



**SHERLOCK**

# **SHERLOCK SECURITY REVIEW FOR**



<b>Prepared for:</b>	<b>JOJO</b>
<b>Prepared by:</b>	<b>Sherlock</b>
<b>Lead Security Expert:</b>	<b><u>0x52</u></b>
<b>Dates Audited:</b>	<b>April 24 - May 10, 2023</b>
<b>Prepared on:</b>	<b>June 23, 2023</b>

## Introduction

JOJO Exchange is a DeFi-Native Perpetual Contract that is: Liquid, Safer, and Faster. Security is everything.

## Scope

Repository: JOJOexchange/smart-contract-EVM

Branch: main

Commit: 4a95a8e9a6367ae88dc827e29467229cb5bbad4f

---

Repository: JOJOexchange/JUSDV1

Branch: main

Commit: 3fc8dc85d0a54df5205b9d79e3ef1bf95fbfe012

---

For the detailed scope, see the [contest details](#).

## Findings

Each issue has an assigned severity:

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

## Issues found

Medium	High
13	1

## Issues not fixed or acknowledged

Medium	High
0	0



## Security experts who found valid issues

0x52  
ast3ros  
GalloDaSballo  
monrel  
carrotsmuggler  
cccz  
peakbolt  
rvierdiiev  
ArbitraryExecution  
Ruhum  
p0wd3r  
0xbepresent  
deadrksezzz  
caventa  
immeas

Inspex  
jprod15  
igungu  
y1cunhui  
BenRai  
m9800  
RaymondFam  
Bauer  
ctf\_sec  
J4de  
Ace-30  
T1MOH  
BowTiedOriole  
tvdung94  
XDZIBEC

lil.eth  
Jigsaw  
dingo  
MalfurionWhitehat  
yy  
0xkazim  
Aymen0909  
0xStalin  
Nyx  
Brenzee  
0xImanini  
nobody2018  
n1punp  
yixxas



# Issue H-1: All allowances to DepositStableCoinToDealer and GeneralRepay can be stolen due to unsafe call

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/428>

## Found by

0x52, 0xkazim, ArbitraryExecution, Bauer, BowTiedOriole, GalloDaSballo, Inspex, Jigsaw, MalfurionWhitehat, XDZIBEC, ctf\_sec, dingo, jprod15, lil.eth, tvdung94, yy

## Summary

DepositStableCoinToDealer.sol and GeneralRepay.sol are helper contracts that allow a user to swap and enter JOJODealer and JUSDBank respectively. The issue is that the call is unsafe allowing the contract to call the token contracts directly and transfer tokens from anyone who has approved the contract.

## Vulnerability Detail

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/stableCoin/DepositStableCoinToDealer.sol#L30-L44>

```
IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
(address approveTarget, address swapTarget, bytes memory data) = abi
.decode(param, (address, address, bytes));
// if usdt
IERC20(asset).approve(approveTarget, 0);
IERC20(asset).approve(approveTarget, amount);
(bool success, ) = swapTarget.call(data);
if (success == false) {
    assembly {
        let ptr := mload(0x40)
        let size := returndatasize()
        returndatacopy(ptr, 0, size)
        revert(ptr, size)
    }
}
```

We can see above that the call is totally unprotected allowing a user to make any call to any contract. This can be abused by calling the token contract and using the allowances of others. The attack would go as follows:

1. User A approves the contract for 100 USDT
2. User B sees this approval and calls depositStableCoin with the swap target as the USDT contract with themselves as the receiver



3. This transfers all of user A USDT to them

## Impact

All allowances can be stolen

## Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/stableCoin/DepositStableCoinToDealer.sol#L23C14-L50>

## Tool used

Manual Review

## Recommendation

Only allow users to call certain whitelisted contracts.

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/5770d15edac41c78d9726f02e988aa8e14601f3e> <https://github.com/JOJOexchange/smart-contract-EVM/commit/94ea554ec1c563e945bd388051f6438826818b47>

**IAm0x52**

Fixes look good. GeneralRepay and DepositStableCoinToDealer now implement contract whitelists



## Issue M-1: JUSD borrow fee rate is less than it should be

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/73>

### Found by

BenRai, RaymondFam, Ruhum, carrotsmuggler, y1cunhui

### Summary

The borrow fee rate calculation is wrong causing the protocol to take less fees than it should.

### Vulnerability Detail

The borrowFeeRate is calculated through `getTRate()`:

`t0Rate` is initialized as  $1e18$  in the test contracts:

`SECONDS_PER_YEAR` is equal to 365 days which is  $60 * 60 * 24 * 365 = 31536000$ :

As time passes, `getTRate()` value will increase. When a user borrows JUSD the contract doesn't save the actual amount of JUSD they borrow, `tAmount`. Instead, it saves the current "value" of it, `t0Amount`:

When you repay the JUSD, the same calculation is done again to decrease the borrowed amount. Meaning, as time passes, you have to repay more JUSD.

Let's say that JUSDBank was live for a year with a borrowing fee rate of 10% ( $1e17$ ). `getTRate()` would then return:  $1e18 + 1e17 * 31536000 / 31536000 = 1.1e18$

If the user now borrows 1 JUSD we get:  $1e6 * 1e18 / 1.1e18 = 909091$  for `t0Amount`. That's not the expected 10% decrease. Instead, it's about 9.1%.

### Impact

Users are able to borrow JUSD for cheaper than expected

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L274-L286> <https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/lib/JOJOConstant.sol#L7> <https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L37>

### Tool used

Manual Review



## Recommendation

Change formula to: `t0Amount = tAmount - tAmount.decimalMul(tRate)` where `t0Rate` is initialized with 0 instead of `1e18`.

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/334fb691eea57a96fb7220e67e31517638725a80>

**IAm0x52**

Fix looks good. Interest is now accumulated with each state update rather than only when the rate is changed.



## Issue M-2: Subaccount#execute lacks payable

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/111>

### Found by

0xImanini, Brenzee, T1MOH, ctf\_sec, immeas, n1punp, nobody2018, p0wd3r, rvierdiev, yixxas

### Summary

Subaccount#execute lacks payable. If value in Subaccount#execute is not zero, it could always revert.

### Vulnerability Detail

Subaccount#execute lacks payable. The caller cannot send the value.

```
function execute(address to, bytes calldata data, uint256 value) external
↳ onlyOwner returns (bytes memory){
    require(to != address(0));
-> (bool success, bytes memory returnData) = to.call{value: value}(data);
    if (!success) {
        assembly {
            let ptr := mload(0x40)
            let size := returndatasize()
            returndatacopy(ptr, 0, size)
            revert(ptr, size)
        }
    }
    emit ExecuteTransaction(owner, address(this), to, data, value);
    return returnData;
}
```

The Subaccount contract does not implement `receive()` payable or `fallback()` payable, so it is unable to receive value (eth) . Therefore, Subaccount#execute needs to add payable.

### Impact

Subaccount#execute cannot work if value != 0.

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/subaccount/Subaccount.sol#L45-L58>





## Tool used

Manual Review

## Recommendation

Add a `receive()` external payable to the contract or `execute()` to add a payable modifier.

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/smart-contract-EVM/commit/64dfd055deae857fa99d4703cdbf7ba1291b8ad>

**IAm0x52**

Fix looks good. `execute()` is now payable and value is check to make sure no ETH is left in the contract



## Issue M-3: It's possible to reset primaryCredit and secondaryCredit for insurance account

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/159>

### Found by

GalloDaSballo, p0wd3r, rvierdiev

### Summary

When because of negative credit after liquidations of another accounts, insurance address doesn't pass `isSafe` check, then malicious user can call `JOJOExternal.handleBadDebt` and reset both `primaryCredit` and `secondaryCredit` for insurance account.

### Vulnerability Detail

insurance account is handled by JOJO team. Team is responsible to top up this account in order to cover losses. When bad debt is handled, then its negative credit value is added to the insurance account. Because of that it's possible that `primaryCredit` of insurance account is negative and `Liquidation._isSafe(state, insurance) == false`.

Anyone can call `JOJOExternal.handleBadDebt` function.

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/lib/Liquidation.sol#L399-L418>

```
function handleBadDebt(Types.State storage state, address liquidatedTrader)
    external
{
    if (
        state.openPositions[liquidatedTrader].length == 0 &&
        !Liquidation._isSafe(state, liquidatedTrader)
    ) {
        int256 primaryCredit = state.primaryCredit[liquidatedTrader];
        uint256 secondaryCredit = state.secondaryCredit[liquidatedTrader];
        state.primaryCredit[state.insurance] += primaryCredit;
        state.secondaryCredit[state.insurance] += secondaryCredit;
        state.primaryCredit[liquidatedTrader] = 0;
        state.secondaryCredit[liquidatedTrader] = 0;
        emit HandleBadDebt(
            liquidatedTrader,
            primaryCredit,
            secondaryCredit
        );
    }
}
```



```
}  
}
```

So it's possible for anyone to call `handleBadDebt` for insurance address, once its `primaryCredit` is negative and `Liquidation._isSafe(state, insurance) == false`. This will reset both `primaryCredit` and `secondaryCredit` variables to 0 and break insurance calculations.

## Impact

Insurance `primaryCredit` and `secondaryCredit` variables are reset.

## Code Snippet

Provided above

## Tool used

Manual Review

## Recommendation

Do not allow `handleBadDebt` call with insurance address.

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/smart-contract-EVM/commit/78c53b4721ae7bb97fb922f78342d0ee4a1825dd>

**IAm0x52**

Fix looks good. Since the order has been changed, clearing bad debt on the insurance account will result it in still having the same debt before and after the call



## Issue M-4: Unable to liquidate USDC blacklisted user's loan due to transferring leftover collateral back in USDC

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/206>

### Found by

Inspex, jprod15, m9800, monrel, peakbolt

### Summary

During the loan liquidation process, any remaining collateral will be swapped to USDC tokens and transferred to the liquidated user. However, if the USDC contract blacklists the liquidated user, the liquidation transaction will be revert. As a result, the user's loan will be unable to be liquidated if they have been blacklisted by the USDC token contract.

### Vulnerability Detail

During the liquidation process, any remaining tokens will be transferred to the owner of the loan. However, if the loan owner has been blacklisted by USDC token, this flow will be reverted due to the code shown below.

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSD Bank.sol#L199-L204>

As a result, users who have been blacklisted by USDC will be unable to liquidate their loan positions during the period of the blacklisting.

### Impact

The liquidation process might DoS due to its reliance on paying back remaining tokens in USDC only. This will error where transferring USDC tokens to blacklisted users can cause the transaction to be reverted, disrupting the liquidation flow. This will result in a bad debt for the platform.

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSD Bank.sol#L199-L204>

### Tool used

Manual Review



## Recommendation

We suggest implementing one or all of the following solutions:

1. Prevent USDC blacklisted users from opening a loan position until they are no longer blacklisted. This can be done by implementing a blacklist check during the borrowing process.
2. Remove the transfer of remaining USDC tokens to the liquidated user during the liquidation flow. Instead, allow the user to withdraw their remaining USDC tokens on their own after the liquidation process is complete.

## Discussion

**JoscelynFarr**

We will allow partial liquidation to avoid this happened.



## Issue M-5: When the `JUSDBank.withdraw()` is to another internal account the `ReserveInfo.isDepositAllowed` is not validated

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/230>

### Found by

Oxbepresent, carrotsmuggler, caventa

### Summary

The internal `withdraw` does not validate if the collateral reserve has activated/deactivated the `isDepositAllowed` variable

### Vulnerability Detail

The `JUSDBank.withdraw()` function has a param called `isInternal` that helps to indicate if the withdraw amount is internal between accounts or not. When the withdraw is internal the `ReserveInfo.isDepositAllowed` is not validated.

```
File: JUSDBank.sol
332:     function _withdraw(
333:         uint256 amount,
334:         address collateral,
335:         address to,
336:         address from,
337:         bool isInternal
338:     ) internal {
...
...
348:         if (isInternal) {
349:             DataTypes.UserInfo storage toAccount = userInfo[to];
350:             _addCollateralIfNotExists(toAccount, collateral);
351:             toAccount.depositBalance[collateral] += amount;
352:             require(
353:                 toAccount.depositBalance[collateral] <=
354:                     reserve.maxDepositAmountPerAccount,
355:                 JUSDErrors.EXCEED_THE_MAX_DEPOSIT_AMOUNT_PER_ACCOUNT
356:             );
...
...
```

In the other hand, the `isDepositAllowed` is validated in the `deposit` function in the code line 255 but the withdraw to internal account is not validated.



```

File: JUSDBank.sol
247:     function _deposit(
248:         DataTypes.ReserveInfo storage reserve,
249:         DataTypes.UserInfo storage user,
250:         uint256 amount,
251:         address collateral,
252:         address to,
253:         address from
254:     ) internal {
255:         require(reserve.isDepositAllowed,
↳ JUSDErrors.RESERVE_NOT_ALLOW_DEPOSIT);

```

Additionally, the `ReserveInfo.isDepositAllowed` can be modified via the `JUSDOperation.delistReserve()` function. So any collateral's deposits can be deactivated at any time.

```

File: JUSDOperation.sol
227:     function delistReserve(address collateral) external onlyOwner {
228:         DataTypes.ReserveInfo storage reserve = reserveInfo[collateral];
229:         reserve.isBorrowAllowed = false;
230:         reserve.isDepositAllowed = false;
231:         reserve.isFinalLiquidation = true;
232:         emit RemoveReserve(collateral);
233:     }

```

## Impact

The collateral's reserve can get deposits via the internal withdraw even when the `Reserve.isDepositAllowed` is turned off making the `Reserve.isDepositAllowed` useless because the collateral deposits can be via internal withdrawals.

## Code Snippet

The internal withdraw code

## Tool used

Manual review

## Recommendation

Add a `Reserve.isDepositAllowed` validation when the withdrawal is to another internal account.



```

File: JUSDBank.sol
    function _withdraw(
        uint256 amount,
        address collateral,
        address to,
        address from,
        bool isInternal
    ) internal {
...
...
        if (isInternal) {
++            require(reserve.isDepositAllowed,
↪ JUSDErrors.RESERVE_NOT_ALLOW_DEPOSIT);
            DataTypes.UserInfo storage toAccount = userInfo[to];
            _addCollateralIfNotExists(toAccount, collateral);
            toAccount.depositBalance[collateral] += amount;
            require(
                toAccount.depositBalance[collateral] <=
                    reserve.maxDepositAmountPerAccount,
                JUSDErrors.EXCEED_THE_MAX_DEPOSIT_AMOUNT_PER_ACCOUNT
            );
...
...

```

## Discussion

### JoscelynFarr

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/4071d470c126bac25b1a391d5dc1582db258280d>

### IAm0x52

Fix looks good. I don't agree with the validity of this issue as it's not really a deposit (no new assets are entering the bank) but more of a transfer. However this fix does prevent this behavior.





## Issue M-6: Lack of burn mechanism for JUSD repayments causes oversupply of JUSD

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/306>

### Found by

Ruhum, igungu, immeas, peakbolt

### Summary

JUSDBank.repay() allow users to repay their JUSD debt and interest by transferring in JUSD tokens. Without a burn mechanism, it will cause an oversupply of JUSD that is no longer backed by any collateral.

### Vulnerability Detail

JUSDBank receives JUSD tokens for the repayment of debt and interest. However, there are no means to burn these tokens, causing JUSD balance in JUSDBank to keep increasing.

That will lead to an oversupply of JUSD that is not backed by any collateral. And the oversupply of JUSD will increase significantly during market due to mass repayments from liquidation.

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L307-L330>

```
function _repay(
    DataTypes.UserInfo storage user,
    address payer,
    address to,
    uint256 amount,
    uint256 tRate
) internal returns (uint256) {
    require(amount != 0, JUSDErrors.REPAY_AMOUNT_IS_ZERO);
    uint256 JUSDBorrowed = user.t0BorrowBalance.decimalMul(tRate);
    uint256 tBorrowAmount;
    uint256 t0Amount;
    if (JUSDBorrowed <= amount) {
        tBorrowAmount = JUSDBorrowed;
        t0Amount = user.t0BorrowBalance;
    } else {
        tBorrowAmount = amount;
        t0Amount = amount.decimalDiv(tRate);
    }
}
```



```
IERC20(JUSD).safeTransferFrom(payer, address(this), tBorrowAmount);
user.t0BorrowBalance -= t0Amount;
t0TotalBorrowAmount -= t0Amount;
emit Repay(payer, to, tBorrowAmount);
return tBorrowAmount;
}
```

## Impact

To maintain its stability, JUSD must always be backed by more than 1 USD worth of collateral.

When there is oversupply of JUSD that is not backed by any collateral, it affects JUSD stability and possibly lead to a depeg event.

## Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L307-L330>

## Tool used

Manual review

## Recommendation

Instead of transferring to the JUSDBank upon repayment, consider adding a burn mechanism to reduce the supply of JUSD so that it will be adjusted automatically.

## Discussion

### JoscelynFarr

Will add burn mechanism in the contract

### JoscelynFarr

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/a72604efba3a9cbce997aefde742be4c5036a039>

### IAm0x52

Fix looks good. Excess JUSD can now be refunded by the owner



## Issue M-7: UniswapPriceAdaptor fails after updating impact

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/364>

### Found by

ast3ros

### Summary

The `impact` variable can have a maximum value of `uint32` (=4.294.967.295) after updating. This is too low and will cause the `UniswapPriceAdaptor#getMarkPrice()` function to revert.

### Vulnerability Detail

When initialized, the `impact` variable is a `uint256`. However, in the `updateImpact` function, the `newImpact` is a `uint32`.

```
function updateImpact(uint32 newImpact) external onlyOwner {  
    emit UpdateImpact(impact, newImpact);  
    impact = newImpact;  
}
```

The new `impact` variable will be too small because in the `getMarkPrice()` function, we need `diff * 1e18 / JOJOPriceFeed <= impact`:

```
require(diff * 1e18 / JOJOPriceFeed <= impact, "deviation is too big");
```

The result of `diff * 1e18 / JOJOPriceFeed <= impact` is a number with `e18` power. It is very likely that it is larger than the `impact` variable which is a `uint32`. The function `getMarkPrice()` will revert.

### Impact

The `UniswapPriceAdaptor` will malfunction and not return the price from Uniswap Oracle.

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/adaptor/uniswapPriceAdaptor.sol#L52>

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/adaptor/uniswapPriceAdaptor.sol#L61-L64>



## Tool used

Manual Review

## Recommendation

Change the newImpact argument from uint32 to uint256.

```
-   function updateImpact(uint32 newImpact) external onlyOwner {  
+   function updateImpact(uint256 newImpact) external onlyOwner {  
       emit UpdateImpact(impact, newImpact);  
       impact = newImpact;  
   }
```

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/smart-contract-EVM/commit/6138d912bdfe7e644b74a182749ab79bb5dc6028> <https://github.com/JOJOexchange/JUSDV1/commit/cc6c7518719406923e4678478b7ea0eebfa0b079>

**JoscelynFarr**

And we also think this is a low issues to update uint32 to uint256

**thangtranth**

Escalate for 10 USDC.

The impact of this issue is that “The UniswapPriceAdaptor will malfunction and not return the price from Uniswap Oracle”. This breaks the functionality of the protocol as explained above and therefore it definitely qualifies as a medium issue. The simplicity of the fix does not imply a low severity issue.

**sherlock-admin**

Escalate for 10 USDC.

The impact of this issue is that “The UniswapPriceAdaptor will malfunction and not return the price from Uniswap Oracle”. This breaks the functionality of the protocol as explained above and therefore it definitely qualifies as a medium issue. The simplicity of the fix does not imply a low severity issue.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.



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

**JoscelynFarr**

Ok

**hrishibhat**

Result: Medium Unique This is a valid medium issue as pointed out by the escalation

**sherlock-admin**

Escalations have been resolved successfully!

Escalation status:

- thangtranth: accepted

**IAm0x52**

Fix looks good. Type of newImpact changed from uint32 to uint256



## Issue M-8: In over liquidation, if the liquidatee has USDC-denominated assets for sale, the liquidator can buy the assets with USDC to avoid paying USDC to the liquidatee

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/369>

### Found by

cccz, monrel

### Summary

In over liquidation, if the liquidatee has USDC-denominated assets for sale, the liquidator can buy the assets with USDC to avoid paying USDC to the liquidatee

### Vulnerability Detail

In JUSDBank contract, if the liquidator wants to liquidate more collateral than the borrowings of the liquidatee, the liquidator can pay additional USDC to get the liquidatee's collateral.

```
} else {
    //      actualJUSD = actualCollateral * priceOff
    //      = JUSDBorrowed * priceOff / priceOff * (1-insuranceFeeRate)
    //      = JUSDBorrowed / (1-insuranceFeeRate)
    //      insuranceFee = actualJUSD * insuranceFeeRate
    //      = actualCollateral * priceOff * insuranceFeeRate
    //      = JUSDBorrowed * insuranceFeeRate / (1- insuranceFeeRate)
    liquidateData.actualCollateral = JUSDBorrowed
        .decimalDiv(priceOff)
        .decimalDiv(JOJOConstant.ONE - reserve.insuranceFeeRate);
    liquidateData.insuranceFee = JUSDBorrowed
        .decimalMul(reserve.insuranceFeeRate)
        .decimalDiv(JOJOConstant.ONE - reserve.insuranceFeeRate);
    liquidateData.actualLiquidatedTO = liquidatedInfo.t0BorrowBalance;
    liquidateData.actualLiquidated = JUSDBorrowed;
}

liquidateData.liquidatedRemainUSDC = (amount -
    liquidateData.actualCollateral).decimalMul(price);
```

The liquidator needs to pay USDC in the callback and the JUSDBank contract will require the final USDC balance of the liquidatee to increase.

```
require(
```



```
IERC20(primaryAsset).balanceOf(liquidated) -  
    primaryLiquidatedAmount >=  
    liquidateData.liquidatedRemainUSDC,  
    JUSDErrors.LIQUIDATED_AMOUNT_NOT_ENOUGH  
);
```

If the liquidatee has USDC-denominated assets for sale, the liquidator can purchase the assets with USDC in the callback, so that the liquidatee's USDC balance will increase and the liquidator will not need to send USDC to the liquidatee to pass the check in the JUSDBank contract.

## Impact

In case of over liquidation, the liquidator does not need to pay additional USDC to the liquidatee

## Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L188-L204>

## Tool used

Manual Review

## Recommendation

Consider banning over liquidation

## Discussion

### JoscelynFarr

That is in our consideration, if the liquidation triggered, there is a possibility for liquidator to liquidate all collaterals, and the remain collateral will return by the USDC to liquidatee

### JoscelynFarr

In fact, I don't understand how the attack occurs

### Trumpero

This issue states that the USDC balance of a liquidated user will be validated as the result of liquidation. However, the liquidator can purchase USDC instead of directly transfer USDC in the callback function (when the liquidated user sells USDC elsewhere). After that, the balance check for liquidation is still fulfilled, but the liquidated user will lose assets.



## hrishibhat

Additional comment from the Watson:

Assume liquidationPriceOff = 5% and ETH : USDC = 2000 : 1. Alice's unhealthy position is borrowed for 100000 JUSD, collateral is 60 ETH, meanwhile Alice sells 7 ETH for 14000 USDC in other protocol. Bob liquidates 60 ETH of Alice's position, Bob needs to pay 100000 JUSD, and  $60 * 2000 - 100000 / 0.95 = 14737$  USDC. In the JOJOFlashLoan callback, Bob sends 100000 JUSD to the contract and buys the 7 ETH that Alice sold in the other protocol (It increases Alice's USDC balance by 14000), and then Bob just send another  $14737 - 14000 = 737$  USDC to Alice to pass the following check

```
require(
    IERC20(primaryAsset).balanceOf(liquidated) -
    primaryLiquidatedAmount >=
    liquidateData.liquidatedRemainUSDC,
    JUSDErrors.LIQUIDATED_AMOUNT_NOT_ENOUGH
);
```

## JoscelynFarr

fix commit: <https://github.com/JOJOexchange/JUSDV1/commit/5918d68be9b5b021691f768da98df5f712ac6edd>

## IAm0x52

Need validation of amount sent to liquidated

## IAm0x52

Fix looks good. Reentrancy exists if \_primaryAsset is also a collateral but team has explicitly stated that this is never the case.





## Issue M-9: FlashLoanLiquidate.JOJOFlashLoan has no slippage control when swapping USDC

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/373>

### Found by

0x52, 0xStalin, Aymen0909, Bauer, Nyx, T1MOH, cccz, peakbolt, rvierdiev

### Summary

FlashLoanLiquidate.JOJOFlashLoan has no slippage control when swapping USDC

### Vulnerability Detail

In both GeneralRepay.repayJUSD and FlashLoanRepay.JOJOFlashLoan, the user-supplied minReceive parameter is used for slippage control when swapping USDC.

```
function JOJOFlashLoan(
    address asset,
    uint256 amount,
    address to,
    bytes calldata param
) external {
    (address approveTarget, address swapTarget, uint256 minReceive, bytes
    ↪ memory data) = abi
        .decode(param, (address, address, uint256, bytes));
    IERC20(asset).approve(approveTarget, amount);
    (bool success, ) = swapTarget.call(data);
    if (success == false) {
        assembly {
            let ptr := mload(0x40)
            let size := returndatasize()
            returndatacopy(ptr, 0, size)
            revert(ptr, size)
        }
    }
    uint256 USDCAmount = IERC20(USDC).balanceOf(address(this));
    require(USDCAmount >= minReceive, "receive amount is too small");
    ...
    function repayJUSD(
        address asset,
        uint256 amount,
        address to,
        bytes memory param
```



```

    ) external {
        IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);
        uint256 minReceive;
        if (asset != USDC) {
            (address approveTarget, address swapTarget, uint256 minAmount, bytes
↪ memory data) = abi
                .decode(param, (address, address, uint256, bytes));
            IERC20(asset).approve(approveTarget, amount);
            (bool success, ) = swapTarget.call(data);
            if (success == false) {
                assembly {
                    let ptr := mload(0x40)
                    let size := returndatasize()
                    returndatacopy(ptr, 0, size)
                    revert(ptr, size)
                }
            }
            minReceive = minAmount;
        }

        uint256 USDCAmount = IERC20(USDC).balanceOf(address(this));
        require(USDCAmount >= minReceive, "receive amount is too small");
    }

```

However, this is not done in FlashLoanLiquidate.JOJOFlashLoan, and the lack of slippage control may expose the user to sandwich attacks when swapping USDC.

## Impact

The lack of slippage control may expose the user to sandwich attacks when swapping USDC.

## Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/flashloanImpl/FlashLoanLiquidate.sol#L46-L78>

## Tool used

Manual Review

## Recommendation

Consider making FlashLoanLiquidate.JOJOFlashLoan use the minReceive parameter for slippage control when swapping USDC.



## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/b0e7d27cf484d9406a267a1b38ac253113101e8e>

**IAm0x52**

Fix looks good. JOJOFlashloan now validates minReceived when swapping



## Issue M-10: JUSDBank users can bypass individual collateral borrow limits

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/403>

### Found by

0x52, Ace-30, GalloDaSballo, J4de, carrotsmugger, peakbolt

### Summary

JUSDBank imposes individual borrow caps on each collateral. The issue is that this can be bypassed due to the fact that withdraw and borrow use different methods to determine if an account is safe.

### Vulnerability Detail

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSDBank.sol#L105-L117>

```
function borrow(
    uint256 amount,
    address to,
    bool isDepositToJOJO
) external override nonReentrant nonFlashLoanReentrant{
    //      t0BorrowedAmount = borrowedAmount / getTORate
    DataTypes.UserInfo storage user = userInfo[msg.sender];
    _borrow(user, isDepositToJOJO, to, amount, msg.sender);
    require(
        _isAccountSafeAfterBorrow(user, getTRate()),
        JUSDErrors.AFTER_BORROW_ACCOUNT_IS_NOT_SAFE
    );
}
```

When borrowing the contract calls `_isAccountSafeAfterBorrow`. This imposes a max borrow on each collateral type that guarantees that the user cannot borrow more than the max for each collateral type. The issues is that withdraw doesn't impose this cap. This allows a user to bypass this cap as shown in the example below.

Example: Assume WETH and WBTC both have a cap of 10,000 borrow. The user deposits \$30,000 WETH and takes a flashloan for \$30,000 WBTC. Now they deposit both and borrow 20,000 JUSD. They then withdraw all their WBTC to repay the flashloan and now they have borrowed 20,000 against \$30000 in WETH



## Impact

Deposit caps can be easily surpassed creating systematic risk for the system

## Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/JUSD Bank.sol#L105-L117>

## Tool used

Manual Review

## Recommendation

Always use `_isAccountSafeAfterBorrow`

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/611ce809ab1c3c300d888053bea6960ed69ec3c3> <https://github.com/JOJOexchange/JUSDV1/commit/0ba5d98aac0e8109f38ebf2382bf84391a7c846b>

**IAm0x52**

Fixes look good. Borrow specific functions have been replaced with the generic checks, preventing this issue



## Issue M-11: `quoteAllAvailablePoolsWithTimePeriod` can be manipulated with low liquidity pools

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/438>

### Found by

ArbitraryExecution, GalloDaSballo, deadrxsezzz

### Summary

`quoteAllAvailablePoolsWithTimePeriod` can be manipulated with low liquidity pools, and there exist Uniswap V3 pools on Arbitrum that JOJO may quote the price from that are low liquidity and therefore manipulatable.

### Vulnerability Detail

The `quoteAllAvailablePoolsWithTimePeriod` function from the `StaticOracle` contract is used in the `getMarkPrice` function of `uniswapPriceAdaptor.sol` to retrieve the weighted arithmetic mean of the tick prices of all applicable Uniswap V3 pools for the given period. However, the returned price can potentially be manipulated if the liquidity of a queried pool is low enough. This is because the arithmetic mean is susceptible to outliers. The potential for Uniswap V3 pools to be manipulated is usually considered a theoretical vulnerability for high-liquidity pools. However, there are specific instances of low liquidity Uniswap V3 pools on Arbitrum that JOJO will attempt to quote a price from, therefore making manipulation a real attack vector.

In one such instance, the deployed `StaticOracle` contract that JOJO intends to use on Arbitrum returns the following three Uniswap V3 pools for the WBTC/USDC pair: `0xac70bD92F89e6739B3a08Db9B6081a923912f73D`, `0xA62aD78825E3a55A77823F00Fe0050F567c1e4EE`, and `0x83450968eC7606F98Df1C170f8C922d55A13f236`. Two of the three pools have low liquidity, which makes the average arithmetic mean of the three pools manipulatable.

### Impact

Manipulating the price of a token used in a perpetual opens up the opportunity for arbitrage on the JOJO protocol which in turn could increase counterparty risk. Additionally, if the price exceeds the allowed difference set by JOJO, this could cause a permanent DOS of the `uniswapPriceAdaptor` and `emergencyOracle` fallback oracle mechanism. Despite this oracle mechanism being the fallback to Chainlink, a permanent DOS of the backup price oracle system should be considered a critical failure.



## Code Snippet

<https://github.com/Mean-Finance/uniswap-v3-oracle/blob/9935263665c5a16f9c385e909bcc6edcc8d56970/solidity/contracts/StaticOracle.sol#L158-L174>

```
function _quote(
    uint128 _baseAmount,
    address _baseToken,
    address _quoteToken,
    address[] memory _pools,
    uint32 _period
) internal view returns (uint256 _quoteAmount) {
    require(_pools.length > 0, 'No defined pools');
    OracleLibrary.WeightedTickData[] memory _tickData = new
    OracleLibrary.WeightedTickData[](_pools.length);
    for (uint256 i; i < _pools.length; i++) {
        (_tickData[i].tick, _tickData[i].weight) = _period > 0
            ? OracleLibrary.consult(_pools[i], _period)
            : OracleLibrary.getBlockStartingTickAndLiquidity(_pools[i]);
    }
    int24 _weightedTick = _tickData.length == 1 ? _tickData[0].tick :
    OracleLibrary.getWeightedArithmeticMeanTick(_tickData);
    return OracleLibrary.getQuoteAtTick(_weightedTick, _baseAmount, _baseToken,
    _quoteToken);
}
```

## Tool used

Manual review.

## Recommendation

JOJO should consider replacing `quoteAllAvailablePoolsWithTimePeriod` with `quoteSpecificPoolsWithTimePeriod` and selecting a subset of Uniswap V3 pools with sufficient liquidity to avoid price manipulation.

## Discussion

JoscelynFarr

fix link: <https://github.com/JOJOexchange/smart-contract-EVM/commit/1dbc9001be667af42952c110e9fdf04fd7826669> <https://github.com/JOJOexchange/JUSDV1/commit/eed86242c2be0cd70e6b412124eb05ed5e3c92dc>

IAm0x52

Fixes look good. Pools are now specified instead of being pulled dynamically



## Issue M-12: chainlinkAdaptor uses the same heartbeat for both feeds which is highly dangerous

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/449>

### Found by

0x52, ast3ros

### Summary

chainlinkAdaptor uses the same heartbeat for both feeds when checking if the data feed is fresh. The issue with this is that the USDC/USD oracle has a 24 hour heartbeat, whereas the average has a heartbeat of 1 hour. Since they use the same heartbeat the heartbeat needs to be slower of the two or else the contract would be nonfunctional most of the time. The issue is that it would allow the consumption of potentially very stale data from the non-USDC feed.

### Vulnerability Detail

See summary

### Impact

Either near constant downtime or insufficient staleness checks

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/adaptor/chainlinkAdaptor.sol#L43-L55>

### Tool used

Manual Review

### Recommendation

Use two separate heartbeat periods

### Discussion

JoscelynFarr





The contract are trying to get the latest price in here:<https://github.com/sherlock-audit/2023-04-jojo/blob/main/smart-contract-EVM/contracts/adaptor/chainlinkAdaptor.sol#LL47C1-L47C1>

And the heartbeat is trying to prevent chainlink stop updating. It is the same as chainlink's heartbeat.

### **iamjakethehuman**

Escalate for 10 USDC I don't think the sponsor properly understood the issue. On Arbitrum, as well as pretty much any other network, different token pairs have different heartbeats. If the oracle gets the latest price for two pairs with different heartbeats, using the same heartbeat variable for validation would cause either one of the following:

1. Oracle will be down (will revert) most of the time.
2. Oracle will allow for stale prices

When validating prices for two different token pairs, two different heartbeats must be used.

### **sherlock-admin**

Escalate for 10 USDC I don't think the sponsor properly understood the issue. On Arbitrum, as well as pretty much any other network, different token pairs have different heartbeats. If the oracle gets the latest price for two pairs with different heartbeats, using the same heartbeat variable for validation would cause either one of the following:

1. Oracle will be down (will revert) most of the time.
2. Oracle will allow for stale prices

When validating prices for two different token pairs, two different heartbeats must be used.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment.

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

### **JoscelynFarr**

<https://github.com/JOJOexchange/smart-contract-EVM/commit/c4270e0dc4da0db56173e39d8b6318e47999a07d> <https://github.com/JOJOexchange/JUSDV1/commit/f1699ae81e81eb190914d1c2ae491a825389daac> fix

### **hrishibhat**

Result: Medium Has duplicates Given that the code uses the same heartbeat to validate both assets, when both assets can have different heartbeats, considering



this issue a valid medium

Sponsor comment:

got it, we will accept this issue

**sherlock-admin**

Escalations have been resolved successfully!

Escalation status:

- [iamjakethehuman](#): accepted

**IAm0x52**

Fix looks good. Contract now uses separate heartbeats for asset and USDC



## Issue M-13: GeneralRepay#repayJUSD returns excess USDC to `to` address rather than `msg.sender`

Source: <https://github.com/sherlock-audit/2023-04-jojo-judging/issues/459>

### Found by

0x52

### Summary

When using `GeneralRepay#repayJUSD` to repay a position on `JUSDBank`, any excess tokens are sent to the `to` address. While this is fine for users that are repaying their own debt this is not good when repaying for another user. Additionally, specifying an excess to repay is basically a requirement when attempting to pay off the entire balance of an account. This combination of factors will make it very likely that funds will be refunded incorrectly.

### Vulnerability Detail

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/flashloanImpl/GeneralRepay.sol#L65-L69>

```
IERC20(USDC).approve(jusdExchange, borrowBalance);
IJUSDEXchange(jusdExchange).buyJUSD(borrowBalance, address(this));
IERC20(USDC).safeTransfer(to, USDCAmount - borrowBalance);
JUSDAmount = borrowBalance;
}
```

As seen above, when there is an excess amount of USDC, it is transferred to the `to` address which is the recipient of the repay. When `to != msg.sender` all excess will be sent to the recipient of the repay rather than being refunded to the caller.

### Impact

Refund is sent to the wrong address if `to != msg.sender`

### Code Snippet

<https://github.com/sherlock-audit/2023-04-jojo/blob/main/JUSDV1/src/Impl/flashloanImpl/GeneralRepay.sol#L32-L73>

### Tool used

Manual Review



## Recommendation

Either send the excess back to the caller or allow them to specify where the refund goes

## Discussion

**JoscelynFarr**

fix link: <https://github.com/JOJOexchange/JUSDV1/commit/7382ce40dd54f0a396fb5d3f13ab3cfede0493e2>

**IAm0x52**

Fix looks good. Excess USDC is now refunded to msg.sender

