Security Audit Report

Dolomite Protocol (V2)

SECBIT

August 2, 2021

1. Introduction

Dolomite Protocol (V2) is a decentralized exchange that allows for trustless trade settlement. It never assumes custody of its users' funds, offers margin trading using spot settlement, and provides on-chain liquidity through automated market makers (AMM) pools. SECBIT Labs conducted an audit from July 10th to August 2nd, 2021, including an analysis of the contract in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Dolomite Protocol (V2) contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Possible misuse of `Par` and `Wei`.	Medium	Fixed
Design & Implementation	4.3.2 Possible wrong implementation in the `_getAccountsForModifyPosition` function.	Low	Fixed
Design & Implementation	4.3.3 Possible wrong implementation in the `_getActionArgsForModifyPosition` function.	Low	Fixed
Gas Optimization	4.3.4 Unused function `_encodeCallAction`.	Info	Fixed
Design & Implementation	4.3.5 Liquidation conditions.	Info	Discussed
Design & Implementation	4.3.6 Unusual design in the calculation of `totalSolidHeldWei`.	Info	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the Dolomite Protocol (V2) contract is shown below:

- Project website
  - https://dolomite.io
- Smart contract code
  - https://github.com/dolomite-exchange/dolomite-v2-protocol/tree/layer-2/contracts
  - initial review commit *1bb34f5*
  - final review commit *0a33ddf*

2.2 Contract List

The following content shows the contracts included in the Dolomite Protocol (V2) project:

Name	Lines	Description
DolomiteAmmRouterProxy.sol	459	A router contract that handles liquidation addition and token exchange.
LiquidatorProxyV1WithAmmForSoloMargin.sol	335	A core contract to liquidate.
TransferProxy.sol	121	A peripheral contract to assist in constructing liquidation function parameters.
SimpleFeeOwner.sol	86	A contract used to process the administrator's protocol fee.
UniswapV2ERC20.sol	80	Base ERC20 token contract.
UniswapV2Factory.sol	59	This contract is used to set up a new market.
UniswapV2Pair.sol	339	This contract is used to manage the liquidity pool.

Note: The Dolomite V2 is forked from https://github.com/dydxprotocol/solo. The underlying core codebase has not been changed. This audit only includes the files in the table above.

3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in the protocol, namely Governance Account and Common Account.

- Governance Account
  - Description Contract administrator
  - Authority
    - Withdrawing unsupported tokens that are sent to the protocol
    - Adding new markets, represented as ERC-20 tokens
    - Closing old markets, which only disables borrowing. It does not remove a market completely
    - Changing the price oracle for a market
    - Changing the interest accrual method for a market
    - Changing the margin premium for a market
    - Changing the global margin ratio, below which users may be liquidated
    - Changing the global liquidation reward
  - Method of Authorization The creator of the contract, or authorized by the transferring of governance account
- Common Account
  - Description Users participate in margin trading exchange
  - Authority
    - Participating in token exchange
    - Adding liquidity
    - Liquidating of eligible accounts
  - Method of Authorization No authorization required

3.2 Functional Analysis

Dolomite Protocol (V2) is a DeFi protocol that provides spot trading, margin trading, passive liquidity provisioning, and other financial instruments. We can divide the critical functions of the trade contract into several parts:

DolomiteAmmRouterProxy

This contract is used to change liquidity and swap tokens.

The main functions in `DolomiteAmmRouterProxy` are as below:

- `addLiquidity()`

  Users add liquidity to the specified market and receive LP tokens.
- `removeLiquidity()`

  Users burn the liquidity LP tokens and receive the principal and proceeds.
- `swapExactTokensForTokensAndModifyPosition()`

  Users use the target token obtained by the swap to modify the position.
- `swapExactTokensForTokens()`

  Users will only swap the currently held token for the target token.
- `swapTokensForExactTokensAndModifyPosition()`

  Given the exact amount of target tokens that the user wants to swap, the amount of held tokens to be paid is calculated and swapped. At the same time, the position is modified.
- `swapTokensForExactTokens()`

  Given the exact amount of target tokens that the user wants to swap, the amount of held tokens to be paid is calculated and swapped.

LiquidatorProxyV1WithAmmForSoloMargin

This contract is used to liquidate those accounts that meet the conditions.

The main function in `LiquidatorProxyV1WithAmmForSoloMargin` is as below:

- `liquidate()`

  Users call this function to liquidate those accounts that meet the criteria for owning a mortgage and receive the corresponding reward.

UniswapV2Factory

This contract is used to add a new market and set up new token pair.

The main functions in `UniswapV2Factory` are as below:

- `createPair()`

  The administrator calls this function to add a new market.

UniswapV2Pair

This contract is used to manage market liquidity.

The main functions in `UniswapV2Pair` are as below:

- `mint()`

  A User calls this function to add liquidity. At the same time, he receives LP tokens.

- `burn()`

  A User removes the corresponding liquidity. At the same time, he will receive the principal and the corresponding reward.

4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into two types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20	✓

7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Possible misuse of `Par` and `Wei`.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Medium	Implementation logic	Fixed

Description

The `removeLiquidity()` function removes the liquidity, and the user could withdraw the tokenA and tokenB from the liquidity market. The argument `liquidity` represents the amount of DLP tokens that the user wants to remove. The user should receive the corresponding tokenA and tokenB in proportion to the liquidity removed.

But current code uses the shares `Par`, not the actual amount of tokenA and tokenB when handling the tokenA and tokenB that should be received to remove liquidity. In the subsequent processing code, `amount0` and `amount1` of `Par` shares are regarded as `Wei` (the actual amount of tokenA and tokenB) and counted according to the `Wei` processing, which will confuse the processing of funds. As a result, users will lose funds.

```
// Located on DolomiteAmmRouterProxy.sol
function removeLiquidity(
        ......
        uint liquidity,
        ......
    ) public ensure(deadline) returns (uint amountA, uint amountB) {
        address pair =
UniswapV2Library.pairFor(address(UNISWAP_FACTORY), tokenA, tokenB);
        // Send liquidity to pair
        IUniswapV2Pair(pair).transferFrom(msg.sender, pair, liquidity);

        (uint amount0, uint amount1) = IUniswapV2Pair(pair).burn(to,
toAccountNumber);
       ......
    }


// Located on UniswapV2Pair.sol
function burn(address to, uint toAccountNumber) external lock returns
(uint amount0, uint amount1) {
        ......

        uint balance0 = _getTokenBalancePar(_soloMargin, markets[0]);
        uint balance1 = _getTokenBalancePar(_soloMargin, markets[1]);

        bool feeOn;
        {
            uint liquidity = balanceOf[address(this)];

            feeOn = _mintFee(_reserve0, _reserve1);
            uint _totalSupply = totalSupply;
```

```
            //@audit The result calculated here is the share Par
            amount0 = liquidity.mul(balance0) / _totalSupply;
            amount1 = liquidity.mul(balance1) / _totalSupply;

            require(
                amount0 > 0 && amount1 > 0,
                "DLP: INSUFFICIENT_LIQUIDITY_BURNED"
            );

            _burn(address(this), liquidity);
        }

        uint[] memory amounts = new uint[](2);
        amounts[0] = amount0;
        amounts[1] = amount1;


 TransferProxy(soloMarginTransferProxy).transferMultipleWithMarkets(
            0,
            to,
            toAccountNumber,
            markets,
            amounts
        );

        ......
    }


// Located on TransferProxy.sol
function transferMultipleWithMarkets(
        uint fromAccountIndex,
        address to,
        uint toAccountIndex,
        uint[] calldata markets,
        uint[] calldata amounts
    )
    external
    nonReentrant
    {
        _transferMultiple(
            fromAccountIndex,
            to,
```

```
            toAccountIndex,
            markets,
            amounts
        );
    }

function _transferMultiple(
        uint fromAccountIndex,
        address to,
        uint toAccountIndex,
        uint[] memory markets,
        uint[] memory amounts
    )
    internal
    {
        .......
        for (uint i = 0; i < markets.length; i++) {
            Types.AssetAmount memory assetAmount;
            if (amounts[i] == uint(- 1)) {
                assetAmount = Types.AssetAmount(true,
Types.AssetDenomination.Wei, Types.AssetReference.Target, 0);
            } else {
                //@audit It seems that Par is mistakenly used here as
Wei
                assetAmount = Types.AssetAmount(false,
Types.AssetDenomination.Wei, Types.AssetReference.Delta, amounts[i]);
            }

       ......
    }
```

Status

It has been fixed by modifying the calculation in 3ffa432.

4.3.2 Possible wrong implementation in the `_getAccountsForModifyPosition` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Unexpected return	Fixed

Description

The internal function `_getParamsForSwap()` provides the correct parameter format for the `operate()` function. In other words, this function preprocesses the input data. The `_getAccountsForModifyPosition()` function needs to be called when handling the `accounts` parameter.

Consider the following scenario: the user wants to convert the tokenA to tokenB and eventually convert the tokenB obtained to tokenC. A user uses the obtained tokenC as a margin to modify the position. Then the parameter `cache.position.depositToken` will not be `address(0x0)`. The `else` branch of the following code will be executed. Then, according to the code, only `accounts[accounts.length - 1]` accounts will be processed. Other accounts will be initialized, and no information about the corresponding accounts will be recorded.

It will result in the `operate()` function not handling the funds data correctly. It will not fulfill the code design requirements.

```
// Located on DolomiteAmmRouterProxy.sol
function _getParamsForSwap(
        ModifyPositionCache memory cache
    ) internal view returns (
        Account.Info[] memory,
        Actions.ActionArgs[] memory
    ) {
        ......

        Account.Info[] memory accounts =
_getAccountsForModifyPosition(cache, pools);
       ......
       return (accounts, actions);
    }

function _getAccountsForModifyPosition(
        ModifyPositionCache memory cache,
        address[] memory pools
    ) internal pure returns (Account.Info[] memory) {
        Account.Info[] memory accounts;
        if (cache.position.depositToken == address(0)) {
            accounts = new Account.Info[](1 + pools.length);
        } else {
            accounts = new Account.Info[](2 + pools.length);
```

```
            accounts[accounts.length - 1] = Account.Info(cache.account,
0);
            // @audit If this branch is executed, the function will
return from here
            return accounts;
        }

        accounts[0] = Account.Info(cache.account,
cache.position.accountNumber);

        for (uint i = 0; i < pools.length; i++) {
            accounts[i + 1] = Account.Info(pools[i], 0);
        }

        return accounts;
    }
```

Status

It has been fixed by removing unexpected return statements in 3ffa432.

4.3.3 Possible wrong implementation in the `_getActionArgsForModifyPosition` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Incorrect logic	Fixed

Description

The internal function `_getParamsForSwap()` provides the correct parameter format for the `operate()` function. In other words, this function preprocesses the input data. The `_getActionArgsForModifyPosition()` function needs to be called when handling the `actions` parameter.

Consider the following scenario: the user wants to convert the tokenA held by the user into tokenC and deposit some of the tokenC obtained into another account of the user, then the user needs to set the parameter `cache.position.marginDeposit` to a value greater than 0, the value of which represents the amount of tokenC the user wants to transfer.

The code handles this case with the statement `amount = uint(amount);` The parameter `amount` is only initialized. Its value equals zero, which does not meet the corresponding design requirements.

```
// Located on DolomiteAmmRouterProxy.sol
function _getActionArgsForModifyPosition(
        ModifyPositionCache memory cache,
        address[] memory pools,
        uint accountsLength
    ) internal view returns (Actions.ActionArgs[] memory) {
        Actions.ActionArgs[] memory actions;
        if (cache.position.depositToken == address(0)) {
            actions = new Actions.ActionArgs[](pools.length);
        } else {
            actions = new Actions.ActionArgs[](pools.length + 1);

            // if `cache.position.marginDeposit < 0` then the user is
withdrawing to `accountNumber` (index 0).
            // `accountNumber` zero is at index `accountsLength - 1`
            uint amount;
            if (cache.position.marginDeposit == int256(- 1)) {
                amount = uint(- 1);
            } else if (cache.position.marginDeposit == MAX_INT_256) {
                amount = uint(- 1);
            } else if (cache.position.marginDeposit < 0) {
                amount = (~uint(cache.position.marginDeposit)) + 1;
            } else {
                //@audit Purpose of design ?
                amount = uint(amount);
            }

            ......
    }
```

Suggestion

A recommended modification is as follows:

```
function _getActionArgsForModifyPosition(
        ModifyPositionCache memory cache,
        address[] memory pools,
        uint accountsLength
    ) internal view returns (Actions.ActionArgs[] memory) {
        ......
            } else {

                amount = uint(cache.position.marginDeposit);
            }

            ......
    }
```

Status

It has been fixed according to the suggestion in commit 3ffa432.

4.3.4 Unused function `_encodeCallAction`.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

The following internal function does not seem to be called. It will consume additional gas when the code is deployed.

```
function _encodeCallAction(
        uint accountIndex,
        address callee,
        bytes memory data
    ) internal pure returns (Actions.ActionArgs memory) {
        return Actions.ActionArgs({
        actionType : Actions.ActionType.Call,
        accountId : accountIndex,
        amount : Types.AssetAmount(true, Types.AssetDenomination.Wei,
Types.AssetReference.Delta, 0),
        primaryMarketId : uint(- 1),
        secondaryMarketId : uint(- 1),
        otherAddress : callee,
```

```
            otherAccountId : uint(- 1),
            data : data
            });
        }
```

Status

It has been fixed by removing these codes in commit 3ffa432.

4.3.5 Liquidation conditions.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Discussed

Description

In liquidation, it is necessary to determine whether the account being liquidated meets the liquidation conditions. This function is implemented by the `checkRequirements()` function. The code uses the total value of assets pledged (named as `supplyValue`) to the liquidated account and the total value of assets credited (named as `borrowValue`) to determine whether the liquidated account is eligible for liquidation.

Consider the following situation: the user pledges tokenA (held token) and lends tokenB (owed token), assuming that the price of tokenA has fallen and is insufficient to lend the current amount of tokenB, which satisfies the liquidation condition. But then the user pledges tokenC, making the total pledge value rise, thus preventing the liquidation between tokenA and tokenB.

The code is the same as the original dydx implementation. However, it is essential to clarify the intention of this design. Do we calculate debt ratios based on the global state or specific trade pair?

```
function checkRequirements(
        Constants memory constants
    )
    private
    view
    {
        ......

        // require that the liquidAccount is liquidatable
```

```
        (
        Monetary.Value memory liquidSupplyValue,
        Monetary.Value memory liquidBorrowValue
        ) = getCurrentAccountValues(constants,
constants.liquidAccount);
        Require.that(
            liquidSupplyValue.value != 0,
            FILE,
            "Liquid account no supply"
        );
        Require.that(
            SOLO_MARGIN.getAccountStatus(constants.liquidAccount) ==
Account.Status.Liquid ||
            !isCollateralized(liquidSupplyValue.value,
liquidBorrowValue.value, SOLO_MARGIN.getMarginRatio()),
            FILE,
            "Liquid account not liquidatable"
        );
    }

function getCurrentAccountValues(
        Constants memory constants,
        Account.Info memory account
    )
    private
    view
    returns (
        Monetary.Value memory,
        Monetary.Value memory
    )
    {
        Monetary.Value memory supplyValue;
        Monetary.Value memory borrowValue;

        for (uint256 m = 0; m < constants.markets.length; m++) {
            Types.Par memory par = SOLO_MARGIN.getAccountPar(account,
m);
            if (par.isZero()) {
                continue;
            }
            // @audit Current user's token balance
            Types.Wei memory userWei = Interest.parToWei(par,
constants.markets[m].index);
            // @audit Convert tokens to value (e.g. USDT)
```

```
            uint256 assetValue =
userWei.value.mul(constants.markets[m].price.value);
            if (userWei.sign) {
                supplyValue.value = supplyValue.value.add(assetValue);
            } else {
                borrowValue.value = borrowValue.value.add(assetValue);
            }
        }

        return (supplyValue, borrowValue);
    }
```

Status

The issue has been discussed. The developer team updates the comments and clarifies code meaning in commit 0a33ddf.

4.3.6 Unusual design in the calculation of `totalSolidHeldWei`.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Discussed

Description

The parameter `cache.solidHeldUpdateWithReward` indicates the amount of held tokens deducted from the collateral by the liquidated account. These held tokens will be paid to the liquidator, part of which will be used to repay the loan owed token that the liquidator helped the liquidated account repay. The rest will be used as a reward for the liquidator.

In the circumstance that the held token obtained by the liquidator is not sufficient to cover the owed token paid by the liquidator, the liquidator may choose to refuse the liquidation. According to the current code, if the liquidator account originally holds the held token, this will be counted.

Suppose the held token pledged by the liquidated account is not enough to pay the owed token, and the liquidator's account has enough funds. In that case, after completing the liquidation, the liquidator will lose some of the funds (which are used to make up for the lack of funds in the liquidated account). It is necessary to clarify whether this design meets the requirements.

```
function liquidate(
        Account.Info memory solidAccount,
        Account.Info memory liquidAccount,
        uint256 owedMarket,
        uint256 heldMarket,
        address[] memory tokenPath,
        bool revertOnFailToSellCollateral
    )
    public
    nonReentrant
    {
        ......
        uint totalSolidHeldWei = cache.solidHeldUpdateWithReward;
        // @audit purpose ?
        if (cache.solidHeldWei.sign) {
            totalSolidHeldWei =
totalSolidHeldWei.add(cache.solidHeldWei.value);
        }

        (Account.Info[] memory accounts, Actions.ActionArgs[] memory
actions) =
        ROUTER_PROXY.getParamsForSwapTokensForExactTokens(
            constants.solidAccount.owner,
            constants.solidAccount.number,
            uint(- 1), // maxInputWei
            cache.toLiquidate, // the amount of owedMarket that needs
to be repaid. Exact output amount
            tokenPath
        );
        if (revertOnFailToSellCollateral) {
            Require.that(
                totalSolidHeldWei >= actions[0].amount.value,
                FILE,
                "totalSolidHeldWei is too small",
                totalSolidHeldWei,
                actions[0].amount.value
            );
        }
        ......
}
```

Status

The issue has been discussed. The developer team updates the comments and clarifies code meaning in commit 0a33ddf.

5. Conclusion

After auditing and analyzing Dolomite Protocol (V2) contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. The Dolomite Protocol (V2) is a user-friendly, highly integrated, decentralized exchange protocol with high capital efficiency. SECBIT Labs holds the view that Dolomite Protocol (V2) smart contract has high code quality, concise implementation, and detailed documentation.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io