Marginswap
Findings & Analysis Report

2021-05-03

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization that consists of security researchers, auditors, developers, and individuals with domain expertise in the area of smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Marginswap's smart contract system written in Solidity. The code contest took place between April 2 and April 7, 2021.

Wardens

5 Wardens contributed reports to the Marginswap code contest:

- cmichel
- gpersoon
- jvaqa
- pauliax
- s1mo

This contest was judged by Zak Cole.

Final report assembled by sockdrawermoney.

Summary

The C4 analysis yielded an aggregated total of 64 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 12 received a risk rating in the category of HIGH severity, 12 received a risk rating in the category of MEDIUM severity, and 34 received a risk rating in the category of LOW severity.

C4 analysis also identified an aggregate total of 8 non-critical recommendations.

Scope

The code under review can be found within the C4 code contest repository and comprises 19 smart contracts written in the Solidity programming language.

Severity Criteria

C4 assesses severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 3 primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings

[H-01] Re-entrancy bug allows inflating balance

One can call the `MarginRouter.crossSwapExactTokensForTokens` function first with a fake contract disguised as a token pair: `crossSwapExactTokensForTokens(0.0001 WETH, 0, [ATTACKER_CONTRACT], [WETH, WBTC])`. When the amounts are computed by the `amounts = UniswapStyleLib.getAmountsOut(amountIn - fees, pairs, tokens);` call, the attacker contract returns fake reserves that yield 1 WBTC for the tiny input. The resulting amount is credited through `registerTrade`. Afterwards, `_swapExactT4T([0.0001 WETH, 1 WBTC], 0, [ATTACKER_CONTRACT], [WETH, WBTC])` is called with the fake pair and token amounts. At some point `_swap` is called, the starting balance is stored in `startingBalance`, and the attacker contract call allows a re-entrancy:

```
pair.swap(0.0001 WETH, 1 WBTC, FUND, new bytes(0)); // can re-er
```

From the ATTACKER*CONTRACT we re-enter the ÌMarginRouter.crossSwapExactTokensForTokens(30 WETH, 0, WETH*WBTC*PAIR, [WETH, WBTC])* `function with the actual WETH <> WBTC pair contract. All checks pass, the FUND receives the actual amount, the outer` swap `continues execution after the re-entrancy and the` endingBalance >= startingBalance + amounts[amounts.length - 1]`check passes as well because the inner swap successfully deposited these funds. We end up doing 1 real trade but being credited twice the output amount.

This allows someone to be credited multiples of the actual swap result. This can be repeated many times and finally, all tokens can be stolen.

Recommend adding re-entrancy guards (from OpenZeppelin) to all external functions of `MarginRouter`. There might be several attack vectors of this function as the attacker controls many parameters. The idea of first doing an estimation with `UniswapStyleLib.getAmountsOut(amountIn - fees, pairs, tokens)` and updating the user with these estimated amounts, before doing the actual trade, feels

quite vulnerable to me. Consider removing the estimation and only doing the actual trade first, then calling `registerTrade` with the actual trade amounts returned.

[H-02] Missing `fromToken != toToken` check

Attacker calls `MarginRouter.crossSwapExactTokensForTokens` with a fake pair and the same token[0] == token[1]. `crossSwapExactTokensForTokens(1000 WETH, 0, [ATTACKER_CONTRACT], [WETH, WETH])`. When the amounts are computed by the `amounts = UniswapStyleLib.getAmountsOut(amountIn - fees, pairs, tokens);` call, the attacker contract returns fake reserves that yield 0 output. When `_swapExactT4T` is called, the funds are sent to the fake contract and doing nothing passes all checks in `_swap` call that follows because the `startingBalance` is stored *after* the initial Fund withdraw to the pair.

```
function _swapExactT4T() {
  // withdraw happens here
    Fund(fund()).withdraw(tokens[0], pairs[0], amounts[0]);
    _swap(amounts, pairs, tokens, fund());
}

function _swap() {
  uint256 startingBalance = IERC20(outToken).balanceOf(_to);
  uint256 endingBalance = IERC20(outToken).balanceOf(_to);
  // passes as startingBalance == endingBalance + 0
  require(
      endingBalance >= startingBalance + amounts[amounts.length
      "Defective AMM route; balances don't match"
  );
}
```

The full impact is not yet known as `registerTrade` could still fail when subtracting the `inAmount` and adding 0 `outAmount`. At least, this attack is similar to a withdrawal which is supposed to only occur after a certain `coolingOffPeriod` has passed, but this time-lock is circumvented with this attack.

Recommend moving the fund withdrawal to the first pair **after** the `startingBalance` assignment. Check `fromToken != toToken` as cyclical trades

(arbitrages) are likely not what margin traders are after. Consider if the same check is required for `registerTradeAndBorrow` / `adjustAmounts` functions.

[H-03] Price feed can be manipulated

Anyone can trigger an update to the price feed by calling `PriceAware.getCurrentPriceInPeg(token, inAmount, forceCurBlock=true)`. If the update window has passed, the price will be computed by simulating a Uniswap-like trade with the amounts. This simulation uses the reserves of the Uniswap pairs which can be changed drastically using flash loans to yield almost arbitrary output amounts, and thus prices. Wrong prices break the core functionality of the contracts such as borrowing on margin, liquidations, etc.

Recommend against using the Uniswap spot price as the real price. Uniswap itself warns against this and instead recommends implementing a [TWAP price oracle] using the `price*CumulativeLast` variables.

[H-04] Inconsistent usage of `applyInterest`

It is unclear if the function `applyInterest` is supposed to return a new balance with the interest applied or only the accrued interest? There are various usages of it, some calls add the return value to the old amount:

```
return
bond.amount +
applyInterest(bond.amount, cumulativeYield, yieldQuotientFP);
and some not:

balanceWithInterest = applyInterest(
balance,
yA.accumulatorFP,
yieldQuotientFP
);
```

This makes the code misbehave and return the wrong values for the balance and accrued interest.

Recommend making it consistent in all cases when calling this function.

[H-05] Wrong liquidation logic

The `belowMaintenanceThreshold` function decides if a trader can be liquidated:

```
    function belowMaintenanceThreshold(CrossMarginAccount storage ac
        internal
        returns (bool)
    {
        uint256 loan = loanInPeg(account, true);
        uint256 holdings = holdingsInPeg(account, true);
        // The following should hold:
        // holdings / loan >= 1.1
        // =>
        return 100 * holdings >= liquidationThresholdPercent * loan;
    }
```

The inequality in the last equation is wrong because it says the higher the holdings (margin + loan) compared to the loan, the higher the chance of being liquidated. The inverse equality was probably intended `return 100 * holdings <= liquidationThresholdPercent * loan;`. Users that shouldn't be liquidated can be liquidated, and users that should be liquidated cannot get liquidated.

[H-06] Users are credited more tokens when paying back debt with `registerTradeAndBorrow`

The `registerTradeAndBorrow` is called with the results of a trade (`inAmount`, `outAmount`). It first tries to pay back any debt with the `outAmount`. However, the **full** `outAmount` is credited to the user again as a deposit in the `adjustAmounts(account, tokenFrom, tokenTo, sellAmount, outAmount);` call. As the user pays back their debt and is credited the same amount again, they are essentially credited twice the `outAmount`, making a profit of one `outAmount`. This can be withdrawn and the process can be repeated until the funds are empty.

In the `adjustAmounts` call, it should only credit `outAmount - extinguishableDebt` as a deposit like in `registerDeposit`. The `registerDeposit` function correctly handles this case.

[H-07] `account.holdsToken` is never set

The `addHolding` function does not update the `account.holdsToken` map.

```
function addHolding(
    CrossMarginAccount storage account,
    address token,
    uint256 depositAmount
) internal {
    if (!hasHoldingToken(account, token)) {
        // SHOULD SET account.holdsToken here
        account.holdingTokens.push(token);
    }

    account.holdings[token] += depositAmount;
}
```

This leads to a critical vulnerability where deposits of the same token keep being pushed to the `account.holdingTokens` array but the sum is correctly updated in `account.holdings[token]`. However, because of the duplicate token in the `holdingTokens` array the same token is counted several times in the `getHoldingAmounts` function:

```
function getHoldingAmounts(address trader)
    external
    view
    override
    returns (
        address[] memory holdingTokens,
        uint256[] memory holdingAmounts
    )
{
    CrossMarginAccount storage account = marginAccounts[trader];
    holdingTokens = account.holdingTokens;

    holdingAmounts = new uint256[](account.holdingTokens.length);
    for (uint256 idx = 0; holdingTokens.length > idx; idx++) {
        address tokenAddress = holdingTokens[idx];
        // RETURNS SUM OF THE BALANCE FOR EACH TOKEN ENTRY
        holdingAmounts[idx] = account.holdings[tokenAddress];
    }
```

```
    }
```

The `MarginRouter.crossCloseAccount` function uses these wrong amounts to withdraw all tokens:

```
    function crossCloseAccount() external {
        (address[] memory holdingTokens, uint256[] memory holdingAmo
            IMarginTrading(marginTrading()).getHoldingAmounts(msg.se

        // requires all debts paid off
        IMarginTrading(marginTrading()).registerLiquidation(msg.send

        for (uint256 i; holdingTokens.length > i; i++) {
            Fund(fund()).withdraw(
                holdingTokens[i],
                msg.sender,
                holdingAmounts[i]
            );
        }
    }
```

An attacker can just deposit the same token X times which increases their balance by X times the actual value. This inflated balance can then be withdrawn to steal all tokens.

Recommend correctly setting the `account.holdsToken` map in `addHolding`.

[H-08] Rewards cannot be withdrawn

The rewards for a recipient in `IncentiveDistribution.sol` are stored in the storage mapping indexed by recipient `accruedReward[recipient]` and the recipient is the actual margin trader account, see `updateAccruedReward`.

These rewards are supposed to be withdrawn through the `withdrawReward` function but `msg.sender` is used here instead of a `recipient` (`withdrawer`) parameter. However, `msg.sender` is enforced to be the incentive reporter and can therefore not be the margin trader.

Nobody can withdraw the rewards.

Recommend removing the `isIncentiveReporter(msg.sender)` check from `withdrawReward` function.

[H-09] lastUpdatedDay not initialized

The variable lastUpdatedDay in IncentiveDistribution.sol is not (properly) initialized. This means the function updateDayTotals will end up in a very large loop which will lead to an out of gas error. Even if the loop would end, the variable currentDailyDistribution would be updated very often. Thus updateDayTotals cannot be performed.

The entire IncentiveDistribution does not work. If the loop would stop, the variable currentDailyDistribution is not accurate, resulting in a far lower incentive distribution than expected.

Recommend initializing lastUpdatedDay with something like block.timestamp / (1 days)

```
uint256 lastUpdatedDay; # ==> lastUpdatedDay = 0

# When the function updateDayTotals is called:
uint256 public nowDay = block.timestamp / (1 days); #==> ~ 18721
uint256 dayDiff = nowDay - lastUpdatedDay; #==> 18721-0 = 18721

for (uint256 i = 0; i < dayDiff; i++) { # very long loop (18721)
currentDailyDistribution = ....
}
#will result in an out of gas error

## [[H-10] function buyBond charges msg.sender twice](https://gi

function buyBond transfers amount from msg.sender twice:
Fund(fund()).depositFor(msg.sender, issuer, amount);
...
collectToken(issuer, msg.sender, amount);
```

[H-11] Impossible to call withdrawReward fails due to run out of gas

The withdrawReward (https://github.com/code-423n4/marginswap/blob/main/contracts/IncentiveDistribution.sol#L224) fails due to the loop at https://github.com/code-423n4/marginswap/blob/main/contracts/IncentiveDistribution.sol#L269. Based on testing, the dayDiff would be 18724 and with a gasLimit of 9500000 it stops at iteration 270 due to the fact that lastUpdatedDay is not initialized so is 0. Other than that it could run out of gas also for the loop of allTranches (https://github.com/code-423n4/marginswap/blob/main/contracts/IncentiveDistribution.sol#L281) because it's an unbounded array.

I'm not sure of the logic behind the shrinking of the daily distribution but i think that maybe you just missed to initialize the lastUpdatedDay to the day of deployment? If that's the case it resolves partially the problem because allTranches is theoretically unbounded even though only the owner can add element to it and you should do deeply testing to understand how many elements it can have until it run out of gas. I read the comment that says you tried to shift the gas to the withdrawal people maybe you went too further and is it worth rethinking the design?

werg (Marginswap) confirmed:

> Exactly. we need to initialize lastUpdatedDay.

Medium Risk Findings

[M-01] No default `liquidationThresholdPercent`

The `IsolatedMarginTrading` contract does not define a default `liquidationThresholdPercent` which means it is set to 0. The `belowMaintenanceThreshold` function uses this value and anyone could be liquidated due to `100 * holdings >= liquidationThresholdPercent * loan = 0` being always true.

Anyone can be liquidated immediately. If the faulty `belowMaintenanceThreshold` function is fixed (see other issue), then nobody could be liquidated which is bad as

well.

Recommend setting a default liquidation threshold like in `CrossMarginTrading` contracts.

[M-02] Missing checks if pairs equal tokens

The `UniswapStyleLib.getAmountsOut`, `PriceAware.setLiquidationPath` (and others) don't check that `path.length + 1 == tokens.length` which should always hold true. Also, it does not check that the tokens actually match the pair. It's easy to set faulty liquidation paths which then end up reverting the liquidation transactions.

[M-03] No entry checks in crossSwap[Exact]TokensFor[Exact]Tokens

The functions crossSwapTokensForExactTokens and crossSwapExactTokensForTokens of MarginRouter.sol do not check who is calling the function. They also do not check the contents of pairs and tokens nor do they check if the size of pairs and tokens is the same.

`registerTradeAndBorrow` within `registerTrade` does seem to do an entry check (require(isMarginTrader(msg.sender)...) however as this is an external function msg.sender is the address of MarginRouter.sol, which will verify ok.

Calling these functions allow the caller to trade on behalf of marginswap, which could result in losing funds. It's possible to construct all parameters to circumvent the checks. Also the "pairs" can be fully specified; they are contract addresses that are called from getAmountsIn / getAmountsOut and from pair.swap. This way you can call arbitrary (self constructed) code, which can reentrantly call the marginswap code.

Recommend limiting who can call the functions. Perhaps whitelist contents of pairs and tokens. Check the size of pairs and tokens is the same.

werg (Marginswap) confirmed:

> This has merit: particularly the part about self-constructed pairs. We either need much more rigorous checks or a a process for vetting & approving pairs. The latter is likely more gas efficient.

[M-04] maintainer can be pushed out

The function liquidate (in both `CrossMarginLiquidation.sol` and `IsolatedMarginLiquidation.sol`) can be called by everyone. If an attacker calls this repeatedly then the maintainer will be punished and eventually be reported as maintainerIsFailing And then the attacker can take the payouts.

When a non authorized address repeatedly calls liquidate then the following happens: `isAuthorized = false` which means `maintenanceFailures[currentMaintainer]` increases. After sufficient calls it will be higher than the threshold and then `maintainerIsFailing()` will be true. This results in `canTakeNow` being true, which finally means the following will be executed:

```
Fund(fund()).withdraw(PriceAware.peg, msg.sender, maintainerCut);
```

An attacker can push out a maintainer and take over the liquidation revenues.

Recommend put authorization on who can call the liquidate function, review the maintainer punishment scheme.

werg (Marginswap) disputed:

> I believe this issue is not a vulnerability, due to the checks in lines 326-335. Even if someone comes in first and claims the maintainer is failing they can do their job in the same or next block and get all / most of their failure record extinguished.

zscole (Judge):

> Acknowledging feedback from @werg, but maintaining the reported risk level of `medium` since this has implications on token logic.

[M-05] Several function have no entry check

The following functions have no entry check or a trivial entry check:

```
withdrawHourlyBond Lending.sol
closeHourlyBondAccount Lending.sol
haircut Lending.sol
addDelegate(own adress...) Admin.sol
removeDelegate(own adress...) Admin.sol
depositStake Admin.sol
disburseLiqStakeAttacks CrossMarginLiquidation.sol
disburseLiqStakeAttacks IsolatedMarginLiquidation.sol
getCurrentPriceInPeg PriceAware.sol
```

By manipulating the input values (for example extremely large values), you might be able to disturb the internal administration of the contract, thus perhaps locking function or giving wrong rates.

Recommend checking the functions to see if they are completely risk free and add entry checks if they are not, and add a comment to notify the function is meant to be called by everyone.

werg (Marginswap):

- `withdrawHourlyBond` : could not find vulnerability, since solidity 0.8.x fails on underflow in HourlyBondSubscriptionLending.sol:115 in case of unauthorized access.
- `closeHourlyBondAccount` : same story since both call into `_withdrawHourlyBond`
- `haircut` : trivially guarded in one way, though this actually has merit in another way — if at some point down the road an attacker were able to establish a token, make it popular enough for us to add it to cross margin, but include in that token contract a malicious function that calls haircut, they could then void everybody's bonds in their token. I don't see how it would be profitable, it's definitely an expensive long con, but... we should add an extra guard to make sure it's an isolated margin trading contract.
- `addDelegate` has a guard.
- `removeDelegate` has a guard as well, or am I missing something here?

- `depositStake` fails for unfunded requests in the safe transfer in `Fund.depositFor`
- `disburseLiqStakeAttacks` should be universally accessible by design
- `getCurrentPriceInPeg` only updates state in a rate limited way, hence fine for it to be public

> I will add comments to the effect. Thanks again

[M-06] Users Can Drain Funds From MarginSwap By Making Undercollateralized Borrows If The Price Of A Token Has Moved More Than 10% Since The Last MarginSwap Borrow/Liquidation Involving Accounts Holding That Token.

Users Can Drain Funds From MarginSwap By Making Undercollateralized Borrows If The Price Of A Token Has Moved More Than 10% Since The Last MarginSwap Borrow/Liquidation Involving Accounts Holding That Token.

MarginSwap's internal price oracle is only updated for a particular token if a borrow or liquidation is attempted for an account that is lending/borrowing that particular token.

For a less popular token, the price could move quite a bit without any borrow or liquidation being called on any account lending/borrowing that token, especially if MarginSwap does not end up being wildly popular, or if it supports lesser known assets. If the Uniswap price has moved more than 10% (liquidationThresholdPercent - 100) without a borrow or liquidation on an account lending/borrowing that particular token occurring on MarginSwap, then Alice can make undercollateralized loans, leaving behind her collateral and draining funds from the contract.

(1) Alice waits for the Uniswap price for any token to move more than 10% (liquidationThresholdPercent - 100) without a borrow or liquidation occurring for any account lending/borrowing that token occurring on MarginSwap. (2) When this condition is satisfied, Alice can loop the following actions: (2.1) If the price has fallen, Alice can use the token as collateral (making sure to use more than UPDATE*MAX*PEG*AMOUNT worth of the token in ETH), borrow ether from MarginSwap, sell the ether for the token on Uniswap, and repeat, leaving coolingOffPeriod blocks between each lend and borrow. (2.2) If the price has risen,*

*Alice can use ether as collateral, borrow the token from MarginSwap (making sure to use more than UPDATE*MAX*PEG*AMOUNT worth of the token in ETH), sell the token for ether on Uniswap, and repeat, leaving coolingOffPeriod blocks between each lend and borrow.

Because the MarginSwap price is now stale, Alice can borrow more than 100% of the actual value of her collateral, since MarginSwap believes the borrowed funds to only be worth 90% or less than their actual current market value.

The various defenses that MarginSwap has employed against undercollateralized loans are all bypassed: (a) The exponential moving price average stored within MarginSwap is not updated, because Alice borrows at least UPDATE*MAX*PEG*AMOUNT worth of the token in ETH, so MarginSwap.PriceAware.getPriceFromAMM skips the price update due to the "outAmount < UPDATE*MAX*PEG*AMOUNT" condition failing. (b) CoolingOffPeriod can be bypassed by Alice splitting her deposits and borrows up by 20 blocks (the current value of CoolingOffPeriod). Since deposits do not trigger a price oracle update, Alice can even deposit ahead of time as the price is nearing 10% off of peg, allowing her to perform the borrow right when 10% is passed. (c) MarginSwap.Lending.registerBorrow check is bypassed. The check is "meta.totalLending >= meta.totalBorrowed", but this is a global check that only ensures that the contract as a whole has sufficient tokens to fund Alice's borrow. Alice simply needs to ensure that she only borrows up to the amount of tokens that the contract currently owns.

Even if the issue of the price oracle stalling were to be fixed by using a large enough UPDATE*MAX*PEG_AMOUNT, since the moving average updates so slowly (MarginSwap is currently set at moving 8/1000th towards new price every 8 blocks) and only when actions are taken on MarginSwap (which may not be frequent on lesser known tokens or if MarginSwap is not too popular), Alice can still take out undercollateralized loans for a period of time before the price oracle catches up. The real solution here is to use UniswapV2/SushiSwap/UniswapV3's built in TWAP price oracle, especially since MarginSwap is built on top of Uniswap/Sushiswap.

werg (Marginswap) acknowledged:

> I believe the issue above is referring to the "overcollateralized borrow" functionality, because there is talk of withdrawing. In any case of withdrawal

> (whether immediately or not) it is not the `liquidationThresholdPercent` which governs how much a user may withdraw. Rather, we check whether the account has positive balance (i.e. the value of assets exceeds the loan).

> In the current system, at 3x possible leverage level, users can withdraw maximally 66% of the face value (to the system) they deposited. If a user deposited collateral that had dropped in price by 10% it would allow them to withdraw around 73% of the real value. — Not undercollateralized. The price of an asset would have to have dropped by 33%, without the system catching on, for Alice to break even (without considering gas cost).

> Cross margin trading will only be available for a select set of tokens with high enough trading volume. Anyone will be able to update the price if our exponential weighted average is out of date. Nevertheless, risk remains as in any lending system.

- We will consider adding an additional buffer around immediate withdrawals
- If staleness becomes an issue the protocol can institute rewards for updating the price

> also of course there are liquidators waiting in the wings to make their cut on underwater accounts and liquidators can update the price.

[M-07] diffMaxMinRuntime gets default value of 0

`uint256 public diffMaxMinRuntime;`` This variable is never set nor updated so it gets a default value of 0.` diffMaxMinRuntimè with 0 value is making the calculations that use it either always return 0 (when multiplying) or fail (when dividing) when calculating bucket indexes or sizes.

Recommend setting the appropriate value for diffMaxMinRuntime and update it whenever min or max runtime variables change.

[M-08] PriceAware uses prices from getAmountsOut

`getPriceFromAMM` relies on values returned from getAmountsOut which can be manipulated (e.g. with the large capital or the help of flash loans). The impact is

reduced with UPDATE*MIN*PEG*AMOUNT and UPDATE*MAX*PEG*AMOUNT*, however, it is not entirely eliminated.

Uniswap v2 recommends using their TWAP oracle: https://uniswap.org/docs/v2/core-concepts/oracles/

[M-09] Isolated margin contracts declare but do not set the value of liquidationThresholdPercent

CrossMarginTrading sets value of liquidationThresholdPercent in the constructor: `liquidationThresholdPercent = 110;` Isolated margin contracts declare but do not set the value of liquidationThresholdPercent.

Recommend setting the initial value for the liquidationThresholdPercent in Isolated margin contracts.

This makes function belowMaintenanceThreshold to always return true unless a value is set via function setLiquidationThresholdPercent. Comments indicate that the value should also be set to 110:

```
    // The following should hold:
    // holdings / loan >= 1.1
    // => holdings >= loan \* 1.1
```

[M-10] Add a timelock to functions that set key variables

Functions like `setLeveragePercent` and `setLiquidationThresholdPercent` for both `IsolatedMarginTrading` and `CrossMarginTrading` should be put behind a timelock because they would give more trust to users. Currently, the owner could call them whenever they want and a position could become liquidable from a block to the other.

werg (Marginswap) acknowledged:

> Timelock will be handled by governance

zscole commented:

> Maintaining submission rating of `2 (Med Risk)` because this presents a vulnerability at the time of review.

Low Risk Findings

[L-01] Events not indexed

The `CrossDeposit`, `CrossTrade`, `CrossWithdraw`, `CrossBorrow`, `CrossOvercollateralizedBorrow` events in `MarginRouter` are not indexed. Off-chain scripts cannot efficiently filter these events.

Recommend adding an index on important arguments like `trader`.

[L-02] `getReserves` does not check if tokens match

The `UniswapStyleLib.getReserves` function does not check if the tokens are the pair's underlying tokens. It blindly assumes that the tokens are in the wrong order if the first one does not match but they could also be completely different tokens.

It could be the case that output amounts are computed for completely different tokens because a wrong pair was provided.

[L-03] Role 9 in Roles.sol

Roles.sol contains the following: `roles[msg.sender][9] = true;`

It's not clear what the number 9 means. In RoleAware.sol there is a constant with the value 9: `uint256 constant TOKEN_ACTIVATOR = 9;`

The code is more difficult to read without an explanation for the number 9. In case the code would be refactored in the future and the constants in RoleAware.sol are renumbered, the value in Roles.sol would no longer correspond to the right value.

Recommend moving the constants from `Roles.sol` to `RoleAware.sol` and replace 9 with the appropriate constant.

werg (Marginswap) confirmed

[L-04] Multisig wallets can’t be used for liquidate

The function liquidate, which is defined in both `CrossMarginLiquidation.sol` and `IsolatedMarginLiquidation.sol` , includes the modifier noIntermediary. This modifier prevents the use of Multisig wallets.

If the maintainer happens to use a multisig wallet they might not experience any issues until they try to call the function liquidate. At that moment they can’t successfully call the function.

Recommend verifying if the prevention to use multisig wallets is intentional. In that case add a comment to the liquidate functions. If it is not intentional update the code so multisigs wallets can be supported.

[L-05] Different solidity version in UniswapStyleLib.sol

The solidity version in UniswapStyleLib.sol (>=0.5.0) is different than the solidity version in the other contracts (e.g. ^0.8.0) Also math actions are present in the functions getAmountOut and getAmountIn that could easily lead to an underflow or division by 0; (note safemath is not used). Note: In solidity 0.8.0 safemath like protections are default.

The impact is low because UniswapStyleLib is a library and the solidity version of the contract that uses the library is used (e.g. ^0.8.0), which has safemath like protections. It is cleaner to have the same solidity version everywhere.

getAmountIn(3,1,1000) would give division by 0 getAmountIn(1,1,1) will underflow denominator

[L-06] sortTokens can be simplified

The function sortTokens UniswapStyleLib.sol returns 2 values, but only the first return value is used: `MarginRouter.sol: (address token0, ) = UniswapStyleLib.sortTokens... UniswapStyleLib.sol: (address token0, ) = sortTokens..`

In both cases the used return value is compared to the first parameter of the function call. Conclusion: the function is only used to determine the smaller of the

two tokens, not really to sort tokens.

Recommend simplifying the code:

```
function ASmallerThanB(address tokenA, address tokenB)
internal
pure
returns (bool)
{
    require(tokenA != tokenB, "Identical address!");
    require(tokenA != address(0), "Zero address!");
    require(tokenB != address(0), "Zero address!");
    return tokenA < tokenB;
}
```

[L-07] Duplicated Code In Admin.viewCurrentMaintenanceStaker()

There are four lines of code that are duplicated in `viewCurrentMaintenanceStaker`.

Change this:

```
if (maintenanceStakePerBlock > currentStake) {
    // skip
    staker = nextMaintenanceStaker[staker];
    currentStake = getMaintenanceStakerStake(staker);
} else {
    startBlock += currentStake / maintenanceStakePerBlock;
    staker = nextMaintenanceStaker[staker];
    currentStake = getMaintenanceStakerStake(staker);
}
```

To this:

```
if (maintenanceStakePerBlock <= currentStake) {
     += currentStake / maintenanceStakePerBlock;
}
staker = nextMaintenanceStaker[staker];
```

```
currentStake = getMaintenanceStakerStake(staker);
```

[L-08] Magic Numbers used in Admin._stake() When Constant Defined Above Can Be Used Instead

Magic Numbers are used in `Admin.\_stake()`, which both obscure the purpose of the function and unnecessarily lead to potential error if the constants are changed during development. Since they are used to refer to a constant defined in RoleAware, and Admin inherits from RoleAware, then Admin can simply call that constant.

In `Admin.\_stake()`, change this:

```
IncentiveDistribution(incentiveDistributor()).addToClaimAmount(1
```

to this:

```
IncentiveDistribution(incentiveDistributor()).addToClaimAmount(
FUND_TRANSFERER,
holder,
amount
);
```

[L-09] function initTranche should check that the share parameter is > 0

function `initTranche` should check that the “share” parameter is > 0, otherwise, it may be possible to initialize the same tranche again.

[L-10] runtime > 1 hours error message discrepancy

Here, the revert message says that the value needs to be at least 1 hour, however, the code allows value only above the 1 hour (> instead of >=): `require(runtime > 1 hours, "Min runtime needs to be at least 1 hour");`

There is no real impact on security here, just a discrepancy between the check and message.

[L-11] setLeveragePercent should check that new _leveragePercent >= 100

function `setLeveragePercent` should check that the `\_leveragePercent >= 100` so that this calculation will not fail later: `(leveragePercent - 100)`

This variable can only be set by admin so as long as he sets the appropriate value it should be fine.

[L-12] An erroneous constructor's argument could block the withdrawReward

The constructor of `IncentiveDistribution` take as argument the address of MFI token but it doesn't check that is != address(0). Not worth an issue alone but `IncentiveDistribution` imports `IERC20.sol` and never uses it.

In case the address(0) is passed as argument the `withdrawReward` would fail and due to the fact that MFI is immutable the only solution would be to redeploy the contract meanwhile losing trust from the users.

Deploy `IncentiveDistribution` with 0 as `\_MFI argument` and then call `withdrawReward`.

[L-13] Not emitting event for important state changes

When changing state variables events are not emitted.

PriceAware:

- setPriceUpdateWindow
- setUpdateRate
- setUpdateMaxPegAmount
- setUpdateMinPegAmount Lending (https://github.com/code-423n4/marginswap/blob/main/contracts/Lending.sol):

- activateIssuer
- deactivateIssuer
- setLendingCap
- setLendingBuffer
- setHourlyYieldAPR
- setRuntimeWeights IncentiveDistribution (https://github.com/code-423n4/marginswap/blob/main/contracts/IncentiveDistribution.sol#L261):
- setTrancheShare
- initTranche IsolatedMarginTrading and CrossMarginTrading (https://github.com/code-423n4/marginswap/blob/main/contracts/IsolatedMarginTrading.sol - https://github.com/code-423n4/marginswap/blob/main/contracts/CrossMarginTrading.sol):
- setCoolingOffPeriod
- setLeveragePercent
- setLiquidationThresholdPercent

The events emitted by MarginRouter don't have indexed parameter.

Recommended mitigation:

- For `set... function` emit events with old and new value.
- For `initTranche` , event `InitTranche(uint256 tranche, uint256 share)` 0 For `activateIssuer` , event `ActivateIssuer(address issuer, address token)` For `deactivateIssuer` , event `DeactivateIssuer(address issuer)`

For events emitted by MarginRouter, recommend indexing the trader address to make it filterable.

werg (Marginswap):

> We may sprinkle in a few more events before launch, but in the interest of gas savings we try not to emit events for state that can be queried using view functions.

zscole commented:

> Reducing this from submitted rating of `2 (Med Risk)` to `1 (Low Risk)` since it presents no immediate risk to the security of the system, but could have implications on overall functionality.

Non-Critical Findings

- [N-01] Liquidations can be sandwich attacked
- [N-02] Unlocked Pragma
- [N-03] No function for TOKEN_ADMIN in RoleAware.sol
- [N-04] isStakePenalizer differtent than other functions in RoleAware.sol
- [N-05] Natspec comments not used in a consistent way
- [N-06] Function parameter named timestamp
- [N-07] Naming convention for internal functions not used consistently
- [N-08] Todo's left in code
- [N-09] The First User To Borrow a Particular Token Can Drain Funds In MarginSwap by Making An Undercollateralized Borrow Using Flash Loans
- [N-10] function crossWithdrawETH does not emit withdraw event
- [N-11] All caps indicates that the value should be constant
- [N-12] TODOs left in code
- [N-13] Consistent function names
- [N-14] Useless overflow comments
- [N-15] Variable is declared and initialized with different values
- [N-16] Code duplication in viewCurrentMaintenanceStaker
- [N-17] Misleading revert messages
- [N-18] setUpdateMaxPegAmount and setUpdateMinPegAmount do not check boundaries
- [N-22] Liquidators may be a subject of front-running attacks

Gas Optimizations

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code, but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top