d87d82fba9492538e16476eb22ff5475515681
LWAV	ethereum/contracts/libraries/LibWormholeAdapterVerify.sol	61cbdc216f6017c2209841874add0e32d7d2c44f
MUL	ethereum/contracts/dolaportal/Multicall.sol	0cf42f8bd511009c6b97525ad094763a1ee3e417
SYS2	ethereum/contracts/dolaportal/System.sol	73a3d212c1f26b8213960e64cb5f9e39b7745b40
LEN2	ethereum/contracts/dolaportal/Lending.sol	fb9fcd2ab046a596d28c3ea4c7c14ea0cdf7ce36
WAP2	ethereum/contracts/omnipool/WormholeAdapterPool.sol	7c2621a21d0db515c84155fdc3587728ba9ff5f3

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	23	17	6
Informational	0	0	0
Minor	8	8	0
Medium	3	3	0
Major	12	6	6
Critical	0	0	0

1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow
- Number of rounding errors
- Unchecked External Call
- Unchecked CALL Return Values
- Functionality Checks
- Reentrancy
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic issues
- Gas usage
- Fallback function usage
- tx.origin authentication
- Replay attacks
- Coding style issues

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Dola Protocol ETH to identify any potential issues and vulnerabilities in the source code of the Dola Protocol ETH smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 23 issues of varying severity, listed below.

ID	Title	Severity	Status
DPO-1	Spender can Bypass the Warmhole Checks to Register New Spenders and then Withdraw Funds from the Protocol	Major	Acknowledged
DPO-2	When the Supply Limit is Reached on the Sui Chain, Funds Supplied by Users on the Source Chain may be Locked for A Period of Time	Major	Acknowledged
LAS-1	Revert on Large Approvals & Transfers	Major	Fixed
LAS-2	Not Supported for FEE-ON-TRANSFER Tokens	Major	Acknowledged
LAS-3	`isContract` Unsafe	Medium	Fixed
LAS-4	`SafeApprove` Deprecated	Medium	Fixed
LDE-1	When Transferring Assets Cross-Chain From High Precision to Low Precision, the Loss of Precision Can Result in Asset Loss for Users	Major	Acknowledged

LEN-1	The Required Fee is not being Passed When Calling `sendMessage()`	Major	Fixed
LEN-2	Unchecked Fee Transferred to the Relayer	Major	Acknowledged
LEN-3	The Pause Functionality is Necessary to Address Emergency Situations	Major	Fixed
LEN-4	Use Calldata Instead of Memory for Function Arguments That Do not Get Mutated	Minor	Fixed
LEN-5	Using Private rather than Public for Constants, Saves Gas	Minor	Fixed
LWA-1	Using Bools for Storage Incurs Overhead	Minor	Fixed
MUL-1	For Operations That will not Overflow, You could Use Unchecked	Minor	Fixed
MUL-2	`++i` Costs Less Gas Than `i++`, Especially When It's Used in For-loops (`--i/i--` Too)	Minor	Fixed
ORA-1	If `price.expo > 0`, Wrong Prices will be Calculated	Major	Fixed
ORA-2	The Oracle Report Lacks Checks for Price being 0 and Confidence being 0	Medium	Fixed
WAP-1	If `msg.value` is greater than `wormholeFee`, it could lead to a loss of funds	Major	Fixed

WAP-2	No Access Control for `WormholeAdapterPool.sendMessage()`	Major	Fixed
WAP-3	Financial Losses Caused by The Account Abstraction Wallet	Major	Acknowledged
WAP-4	Missing 0 Address Check	Minor	Fixed
WAP-5	Lack of Restriction in the `removeRelayer()` Function Raises Concerns about Emptying `Relayers`	Minor	Fixed
WAP-6	`require()` / `revert()` Statements Should Have Descriptive Reason Strings	Minor	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the Dola Protocol ETH Smart Contract :

Governance

- Governance can call `registerSpender` to add new spenders to the Dola Pool.
- Governance can use `deleteSpender` to remove spenders from the Dola Pool.
- Governance can invoke `registerRelayer` to register new relayers.
- Governance can utilize `removeRelayer` to deregister existing relayers.

Applications

- Applications can use the `sendDeposit` function to deposit assets into the Dola Pool.
- Applications can call `sendMessage` to send messages through the Wormhole network that do not involve any incoming or outgoing funds.

Relayer

- Only registered relayers can use `receiveWithdraw` to process withdrawal requests from the Dola Pool.

User

- Users can utilize the `aggregate` function to execute multiple calls within a single transaction.
- Users can call the `blockAndAggregate` function executing multiple calls in one transaction and returns the current block's hash and number for blockchain state verification.
- Users can call the `binding` function to establish a link between different Dola chains.
- Users can use the `unbinding` function to sever existing links between Dola chains.
- Users can utilize the `supply` function to deposit assets into the contract for cross-chain operations.
- Users can call the `withdraw` function to retrieve assets from a cross-chain pool.
- Users can call the `borrow` function to take out loans from the cross-chain pool.
- Users can use the `repay` function to settle their previous loans.
- Users can call the `liquidate` function to liquidate bad loans.

- Users can utilize the `as_collateral` function to use their assets as collateral.
- Users can use the `cancel_as_collateral` function to cancel their assets previously set as collateral.
- Users can call the `sponsor` function to deposit assets in support of the platform, distinct from lending or borrowing actions.
- Users can use the `claim_reward` function to claim rewards from specific pools.

4 Findings

DPO-1 Spender can Bypass the Warmhole Checks to Register New Spenders and then Withdraw Funds from the Protocol

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/omnipool/DolaPool.sol#40-44;

ethereum/contracts/omnipool/DolaPool.sol#47-63

Descriptions:

The `DolaPool.registerSpender()` function has an `isSpender()` modifier, meaning any spender can add new spenders.

```
function registerSpender(address newSpender) public isSpender(msg.sender) {
    require(spenders[newSpender] == 0, "HAS REGISTER SPENDER");
    allSpenders.push(newSpender);
    spenders[newSpender] = allSpenders.length;
}
```

If the owner adds a new spender, the new spender can directly call `registerSpender()` to add another spender, and then call `DolaPool.withdraw()` to take all the funds from the protocol.

```
function withdraw(
    LibDolaTypes.DolaAddress memory userAddress,
    uint64 amount,
    LibDolaTypes.DolaAddress memory poolAddress
) public isSpender(msg.sender) {
    address pool = LibDolaTypes.dolaAddressToAddress(poolAddress);
    address user = LibDolaTypes.dolaAddressToAddress(userAddress);
    uint256 fixedAmount = LibDecimals.restoreAmountDecimals(
        amount,
        LibAsset.queryDecimals(pool)
    );
    require(userAddress.dolaChainId == dolaChainId, "INVALID DST CHAIN");
    LibAsset.transferAsset(pool, payable(user), fixedAmount);
```

```
    emit WithdrawPool(pool, user, fixedAmount);
}
```

The same permission issue exists with `deleteSpender()`.

Suggestion:

It is recommended that only the owner or `WormholeAdapterPool` can call these functions.

DPO-2 When the Supply Limit is Reached on the Sui Chain, Funds Supplied by Users on the Source Chain may be Locked for A Period of Time

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/omnipool/DolaPool.sol#75

Descriptions:

When users call `Lending.supply()` to transfer assets from the Ethereum chain to Sui, the protocol transfers all assets to DolaPool and then sends a cross-chain message.

```
function sendDeposit(
      address token,
      uint256 amount,
      uint16 appId,
      bytes memory appPayload
    ) external payable returns (uint64) {
      uint256 wormholeFee = wormhole.messageFee();
      require(msg.value >= wormholeFee, "FEE NOT ENOUGH");
      // Deposit assets to the pool and perform amount checks
      LibAsset.depositAsset(token, amount);
      if (!LibAsset.isNativeAsset(token)) {
          LibAsset.safeApproveERC20(IERC20(token), address(dolaPool), amount);
      }

      bytes memory payload = dolaPool.deposit{value: msg.value - wormholeFee}(
          token,
          amount,
          appId,
          appPayload
      );
      return
          wormhole.publishMessage{value: wormholeFee}(
              0,
              payload,
              involveFundConsistency
```

```
        );
    }
```

Subsequently, when executing `wormhole_adapter.supply()` on the Sui chain, there's a ceiling. When this ceiling is reached, the protocol prohibits further deposits, causing the cross-chain message to fail.

```
    storage::ensure_user_info_exist(storage, clock, dola_user_id);
    assert!(storage::exist_reserve(storage, dola_pool_id), EINVALID_POOL_ID);
    assert!(not_reach_supply_ceiling(storage, dola_pool_id, supply_amount),
EREACH_SUPPLY_CEILING);
    boost::boost_pool(storage, dola_pool_id, dola_user_id,
lending_codec::get_supply_type(), clock);
```

Users' assets remain locked on the Ethereum chain until they can be supplied on the target chain. If users happen to notice an investment opportunity on the target chain during this time but are unable to transfer their assets, they miss out on this investment opportunity.

Suggestion:

It is recommended to prevent users from supplying on the source chain when the supply limit is reached on the target chain.

Resolution:

Perform front-end validation for the supply ceiling and promptly raise the limit through governance.

LAS-1 Revert on Large Approvals & Transfers

Severity: Major

Status: Fixed

Code Location:

ethereum/contracts/libraries/LibAsset.sol#58

Descriptions:

In the `maxApproveERC20()` function, if `allowance < amount`, the protocol will perform an approve operation for spender with the value set to `MAX_INT`.

```
function maxApproveERC20(
    IERC20 assetId,
    address spender,
    uint256 amount
) internal {
    if (address(assetId) == NATIVE_ASSETID) return;
    if (spender == NULL_ADDRESS) revert("NullAddrIsNotAValidSpender");
    uint256 allowance = assetId.allowance(address(this), spender);
    if (allowance < amount)
        SafeERC20.safeApprove(IERC20(assetId), spender, MAX_INT);
}
```

However, some tokens (e.g., UNI, COMP) revert if the value passed to approve or transfer is larger than uint96.

Suggestion:

It is recommended to approve a specified amount of tokens as needed.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

LAS-2 Not Supported for FEE-ON-TRANSFER Tokens

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/libraries/LibAsset.sol#111

Descriptions:

In the `depositAsset()` function, if the token is not the native token, the protocol records the protocol's balance `_fromTokenBalance` before transferring from the user. Then, it calls `LibAsset.transferFromERC20()` to execute the transfer from the user. Finally, it calculates the protocol's balance `LibAsset.getOwnBalance()`, and the protocol's balance subtracted by `_fromTokenBalance` should equal `amount`.

```
uint256 _fromTokenBalance = LibAsset.getOwnBalance(tokenId);
    LibAsset.transferFromERC20(
        tokenId,
        msg.sender,
        address(this),
        amount
    );
    if (LibAsset.getOwnBalance(tokenId) - _fromTokenBalance != amount)
        revert("InvalidAmount");
}
```

However, some tokens take a transfer fee(e.g. STA,PAXG) may result in receiving fewer tokens than expected. In this case, the verification will fail.

```
    if (LibAsset.getOwnBalance(tokenId) - _fromTokenBalance != amount)
        revert("InvalidAmount");
```

Suggestion:

It is recommended to consider the use cases of tokens that charge a transfer fee.

LAS-3 `isContract` Unsafe

Severity: Medium

Status: Fixed

Code Location:

ethereum/contracts/libraries/LibAsset.sol#147-155

Descriptions:

This function utilizes `extcodesize` to obtain the length of the bytecode (runtime) stored at an address. If the length is greater than zero, it is identified as a contract; otherwise, it is considered an Externally Owned Account (EOA). However, there is a problem with this approach. During the creation of a contract, the runtime bytecode is not yet stored at the address, resulting in a bytecode length of zero. This means that if we place our logic within the contract's constructor, it is possible to circumvent the isContract() check.

```
function isContract(address contractAddr) internal view returns (bool) {
    uint256 size;
    // solhint-disable-next-line no-inline-assembly
    assembly {
        size := extcodesize(contractAddr)
    }
    return size > 0;
}
```

Suggestion:

It is recommended to use `tx.origin == msg.sender` to check if the caller is a contract.

Resolution:

This issue has been fixed. The client has already implemented the check `tx.origin == msg.sender` to determine if the caller is a contract.

LAS-4 `SafeApprove` Deprecated

Severity: Medium

Status: Fixed

Code Location:

ethereum/contracts/libraries/LibAsset.sol#58

Descriptions:

The OpenZeppelin SafeERC20 `safeApprove()` function has been deprecated, as seen in the comments of the OpenZeppelin code. Using this deprecated function can lead to unintended reverts and potentially the locking of funds.

Suggestion:

It is recommended to replace `safeApprove()` with `safeIncreaseAllowance()`.

Resolution:

This issue has been fixed. The client has already replaced `SafeApprove` with `safeIncreaseAllowance`.

LDE-1 When Transferring Assets Cross-Chain From High Precision to Low Precision, the Loss of Precision Can Result in Asset Loss for Users

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/libraries/LibDecimals.sol#13

Descriptions:

In the `DolaPool.deposit()` function, the protocol converts the precision of the user's cross-chain assets to 8 digits. The conversion method is as follows: if the precision of the user's asset on the ETH chain is greater than 8 digits, `fixedAmount = uint64(amount / (10**(decimals - 8)))` .

```
function fixAmountDecimals(uint256 amount, uint8 decimals)
        internal
        pure
        returns (uint64)
    {
        uint64 fixedAmount;
        if (decimals > 8) {
            fixedAmount = uint64(amount / (10**(decimals - 8)));
        } else if (decimals < 8) {
            fixedAmount = uint64(amount * (10**(8 - decimals)));
        } else {
            fixedAmount = uint64(amount);
        }
        require(fixedAmount > 0, "Fixed amount too low");
        return fixedAmount;
    }
```

Note that this is rounded down. For example, if a user wants to transfer 3999999999999999999 WETH from the ETH chain to the Aptos chain, `fixedAmount = 399999999` , where the precision of WETH on the Aptos chain is 8 digits. If the user then transfers these WETH from the Aptos chain back to the ETH chain, the final converted result will be 3999999990000000000, resulting in a loss of 999999999 wei. While the loss may seem

small for a single transaction, it can accumulate significantly with multiple users and transactions.

Suggestion:

It is recommended to charge a certain fee when crossing from the source chain to the target chain and round up if division doesn't result in a whole number `fixedAmount = uint64(amount / (10**(decimals - 8))) + 1` .

Resolution:

Take measures from the frontend to limit.

LEN-1 The Required Fee is not being Passed When Calling `sendMessage()`

Severity: Major

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Lending.sol#127

Descriptions:

In the `Lending.withdraw()` function, the protocol calls `WormholeAdapterPool.sendMessage()` to send a message to the `Wormhole`. However, `msg.value` is not passed when calling `sendMessage()`.

```
uint64 sequence = IWormholeAdapterPool(wormholeAdapterPool).sendMessage(
    LENDING_APP_ID,
    appPayload
);
```

Inside the `WormholeAdapterPool.sendMessage()` function, it is required that `msg.value >= wormholeFee`, which causes the `withdraw()` call to fail.

```
function sendMessage(uint16 appId, bytes memory appPayload)
    external
    payable
    returns (uint64)
{
    uint256 wormholeFee = wormhole.messageFee();
    require(msg.value >= wormholeFee, "FEE NOT ENOUGH");
    bytes memory payload = dolaPool.sendMessage(appId, appPayload);
    return
        wormhole.publishMessage{value: msg.value}(
            0,
            payload,
            notInvolveFundConsistency
        );
}
```

Similarly, `borrow()`, `repay()`, `liquidate()`, `as_collateral()`, `cancel_as_collateral()`, and `claim_reward()` do not pass the fee.

Suggestion:

It is recommended to pass `wormholeFee` when calling `sendMessage()`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

LEN-2 Unchecked Fee Transferred to the Relayer

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/dolaportal/Lending.sol#55

Descriptions:

Several functions in the Lending contract, such as `supply()`, `withdraw()`, and `borrow()`, specify a fee parameter, which is ultimately transferred to the relayer.

```
    function supply(
        address token,
        uint256 amount,
        uint256 fee
    ) external payable {
        uint64 nonce = IWormholeAdapterPool(wormholeAdapterPool).getNonce();
        uint64 fixAmount = LibDecimals.fixAmountDecimals(
            amount,
            LibAsset.queryDecimals(token)
        );
......
......
        LibAsset.transferAsset(address(0), payable(relayer), fee);
```

However, there is no validation for the fee parameter, allowing it to be set to 0 and thus avoiding fees.

Suggestion:

It is recommended to validate the fee.

Resolution:

The relay fee is complex and cannot be dynamically evaluated. Consider introducing the option for users to supplement the relay fee during later operations.

LEN-3 The Pause Functionality is Necessary to Address Emergency Situations

Severity: Major

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Lending.sol#62-87

Descriptions:

If a token depegs, causing tokens to move erratically across various chains, as seen in previous incidents like LUA and USDC USDT events, or if there's a security issue with the Wormhole preventing it from functioning properly, the protocol needs to enact a pause to prevent users from executing corresponding operations.

Suggestion:

It is recommended to implement a `whenNotPaused` modifier and apply it to the respective functions.

Resolution:

This issue has been fixed, and the protocol has implemented a pause functionality.

LEN-4 Use Calldata Instead of Memory for Function Arguments That Do not Get Mutated

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Lending.sol#110

Descriptions:

Mark data types as `calldata` instead of memory where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies memory storage.

```
bytes memory token,
bytes memory receiver,

bytes memory token,

bytes memory receiver,
```

Suggestion:

It is recommended to use `calldata` instead of `memory`.

Resolution:

This issue has been fixed. The client has already used `calldata` instead of `memory`.

LEN-5 Using Private rather than Public for Constants, Saves Gas

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Lending.sol#13

Descriptions:

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves 3406-3606 gas in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

```
uint8 public constant LENDING_APP_ID = 1;
```

Suggestion:

It is recommended to use private rather than public for constants.

Resolution:

This issue has been fixed. The client has already used private rather than public for constants.

LWA-1 Using Bools for Storage Incurs Overhead

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/libraries/LibWormholeAdapterVerify.sol#39

Descriptions:

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (20000 gas) when changing from `false` to `true`, after having been `true` in the past. See source.

```
mapping(bytes32 => bool) storage consumedVaas,

mapping(bytes32 => bool) storage consumedVaas,
```

Suggestion:

It is recommended to use `uint256(1)` and `uint256(2)` for true/false.

Resolution:

This issue has been fixed. The client has already used `uint256(1)` and `uint256(2)` for true/false.

MUL-1 For Operations That will not Overflow, You could Use Unchecked

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Multicall.sol#23

Descriptions:

For Operations that will not overflow, you could use unchecked.

```
for (uint256 i = 0; i < calls.length; i++) {
```

Suggestion:

It is recommended to use Solidity's `unchecked` block to save the overflow checks.

Resolution:

This issue has been fixed. The client has already used Solidity's `unchecked` block to save the overflow checks.

MUL-2 `++i` Costs Less Gas Than `i++`, Especially When It's Used in For-loops (`--i/i--` Too)

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/dolaportal/Multicall.sol#23

Descriptions:

The gas cost of `++i` is lower than `i++`, particularly when utilized in for-loops (`--i/i--` as well).

```
for (uint256 i = 0; i < calls.length; i++) {
```

Suggestion:

It is recommended to use `++i` instead of `i++`.

Resolution:

This issue has been fixed. The client has already used `++i` instead of `i++`.

ORA-1 If `price.expo > 0`, Wrong Prices will be Calculated

Severity: Major

Status: Fixed

Code Location:

sui/dola_protocol/sources/oracle/oracle.move#354-363

Descriptions:

When a user calls `Lending.supply()` on the ETH chain to send a cross-chain message, upon execution of `logic.execute_supply()` on Sui, the protocol updates the average liquidity of the user.

```
    update_interest_rate(pool_manager_info, storage, dola_pool_id, 0);
    update_average_liquidity(storage, oracle, clock, dola_user_id);
```

In the `update_average_liquidity()` function, the protocol calculates the value of the user's collateral by multiplying the quantity of collateral by the price. The calculation method is as follows.

```
  public fun calculate_value(oracle: &mut PriceOracle, dola_pool_id: u16, amount: u256):
u256 {
    let (price, decimal, _) = oracle::get_token_price(oracle, dola_pool_id);
    amount * price / (sui::math::pow(10, decimal) as u256)
  }
```

In the oracle, the price update is performed.

```
  fun update_price(price: &mut Price, pyth_price: &price::Price, current_timestamp: u64)
{
    let price_value = pyth::price::get_price(pyth_price);
    let price_value = i64::get_magnitude_if_positive(&price_value);
    let expo = pyth::price::get_expo(pyth_price);
    let expo = i64::get_magnitude_if_negative(&expo);

    price.value = (price_value as u256);
    price.decimal = (expo as u8);
```

```
        price.last_update_timestamp = current_timestamp;
    }
```

There are two issues here:

1. `expo` may be positive. In such cases, executing `let expo = i64::get_magnitude_if_negative(&expo)` will fail.

2.If `expo` is positive, then the calculation of the user's collateral value should be `amount * price * 10^expo`.

https://docs.pyth.network/price-feeds/solana-price-feeds/best-practices

https://github.com/pyth-network/pyth-sdk-rs/blob/main/examples/sol-anchor-contract/programs/sol-anchor-contract/src/lib.rs#L53-L56

Suggestion:

It is recommended to calculate the price as `price * 10^expo` if `expo` is greater than 0.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

ORA-2 The Oracle Report Lacks Checks for Price being 0 and Confidence being 0

Severity: Medium

Status: Fixed

Code Location:

sui/dola_protocol/sources/oracle/oracle.move#354-363

Descriptions:

When executing cross-chain requests from the ETH chain to SUI, such as withdraw, borrow, and liquidate, the protocol utilizes Pyth Oracle quotes for collateral evaluation. However, the protocol fails to consider the scenario where the price from the oracle is 0 when updating the oracle price, leading to unexpected outcomes. For example, this quote may sometimes have a price of 0. https://pyth.network/price-feeds/equity-us-aapl-usd

```
    fun update_price(price: &mut Price, pyth_price: &price::Price, current_timestamp: u64)
{
        let price_value = pyth::price::get_price(pyth_price);
        let price_value = i64::get_magnitude_if_positive(&price_value);
        let expo = pyth::price::get_expo(pyth_price);
        let expo = i64::get_magnitude_if_negative(&expo);

        price.value = (price_value as u256);
        price.decimal = (expo as u8);
        price.last_update_timestamp = current_timestamp;
    }
```

Suggestion:

It is recommended to revert when the price is 0 or the Confidence is 0.

Resolution:

This issue has been fixed. The protocol has added price validation.

WAP-1 If `msg.value` is greater than `wormholeFee`, it could lead to a loss of funds

Severity: Major

Status: Fixed

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#205-219

Descriptions:

The logic for the `msg.value` check is not adequately reasoned, particularly in scenarios where users are depositing ERC20 tokens. The condition `msg.value >= wormholeFee` is not entirely appropriate in this context, as it may lead users to transfer more NativeAsset than the required `wormholeFee`. This issue is also present in the `sendMessage` function. For a better understanding of how this should be handled, reference can be made to the Wormhole project's example code, specifically located at HelloWorld.sol line 61.

```
function sendMessage(uint16 appId, bytes memory appPayload)
    external
    payable
    returns (uint64)
{
    uint256 wormholeFee = wormhole.messageFee();
    require(msg.value >= wormholeFee, "FEE NOT ENOUGH");
    bytes memory payload = dolaPool.sendMessage(appId, appPayload);
    return
      wormhole.publishMessage{value: msg.value}(
        0,
        payload,
        notInvolveFundConsistency
      );
}
```

Additional, the protocol fee is passed as `msg.value` in the `sendMessage` function, and it should be wormholeFee.

Suggestion:

It is recommended to use the condition `msg.value == wormholeFee` in the smart contract code, especially when handling the depositing of ERC20 tokens and in the `sendMessage` function. Additional, replace `msg.value` with `wormholeFee`.

Resolution:

This issue has been fixed. The client has already fixed it according to the recommended method.

WAP-2 No Access Control for `WormholeAdapterPool.sendMessage()`

Severity: Major

Status: Fixed

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#205

Descriptions:

`WormholeAdapterPool.sendMessage()` lacks any permission control, allowing anyone to call this function with malicious payloads. As this is a cross-chain protocol, there may be future integrations where protocols integrate the Dola Pool protocol, allowing users to interact with third-party protocols integrated with Dola Pool. When a user calls a function of a third-party protocol, the protocol can construct a `WithdrawPayload` within the function, set the receiver to its own address, and then call `sendMessage()` to send a cross-chain message. In the `sendMessage()` function, the protocol will use `tx.origin` as the sender. When executed on the Sui chain, the protocol on Sui will transfer funds from the sender address to the receiver. Similarly, it can borrow assets under a different identity.

```
function sendMessage(uint16 appId, bytes memory appPayload)
    external
    payable
    returns (uint64)
{
    uint256 wormholeFee = wormhole.messageFee();
    require(msg.value >= wormholeFee, "FEE NOT ENOUGH");
    bytes memory payload = dolaPool.sendMessage(appId, appPayload);
    return
        wormhole.publishMessage{value: msg.value}(
            0,
            payload,
            notInvolveFundConsistency
        );
}
```

Suggestion:

It is recommended to implement access control.

Resolution:

This issue has been fixed. The protocol restricts only portals from making the call.

WAP-3 Financial Losses Caused by The Account Abstraction Wallet

Severity: Major

Status: Acknowledged

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#180-225

Descriptions:

In the context of a `supply` function, users deposit a certain amount of tokens and then send a cross-chain message. This message includes `tx.origin`, which is used by contracts on the Sui to record the initiator of the transaction. In other words, `tx.origin` records the address that deposited the funds. However, when users employ an account abstraction wallet, the caller is the user's AA Wallet, meaning the `msg.sender` address is the user's AA Wallet address (like `SimpleAccount`). But `tx.origin` is the address of the bundler in the account abstraction, which is the true initiator of the transaction.

When multiple users use the same account abstraction wallet, they may have different `SimpleAccounts`, but they share a common bundler. This means that the funds deposited by different users are recorded under the address of the bundler. This is problematic because it implies that the funds deposited by different users are indistinguishably linked to the bundler address. As a result, after the user deposits, the assets cannot be obtained on the target chain.

Suggestion:

It is recommended to use the `msg.sender` address to record the user's address during both deposit and withdrawal, instead of `tx.origin`.

WAP-4 Missing 0 Address Check

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#46-71

Descriptions:

The constructor fails to implement zero address checks for parameters such as `wormhole` and `dolaChainId`. For specifics, refer to the example code in the Wormhole GitHub repository: HelloWorld.sol lines 25-27.

```
constructor(
    IWormhole _wormhole,
    uint16 _dolaChainId,
    DolaPool _dolaPool,
    uint8 _notInvolveFundConsistency,
    uint8 _involveFundConsistency,
    uint16 _emitterChainId,
    bytes32 _emitterAddress,
    address _initialRelayer
  ) {
    wormhole = _wormhole;
    dolaChainId = _dolaChainId;
    if (address(_dolaPool) == address(0x0)) {
      // First deploy pool
      dolaPool = new DolaPool(_dolaChainId, address(this));
    } else {
      // Upgrade
      dolaPool = _dolaPool;
    }

    notInvolveFundConsistency = _notInvolveFundConsistency;
    involveFundConsistency = _involveFundConsistency;
    registeredEmitters[_emitterChainId] = _emitterAddress;
    registeredRelayers[_initialRelayer] = true;
    relayers.push(_initialRelayer);
  }
```

Suggestion:

It is recommended to add zero address checks for parameters such as `wormhole` and `dolaChainId`.

Resolution:

This issue has been fixed. The client has already added a zero address check.

WAP-5 Lack of Restriction in the `removeRelayer()` Function Raises Concerns about Emptying `Relayers`

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#144

Descriptions:

The `removeRelayer()` function lacks a constraint requiring the retention of at least one `relayer`. This implies that `relayers` can be cleared entirely. About the `deleteSpender()` function in the `DolaPool` contract, which restricts not clearing `spenders`, it raises the question of whether `relayers` should also ensure the preservation of at least one `relayer`.

Suggestion:

It is recommended to add validation in the `removeRelayer()` function to ensure that at least one `relayer` must be retained after deletion.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

WAP-6 `require()` / `revert()` Statements Should Have Descriptive Reason Strings

Severity: Minor

Status: Fixed

Code Location:

ethereum/contracts/omnipool/WormholeAdapterPool.sol#138

Descriptions:

In Solidity, it's essential to include descriptive reason strings within `require()` or `revert()` statements.

```
require(payload.opcode == LibGovCodec.ADD_RELAYER_OPCODE);
```

Suggestion:

It is recommended to add reason strings to `require()` or `revert()` .

Resolution:

This issue has been fixed. The client has already added reason strings to `require()` or `revert()` .

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.