SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	JOJO
Prepared by:	Sherlock
Lead Security Expert:	0x52
Dates Audited:	January 12 - January 18, 2024
Prepared on:	February 20, 2024

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: 4103ea69689b62ea60766314578f410fb364c90f

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
3	2

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: All funds can be stolen from JOJODealer

Source: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/7

Found by

0x52, 0xhashiman, T1MOH, bughuntoor, cawfree, giraffe, vvv

Summary

`Funding._withdraw()` makes arbitrary call with user specified params. User can for example make ERC20 to himself and steal funds.

Vulnerability Detail

User can specify parameters `param` and `to` when withdraws:

```
function executeWithdraw(address from, address to, bool isInternal, bytes memory
↪   param) external nonReentrant {
    Funding.executeWithdraw(state, from, to, isInternal, param);
}
```

In the end of `_withdraw()` function address `to` is called with that `bytes param`:

```
    function _withdraw(
        Types.State storage state,
        address spender,
        address from,
        address to,
        uint256 primaryAmount,
        uint256 secondaryAmount,
        bool isInternal,
        bytes memory param
    )
        private
    {
        ...

        if (param.length != 0) {
@>          require(Address.isContract(to), "target is not a contract");
            (bool success,) = to.call(param);
            if (success == false) {
                assembly {
                    let ptr := mload(0x40)
                    let size := returndatasize()
                    returndatacopy(ptr, 0, size)
                    revert(ptr, size)
                }
            }
        }
    }
```

As an attack vector attacker can execute withdrawal of 1 wei to USDC contract and pass calldata to transfer arbitrary USDC amount to himself via USDC contract.

Impact

All funds can be stolen from JOJODealer

Code Snippet

https://github.com/sherlock-audit/2023-12-jojo-exchange-update/blob/ed4a8483da11bcc04ced10de899038bcead087b3/smart-contract-EVM/src/libraries/Funding.sol#L173-L184

Tool used

Manual Review

Recommendation

Don't make arbitrary call with user specified params

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid and i can validate it with POC from report 076}

JoscelynFarr

Fixed PR: https://github.com/JOJOexchange/smart-contract-EVM/commit/763de53a36243490ef46a2c702c5a1480554f286

IAm0x52

Fix looks good. To must now be a whitelisted contract

Issue H-2: FundingRateArbitrage contract can be drained due to rounding error

Source: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/57

Found by

detectiveking

Summary

In the `requestWithdraw`, rounding in the wrong direction is done which can lead to contract being drained.

Vulnerability Detail

In the `requestWithdraw` function in `FundingRateArbitrage`, we find the following lines of code:

```
jusdOutside[msg.sender] -= repayJUSDAmount;
uint256 index = getIndex();
uint256 lockedEarnUSDCAmount = jusdOutside[msg.sender].decimalDiv(index);
require(
     earnUSDCBalance[msg.sender] >= lockedEarnUSDCAmount, "lockedEarnUSDCAmount
↪  is bigger than earnUSDCBalance"
);
withdrawEarnUSDCAmount = earnUSDCBalance[msg.sender] - lockedEarnUSDCAmount;
```

Because we round down when calculating `lockedEarnUSDCAmount`, `withdrawEarnUSDCAmount` is higher than it should be, which leads to us allowing the user to withdraw more than we should allow them to given the amount of JUSD they repaid.

The execution of this is a bit more complicated, let's go through an example. We will assume there's a bunch of JUSD existing in the contract and the attacker is the first to deposit.

Steps:

1. The attacker deposits 1 unit of USDC and then manually sends in another 100 * 10ˆ6 - 1 (not through deposit, just a transfer). The share price / price per earnUSDC will now be $100. Exactly one earnUSDC is in existence at the moment.

2. Next the attacker creates a new EOA and deposits a little over $101 worth of USDC (so that after fees we can get to the $100), giving one earnUSDC to the EOA. The attacker will receive around $100 worth of JUSD from doing this.
3. Attacker calls `requestWithdraw` with `repayJUSDAmount = 1` with the second newly created EOA
4. `lockedEarnUSDCAmount` is rounded down to 0 (since `repayJUSDAmount` is subtracted from jusdOutside[msg.sender]
5. `withdrawEarnUSDCAmount` will be `1`
6. After `permitWithdrawRequests` is called, attacker will be able to withdraw the $100 they deposited through the second EOA (granted, they lost the deposit and withdrawal fees) while only having sent `1` unit of `JUSD` back. This leads to massive profit for the attacker.

Attacker can repeat steps 2-6 constantly until the contract is drained of JUSD.

Impact

All JUSD in the contract can be drained

Code Snippet

https://github.com/JOJOexchange/smart-contract-EVM/blob/main/src/FundingRateArbitrage.sol#L283-L300

Tool used

Manual Review

Recommendation

Round up instead of down

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid and also a dupp of 054 due to the same underlying cause of first deposit attack; but in this the watson explained the exploit scenario of the inflation attack}

nevillehuang

request poc

sherlock-admin

PoC requested from @detectiveking123

Requests remaining: **4**

JoscelynFarr

I think this issue is similar to https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/54

nevillehuang

@JoscelynFarr Seems right, @detectiveking123 do you agree that this seems to be related to a typical first depositor inflation attack.

detectiveking123

@nevillehuang I am not exactly sure how this should be judged.

The attack that I describe here chains two separate vulnerabilities together (one of which is the rounding error and the other which is the same root cause as the share inflation attack) to drain all the funds existing in the contract, which is clearly a high. **It also doesn't rely on any front-running on Arbitrum assumptions, while the other issue does. In fact, no interaction from any other users is necessary for the attacker to drain all the funds.** The exploit that is described in the other issue cannot actually drain all the funds in the contract like this one can, but simply drain user deposits if they can frontrun them.

To clarify, the rounding error that I describe here is different from the rounding error described in the ERC4626 inflation style exploit (so I guess there are two separate rounding errors that optimally should be chained together for this exploit).

Do you still want me to provide a code POC here? I already have an example in the issue description of how the attack can be performed.

nevillehuang

@detectiveking123 Yes, please provide me a coded PoC in 1-2 days so that I can verify the draining impact, because it does share similar root causes of direct donation of funds as the first inflation attack.

detectiveking123

@nevillehuang let me get it to you by tomorrow

detectiveking123

@nevillehuang

```
function testExploit() public {
    jusd.mint(address(fundingRateArbitrage), 5000e6);
    // net value starts out at 0 :)
    console.log(fundingRateArbitrage.getNetValue());

    vm.startPrank(Owner);
    fundingRateArbitrage.setMaxNetValue(10000000e6);
    fundingRateArbitrage.setDefaultQuota(10000000e6);
    vm.stopPrank();

    initAlice();
    // Alice deposits twice
    fundingRateArbitrage.deposit(1);
    USDC.transfer(address(fundingRateArbitrage), 100e6);
    fundingRateArbitrage.deposit(100e6);
    vm.stopPrank();

    vm.startPrank(alice);
    fundingRateArbitrage.requestWithdraw(1);
    fundingRateArbitrage.requestWithdraw(1);
    vm.stopPrank();

    vm.startPrank(Owner);
    uint256[] memory requestIds = new uint256[](2);
    requestIds[0] = 0;
    requestIds[1] = 1;
    fundingRateArbitrage.permitWithdrawRequests(requestIds);
    vm.stopPrank();

    // Alice is back to her initial balance, but now has a bunch of extra JUSD
     ↪  deposited for her into jojodealer!
    console.log(USDC.balanceOf(alice));
    (,uint secondaryCredit,,,) = jojoDealer.getCreditOf(alice);
    console.log(secondaryCredit);
}
```

Add this to `FundingRateArbitrageTest.t.sol`

You will also need to add:

```
function transfer(address to, uint256 amount) public override returns (bool) {
    address owner = _msgSender();
    _transfer(owner, to, amount);
    return true;
}
```

to TestERC20

And change initAlice to:

```
function initAlice() public {
    USDC.mint(alice, 300e6 + 1);
    jusd.mint(alice, 300e6 + 1);
    vm.startPrank(alice);
    USDC.approve(address(fundingRateArbitrage), 300e6 + 1);
    jusd.approve(address(fundingRateArbitrage), 300e6 + 1);
}
```

FYI for this exploit the share inflation is helpful but not necessary. The main issue is the rounding down of `lockedEarnUSDCAmount` in `requestWithdraw`. Even if the share price is 1 cent for example, we will slowly be able to drain JUSD from the contract. An assumption for profitability is that the share price is nontrivial though (so if it's really small it won't be profitable for the attacker b/c of gas fees and deposit fees, though you can still technically drain).

nevillehuang

This issue is exactly the same as #21 and the **original** submission shares the same root cause of depositor inflation to make the attack feasible, given share price realistically won't be of such a low price. I will be duplicating accordingly. Given and subsequent deposits can be drained, I will be upgrading to high severity

@detectiveking123 If you want to escalate feel free,I will maintain my stance here.

IAm0x52

Same fix as #54

JoscelynFarr

Hey @Czar102 @IAm0x52 @detectiveking123 Have already fixed it here https://github.com/JOJOexchange/smart-contract-EVM/commit/beda757204dd242280ec3e46612e97828ea9ffc6

IAm0x52

Fix looks good

Issue M-1: `JUSDBankStorage::getTRate()`,`JUSDBankStorage::accrueRa` are calculated differently, and the data calculation is biased, Causes the `JUSDBank` contract funciton result to be incorrect

Source: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/1

Found by

FastTiger, T1MOH, bitsurfer, dany.armstrong90, detectiveking, joicygiore, rvierdiiev

Summary

```
    function accrueRate() public {
        uint256 currentTimestamp = block.timestamp;
        if (currentTimestamp == lastUpdateTimestamp) {
            return;
        }
        uint256 timeDifference = block.timestamp - uint256(lastUpdateTimestamp);
@>          tRate = tRate.decimalMul((timeDifference * borrowFeeRate) /
↪   Types.SECONDS_PER_YEAR + 1e18);
        lastUpdateTimestamp = currentTimestamp;
    }

    function getTRate() public view returns (uint256) {
        uint256 timeDifference = block.timestamp - uint256(lastUpdateTimestamp);
@>         return tRate + (borrowFeeRate * timeDifference) /
↪   Types.SECONDS_PER_YEAR;
    }
```

`JUSDBankStorage::getTRate()`,`JUSDBankStorage::accrueRate()` are calculated differently, and the data calculation is biased, resulting in the JUSDBank contract not being executed correctly

Vulnerability Detail

The wrong result causes the funciton calculation results of `JUSDBank::_isAccountSafe()`, `JUSDBank::flashLoan()`, `JUSDBank::_handleBadDebt`, etc. to be biased,and all functions that call the relevant function will be biased

Impact

Causes the `JUSDBank` contract funciton result to be incorrect

Code Snippet

https://github.com/sherlock-audit/2023-12-jojo-exchange-update/blob/main/smart-contract-EVM/src/JUSDBankStorage.sol#L53-L67

Tool used

Manual Review

POC

Please add the test code to `JUSDViewTest.t.sol` for execution

```
function testTRateDeviation() public {
    console.log(block.timestamp);
    console.log(jusdBank.lastUpdateTimestamp());
    vm.warp(block.timestamp + 18_356 days);
    jusdBank.accrueRate();
    console.log("tRate value than 2e18:", jusdBank.tRate());
    // block.timestamp for every 1 increment
    vm.warp(block.timestamp + 1);
    uint256 getTRateNum = jusdBank.getTRate();
    jusdBank.accrueRate();
    uint256 tRateNum = jusdBank.tRate();
    console.log("block.timestamp for every 1 increment, deviation:", tRateNum -
 ↪  getTRateNum);
    // block.timestamp for every 1 days increment
    vm.warp(block.timestamp + 1 days);
    getTRateNum = jusdBank.getTRate();
    jusdBank.accrueRate();
    tRateNum = jusdBank.tRate();
    console.log("block.timestamp for every 1 days increment, deviation:",
 ↪  tRateNum - getTRateNum);
}
```

Recommendation

Use the same calculation formula:

```
    function accrueRate() public {
        uint256 currentTimestamp = block.timestamp;
        if (currentTimestamp == lastUpdateTimestamp) {
            return;
        }
        uint256 timeDifference = block.timestamp - uint256(lastUpdateTimestamp);
         tRate = tRate.decimalMul((timeDifference * borrowFeeRate) /
 ↪  Types.SECONDS_PER_YEAR + 1e18);
        lastUpdateTimestamp = currentTimestamp;
    }

    function getTRate() public view returns (uint256) {
        uint256 timeDifference = block.timestamp - uint256(lastUpdateTimestamp);
-       return tRate + (borrowFeeRate * timeDifference) / Types.SECONDS_PER_YEAR;
+       return  tRate.decimalMul((timeDifference * borrowFeeRate) /
 ↪  Types.SECONDS_PER_YEAR + 1e18);
    }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is also valid and a dupp of 016}

JoscelynFarr

After internal discussion, we decide to accrue rate in the view which is `getTRate()` function

JoscelynFarr

Fixed PR: https://github.com/JOJOexchange/smart-contract-EVM/commit/4b591c9fb0a232f784919905752c8e68d32b39ff

IAm0x52

Fix looks good. Math has been updated to use full precision

Issue M-2: Funding#requestWithdraw uses incorrect withdraw address

Source: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/53

Found by

0x52, FastTiger, OrderSol, Varun_05, bughuntoor, dany.armstrong90

Summary

When requesting a withdraw, `msg.sender` is used in place of the `from` address. This means that withdraws cannot be initiated on behalf of other users. This will break integrations that depend on this functionality leading to irretrievable funds.

Vulnerability Detail

Funding.sol#L69-L82

```
function requestWithdraw(
    Types.State storage state,
    address from,
    uint256 primaryAmount,
    uint256 secondaryAmount
)
    external
{
    require(isWithdrawValid(state, msg.sender, from, primaryAmount,
    ↪   secondaryAmount), Errors.WITHDRAW_INVALID);
    state.pendingPrimaryWithdraw[msg.sender] = primaryAmount;
    state.pendingSecondaryWithdraw[msg.sender] = secondaryAmount;
    state.withdrawExecutionTimestamp[msg.sender] = block.timestamp +
    ↪   state.withdrawTimeLock;
    emit RequestWithdraw(msg.sender, primaryAmount, secondaryAmount,
    ↪   state.withdrawExecutionTimestamp[msg.sender]);
}
```

As shown above the withdraw is accidentally queue to `msg.sender` NOT the `from` address. This means that all withdraws started on behalf of another user will actually trigger a withdraw from the `operator`. The result is that withdraw cannot be initiated on behalf of other users, even if the allowance is set properly, leading to irretrievable funds

Impact

Requesting withdraws for other users is broken and strands funds

Code Snippet

Funding.sol#L69-L82

Tool used

Manual Review

Recommendation

Change all occurrences of `msg.sender` in stage changes to `from` instead.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid and a dupp of 082 with minimal impact}

detectiveking123

@nevillehuang do you believe this has enough impact to be considered valid?

JoscelynFarr

Fixed PR: https://github.com/JOJOexchange/smart-contract-EVM/commit/82b5c85c9999ace00265382e7d1bc83036685069

IAm0x52

Fix looks good. Now uses from instead of msg.sender

nevillehuang

@detectiveking123 @JoscelynFarr https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/30#issuecomment-1927840325

Issue M-3: FundRateArbitrage is vulnerable to inflation attacks

Source: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/54

Found by

0x52, Ignite, bughuntoor, detectiveking, giraffe, rvierdiiev

Summary

When index is calculated, it is figured by dividing the net value of the contract (including USDC held) by the current supply of earnUSDC. Through deposit and donation this ratio can be inflated. Then when others deposit, their deposit can be taken almost completely via rounding.

Vulnerability Detail

FundingRateArbitrage.sol#L98-L104

```
function getIndex() public view returns (uint256) {
    if (totalEarnUSDCBalance == 0) {
        return 1e18;
    } else {
        return SignedDecimalMath.decimalDiv(getNetValue(), totalEarnUSDCBalance);
    }
}
```

Index is calculated is by dividing the net value of the contract (including USDC held) by the current supply of totalEarnUSDCBalance. This can be inflated via donation. Assume the user deposits 1 share then donates 100,000e6 USDC. The exchange ratio is now 100,000e18 which causes issues during deposits.

FundingRateArbitrage.sol#L258-L275

```
function deposit(uint256 amount) external {
    require(amount != 0, "deposit amount is zero");
    uint256 feeAmount = amount.decimalMul(depositFeeRate);
    if (feeAmount > 0) {
        amount -= feeAmount;
        IERC20(usdc).transferFrom(msg.sender, owner(), feeAmount);
    }
    uint256 earnUSDCAmount = amount.decimalDiv(getIndex());
    IERC20(usdc).transferFrom(msg.sender, address(this), amount);
    JOJODealer(jojoDealer).deposit(0, amount, msg.sender);
    earnUSDCBalance[msg.sender] += earnUSDCAmount;
    jusdOutside[msg.sender] += amount;
    totalEarnUSDCBalance += earnUSDCAmount;
    require(getNetValue() <= maxNetValue, "net value exceed limitation");
    uint256 quota = maxUsdcQuota[msg.sender] == 0 ? defaultUsdcQuota :
     ↪  maxUsdcQuota[msg.sender];
    require(earnUSDCBalance[msg.sender].decimalMul(getIndex()) <= quota, "usdc
     ↪  amount bigger than quota");
    emit DepositToHedging(msg.sender, amount, feeAmount, earnUSDCAmount);
}
```

Notice earnUSDCAmount is amount / index. With the inflated index that would mean that any deposit under 100,000e6 will get zero shares, making it exactly like the standard ERC4626 inflation attack.

Impact

Subsequent user deposits can be stolen

Code Snippet

FundingRateArbitrage.sol#L258-L275

Tool used

Manual Review

Recommendation

Use a virtual offset as suggested by OZ for their ERC4626 contracts

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid as watson demostrated how this implementation will lead to an inflation attack of the ERC4626 but its medium due to the possibility of it is very low and that front-tun in arbitrum is very unlikely }

detectiveking123

Escalate

I am not completely sure about the judgement here and am therefore escalating to get @Czar102 's opinion on how this should be judged.

I believe that #56 and #21 should be treated as different issues than this one. I am not even sure if this issue and other duplicates are valid, as they rely on the front-running on Arbitrum assumption, which has not been explicitly confirmed to be valid or invalid on Sherlock.

Please take a look at the thread on #56 to better understand the differences. But the TLDR is:

1. This issue requires Arbitrum frontrunning to work, the other one in #56 doesn't
2. The one in #56 takes advantage of a separate rounding error as well to fully drain funds inside the contract

sherlock-admin

> Escalate
>
> I am not completely sure about the judgement here and am therefore escalating to get @Czar102 's opinion on how this should be judged.
>
> I believe that #56 and #21 should be treated as different issues than this one. I am not even sure if this issue and other duplicates are valid, as they rely on the front-running on Arbitrum assumption, which has not been explicitly confirmed to be valid or invalid on Sherlock.
>
> Please take a look at the thread on #56 to better understand the differences. But the TLDR is:
>
> 1. This issue requires Arbitrum frontrunning to work, the other one in #56 doesn't
> 2. The one in #56 takes advantage of a separate rounding error as well to fully drain funds inside the contract

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

JoscelynFarr

Fixed PR: https://github.com/JOJOexchange/smart-contract-EVM/commit/b3cf3d6d6b761059f814efb84af53c5ead4f6446

nevillehuang

@detectiveking123

- I think both this issue and your issue implies that the user needs to be the first depositor, especially the scenario highlighted. This does not explicitly requires a front-run, given a meticulous user can make their own EV calculations.

This issue:

> Assume the user deposits 1 share then donates 100,000e6 USDC. The exchange ratio is now 100,000e18 which causes issues during deposits.

Your issue:

> The execution of this is a bit more complicated, let's go through an example. We will assume there's a bunch of JUSD existing in the contract and the attacker is the first to deposit.

- I think one of the watsons in the discord channel highlighted a valid design of the protocol to mitigate this issue, where withdrawals must be explicitly requested. Because of this, this could possibly be medium severity.

giraffe0x

Disagree that the request/permit design prevents this.

As described in code comments for `requestWithdraw()`: "The main purpose of this function is to capture the interest and avoid the DOS attacks". It is unlikely that withdraw requests are individually scrutinized and manually permitted by owner but automatically executed by bots in batches. Even if the owner does monitor each request, it would be tricky to spot dishonest deposits/withdrawals.

It is better to implement native contract defence a classic ERC4626 attack. Should be kept as a high finding.

nevillehuang

@giraffe0x Again you are speculating on off-chain mechanisms. While it is a valid concern, the focus should be on contract level code logic, and sherlocks assumption is that admin will make the right decisions when permitting withdrawal requests.

Also, here is the most recent example of where first depositor inflation is rated as medium:

https://github.com/sherlock-audit/2023-12-dodo-gsp-judging/issues/55

detectiveking123

@nevillehuang

"I think both this issue and your issue implies that the user needs to be the first depositor, especially the scenario highlighted. This does not explicitly requires a front-run, given a meticulous user can make their own EV calculations."

How would you run this attack without front-running? Share inflation attacks explicitly require front-running

nevillehuang

@detectiveking123 I agree that the only possible reason for this issue to be valid is if

- It so happens that a depositor is the first depositor, and he made his own EV calculations and realized so based on the shares obtained and current exchange ratios (slim chance of happening given front-running is a non-issue on arbitrum, but not impossible)
- If the attack does not explicitly require a first depositor inflation attack (to my knowledge not possible), which I believe none of the **original** issues showed a scenario/explanation it is so (Even #21 and #57 is highlighting a first depositor scenario)

If both of the above scenario does not apply, all of the issues and its duplicates should be low severity.

IAm0x52

Fix looks good. Adds a virtual offset which prevents this issue.

Evert0x

Front-running isn't necessary as the attacker can deposit 1 wei + donate and just wait for someone to make a deposit under 100,000e6.

But this way the attack is still risky to execute as the attacker will lose the donated USDC in case he isn't the first depositor.

However, this can be mitigated if the attacker created a contract the does the 1 wei deposit + donate action in a single transaction BUT revert in case it isn't the first deposit in the protocol.

Planning to reject escalation and keep issue state as is.

detectiveking123

@Evert0x not sure that makes sense, if you just wait for someone then they should just not deposit (it's a user mistake to deposit, they should be informed that they'll retrieve no shares back in the UI).

Czar102

After a discussion with @Evert0x, planning to make it a Medium severity issue – frontend can display information for users not to fall victim to this exploit by displaying a number of output shares. Even though frontrunning a tx can't be done easily (there is no mempool), one can obtain information about a transaction being submitted in another way. Since this puts severe constraints on this being exploitable, planning to consider it a medium severity issue.

detectiveking123

@Czar102 My issue that has been duplicated with this (https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/57) drains the entire contract (clearly a high) and requires no front-running. The purpose of the escalation was primarily to request deduplication.

Czar102

Planning to consider #57 a separate High severity issue. #57 linked this finding together with another bug (ability to withdraw 1 wei of vault token in value for free) to construct a more severe exploit.

Also, planning to consider this issue a Medium severity one, as mentioned above.

deadrosesxyz

With all due respect, this is against the rules

> Issues identifying a core vulnerability can be considered duplicates. Scenario A: There is a root cause/error/vulnerability A in the code. This vulnerability A -> leads to two attack paths:
>
> - B -> high severity path
> - C -> medium severity attack path/just identifying the vulnerability. Both B & C would not have been possible if error A did not exist in the first place. In this case, both B & C should be put together as duplicates.

detectiveking123

It's worth noting that you can still drain the contract with the exploit described in #57 and #21, even without share inflation (The main issue is a rounding issue that allows you to get one more share than intended, so your profit will be the current share value). If the share price is trivial though, the exploiter will likely lose money to gas fees while draining.

This is why I said I'm not sure about the judging of this issue in the initial escalation, as it seems rather subjective.

Czar102

@deadrosesxyz There are two different vulnerabilities requiring two different fixes. In the fragment of the docs you quoted, all B and C are a result of a single vulnerability A, which is not the case here.

Czar102

Planning to make #57 a separate high, I don't see how is #21 presenting the same vulnerability. This issue and duplicates (including #21) will be considered a Medium.

IAm0x52

Why exactly would it be high and this one medium? Both rely on being first depositor (not frontrunning) and inflation

Edit: As stated in the other submission it is technically possible outside of first depositor but profit would be marginal and wouldn't cover gas costs. IMO hard to even call it a different exploit when both have the same prerequisites and same basic attack structure (first depositor and inflation).

Czar102

#57 presents a way to withdraw equivalent of 1 wei of the vault token for free. This issue and duplicates present a way to inflate the share price being the first depositor, and in case of the frontend displaying all needed information, one needs to frontrun a deposit transaction to execute the attack.

Czar102

Result: Medium Has duplicates

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- detectiveking123: accepted

IAm0x52

@Czar102 You should really reconsider this judgement. No way is #57 a high. What is your criteria? You can't make any money it and can't even make any meaningful impact on the holding of the vault unless you are first depositor and inflate the vault. It has no "material impact" on the vault holdings outside of those conditions. Those conditions happen to be the EXACT same as this issue.

IAm0x52

Let's say there is someone who attempts this. Every single withdraw that you do has to be manually approved by admin. You don't think that 1 million withdraws to steal EACH and EVERY SINGLE USDC wouldn't trigger some kind of red flag to the admin that would block that kind of behavior?

IAm0x52

Something else to note. #57 only works if the index is greater than 1e18. As soon as it is less than that this line will no longer return 0 even if you withdraw a single wei.

```
uint256 lockedEarnUSDCAmount = jusdOutside[msg.sender].decimalDiv(index);
```

The statement that it can be used to "drain" the vault is completely inaccurate. There is no way that this should be considered a high. You can only exploit #57 if you rely on inflation. #57 should not be a separate issue.

IAm0x52

Please show me how I can create 10,000 withdrawal requests for $0.01 on Arbitrum to make #57 even remotely possible without inflation

giraffe0x

> Let's say there is someone who attempts this. Every single withdraw that you do has to be manually approved by admin. You don't think that 1 million withdraws to steal EACH and EVERY SINGLE USDC wouldn't trigger some kind of red flag to the admin that would block that kind of behavior?

This is an important point which downgraded/invalidated many other findings. Judges kept reiterating that the assumption is owner will make correct decisions when permitting withdraw requests - rejecting any request that puts the protocol at risk. @Czar102 @nevillehuang

IAm0x52

Something even more wrong with #57. You must withdraw at least the minimum as seen from these lines here:

```
require(
    withdrawEarnUSDCAmount.decimalMul(index) >= withdrawSettleFee, "Withdraw
     ↪  amount is smaller than settleFee"
);
```

Withdrawing a single wei will not work unless the vault has been inflated. Without inflation it doesn't produce any loss of funds PERIOD. This needs to be rejudged and #57 should be a dupe.

nevillehuang

I agree issue #57 should be a duplicate of this based on my comments here and the way i judge it, I think @Czar102 might have possibly made a mistake making #57 a unique high

detectiveking123

First off, apologies for not responding earlier, I was waiting on the green light to share some of the information below.

@IAm0x52, as the Lead Senior Watson, you have audited the latest version of the JOJO contracts (the new, fixed ones as a result of the issues found in this contest). You must therefore agree that the share inflation vulnerability for these fixed contracts has been mitigated.

Let's take a look at the code for these fixed contracts (https://github.com/JOJOexchange/smart-contract-EVM/blob/main/src/FundingRateArbitrage.sol#L99). Even with this virtual offset added, which by the way is the standard, OpenZeppelin recommended way of resolving share inflation, the rounding issue and the associated ability to drain the contracts still exists in the FIXED version of the code. You can figure this out for yourself if you take a look at the deposit function (https://github.com/JOJOexchange/smart-contract-EVM/blob/main/src/FundingRateArbitrage.sol#L260).

The reason the rounding issue still exists is and is profitable for the attacker is because, despite share inflation not being a concern, you are still able to achieve a non-trivial share price. This highlights a fundamental distinction between the share inflation attack and just having a non-trivial share price. The very idea of share inflation is front-running users and donating to make the share price artificially high, thus making them receive 0 shares back. However, a core part of ERC4626 vaults / this JOJO vault is the ability to handle any share price -- whether it is large or small. If users make a bunch of profits, for example, the vault should be able to handle whatever share price results.

This is why OpenZeppelin's recommended solution is to use this concept of virtual offsets, which still allows for a non-trivial share price (as any ERC4626 vault should), but prevents share inflation attacks. The attack in #57 does rely on a non-trivial share price for attacker profitability, but it does not rely on the share inflation attack specifically. The "fixed" version of the contracts show a case where the share inflation attack has been resolved, but this rounding issue lives on and still has the potential to drain the contracts.

A couple of you posted the following from the Sherlock rules:

```
Issues identifying a core vulnerability can be considered duplicates.
Scenario A:
There is a root cause/error/vulnerability A in the code. This vulnerability A ->
↪  leads to two attack paths:

B -> high severity path
C -> medium severity attack path/just identifying the vulnerability.
Both B & C would not have been possible if error A did not exist in the first
↪  place. In this case, both B & C should be put together as duplicates.
```

So in this case the high severity path is #57 and the medium severity path is this issue (#54). However, even when share inflation attack (#54) was resolved, #57 still exists, so the issues should not be put together as duplicates.

Ultimately, I will admit, the interpretation of this specific rule is quite subjective. If you interpret this rule to mean "when the error A is fixed in a reasonable way, then issue B should live on while C should not, otherwise B and C are duplicates", then clearly #54 and #57 are separate issues.

But if you interpret it to mean "if the specific piece of code causing error A is completely removed and all issues relying on it are duplicates", then I am wrong.

I do think the first interpretation is much more fair, but at the end of the day it's the head of judging's call.

IAm0x52

I acknowledge the abuse of rounding exists in two separate spots for this contract. For #54 it exists here:

```
uint256 earnUSDCAmount = amount.decimalDiv(getIndex());
```

Without inflation, this rounding error is a low risk issue. Depositors lose 1 wei each deposit into the contract but that loss is trivial. The only way to make exploit based on it is to use inflation to make the 1 wei being lost into a very large value. Inflation enables this low risk bug to become a higher risk bug.

For #57 the rounding error exists here:

```
uint256 lockedEarnUSDCAmount = jusdOutside[msg.sender].decimalDiv(index);
```

Users can gain 1 wei under certain conditions (index over 1e18, etc.). Withdrawals have minimums and you pay fees on deposits. The vault fees and gas fees makes the system retain all the value "lost" and most of the time retain even more value than that. The only way to extract more value from system than it retains is to have a very high share price, which can only be achieved with inflation. The low risk bug above is now a higher risk bug.

Without inflation both issues have negligible impact and are therefore low. There is a very good reason why issues like this are grouped together. Let's take a hypothetical scenario. Assume there is a way to hijack ownership of the contract. Once you are owner you can destroy the vault in so many ways. You can set the treasury to an address that reverts when receiving USDC. You can set a minimum withdrawal that breaks the vault. You can now break it in at least 10 different ways. Tell me, would it be fair to now count each one of the different ways that the stolen admin can break the contract as a separate high risk vulnerability? To me the root cause of all those things is that owner can be stolen and that is the most fair, otherwise watsons would have to write up 15 reports each. Without admin being stolen those other issues have no impact.

> Ultimately, I will admit, the interpretation of this specific rule is quite subjective. If you interpret this rule to mean "when the error A is fixed in a reasonable way, then issue B should live on while C should not, otherwise B and C are duplicates", then clearly https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/54 and https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/57 are separate issues.

I think the key here is a discussion of impact. I think a better way of framing it would be: "If error A is fixes in a reasonable way, then the IMPACT of issue B should live on while the IMPACT of C should not, otherwise B and C are duplicates." In fact, the rounding in both cases are not changed. Even after inflation is made impossible, the rounding error that causes #54 is still there and users still lose 1 wei each time they deposit! Just without inflation the impact of this lost wei is negligible. Same with #57. The rounding error will exist after inflation is prevented but the impact is now gone.

Sure you can say that after years and years the index could be a point that makes it exploitable but that argument is dismissed in many other vulnerabilities. Take truncation of block.timestamp. Technically after 50 years the timestamp will break but after so long we don't care. Technically addressing inflation doesn't fix either rounding error, but it takes an issue that can be exploit now and make it only exploitable after so long we don't care anymore. Just like the timestamp issue, this contract will be deprecated long before the index is ever big enough to exploit either this issue or #57. Even if that were to occur, withdrawals are all gatekept by admin and they can use deposit fees and minimum withdrawals to prevent any arbitrarily large index from extracting any value from the system.

detectiveking123

@IAm0x52

I am not sure your point makes sense.

You state: "Without inflation both issues have negligible impact and are therefore low."

I have shown you that in the latest, fixed version of the code (which you yourself approved, and thus admitted that share inflation is solved in this version of the code), the rounding issue persists and can be used to drain the contract. **If share inflation is fixed in the recommended way, and the rounding issue still exists and can drain the contract, clearly they should be considered separate issues.**

Will leave the rest up to the head of judging's discretion.

@JoscelynFarr This issue currently affects the latest version of the code on your Github. The recommendation to address it is to round down on the amount of shares the user gets out, rather than up.

IAm0x52

Where have you shown that? Where do you address minimum withdrawals and deposit fees? Assume a minimum withdrawal of 1 USDC (1e6) and a deposit fee of 0.1%. Both are very minimal values and it would take an index of 1000e18 to profit anything, even excluding gas costs. At 10% APR (very generous) it would take over 70 years to get to an index like that. At that point admin can increase minimum to 5 USDC and make it an index of 5000e18. IMO to assume that both of those values are zero (no fee and no minimum) would be gross misconfiguration of the vault by admin.

Where have you addressed that it would take millions of withdrawals to steal any meaningful amount of money? Where have you addressed the gas costs? Where have you addressed that admin would have to approve those millions of withdrawals?

These are the reasons why it is low without inflation because it has no impact.

detectiveking123

@IAm0x52

I am talking about the code here: https://github.com/JOJOexchange/smart-contract-EVM/blob/main/src/FundingRateArbitrage.sol#L99

The exploit here is very simple. The fact that deposit / withdrawal fees will be easily covered is obvious, so I will not include them in the calculations.

1. Deposit $1000. 1000 shares will be minted, for a share price of $1.
2. Let a bunch of other people deposit.
3. Deposit $1 in.
4. Withdraw your $1 by sending in 1 wei of JUSD. You will also have approximately $1 in JUSDBank.
5. Repeat steps 3 and 4 for as long as you want.

Obviously, this attack can be repeated for values higher than $1000 and $1.

IAm0x52

To make that work, that would require an index of 1,000,000e18. How would you get to that index without inflation? You may have a misunderstanding as to how the index works. An index of 1e18 is the starting index of the vault. This means 1 wei of JUSD = 1 wei of USDC. In your example 1 wei of JUSD = 1 USDC (1,000,000 wei) and would therefore require an index of 1,000,000e18.

detectiveking123

@IAm0x52 The 1e18 you are referring to is including the `1e18` factor from SignedDecimalMath right?

Edit: Ah, I know what the point of confusion is. Please click the link I pasted instead of viewing it in your own IDE. It is a different version of the code I am referring to.

IAm0x52

The decimal math is as follows:

```
amount * index / 1e18.
```

So if you have an index of 1e18 then:

```
1 * 1e18 / 1e18 = 1
```

This is why an index of 1e18 means 1 wei JUSD = 1 wei USDC.

1. Deposit $1000. 1000 shares will be minted, for a share price of $1.

This statement here requires an index of 1,000,000e18 and the starting index is 1e18.

detectiveking123

@IAm0x52

Can we agree that the `getIndex` function in the code we are talking about is as follows?

```
function getIndex() public view returns (uint256) {
    return SignedDecimalMath.decimalDiv(getNetValue() + 1, totalEarnUSDCBalance
     ↪  + 1e3);
}
```

IAm0x52

Correct. So if you have a net value of 1 and totalEarnUSDCBalance of 1 then your index would be:

```
1 * 1e18 / 1 = 1e18
```

To be fair though we have to use the pre-audit code which always sets the initial index to 1e18.

detectiveking123

@IAm0x52

Let's focus on this function and the version of the code I linked for now:

```
function getIndex() public view returns (uint256) {
    return SignedDecimalMath.decimalDiv(getNetValue() + 1, totalEarnUSDCBalance
     ↪  + 1e3);
}
```

In the example I gave above, if I deposit
$1000, ‘getNetValue()‘wouldreturn‘1_{0}00_{0}00_{0}00‘$(1000).

`getIndex` would therefore return `1_000_000 * 1e18` or something similar. Do you agree? The share value is now $1 (past decimal math).

Then we do: `uint256 earnUSDCAmount = amount.decimalDiv(getIndex());`, which sets `earnUSDCAmount = 1000`.

Whether or not this version of the code is applicable is a different story we can discuss later, but do you at least agree the version of the code I've linked is exploitable with the rounding error?

IAm0x52

No because during minting it would mint the following amount of shares:

```
1,000,000,000 * 1e18 / 1e18 = 1,000,000,000
```

Therefore index would be:

```
1,000,000,000 * 1e18  / 1,000,000,000 = 1e18
```

By design, index changes only minutely for each deposit and withdraw. Ideally it wouldn't change at all but due to inevitable rounding it can vary by a few wei. That is why the index starts at such a massive value of 1e18.

detectiveking123

@IAm0x52 What do you mean by "during minting"? The assumption here is that `totalEarnUSDCBalance = 0` when the exploiter first deposits.

Your comment:

This also doesn't seem correct? There's a 1e3 in the denominator so it should be 1e15. The math there should be ((1 + 1) * 10ˆ18) / (1 + 1e3) or something.

IAm0x52

Agreed. But all of my references are to pre-audit code since that is the subject of the submission. We can't use post audit code because that is not the target of the contest.

detectiveking123

@IAm0x52 So you agree the post audit code you have audited and agreed solves the share inflation issue is vulnerable to the rounding attack?

IAm0x52

No it's not vulnerable to the rounding attack due to withdrawal minimums and deposit fees.

detectiveking123

@IAm0x52 Okay, my previous exploit assumed that withdrawal minimum and deposit fees were zero. Let's do a concrete example involving them. We will assume that `withdrawalMinimum = 1 USD` and `deposit fee = 0.1%`, like you mentioned in your earlier comment.

Note: This is on the post-audit, fixed codebase.

1. Deposit $1000. 1000 shares will be minted, for a share price of $1. I will pay $1 in deposit fees for this.
2. Let a bunch of other people deposit.
3. Deposit $2 in. I will receive two shares for this. I will pay 0.2 cents in deposit fees for this.
4. Withdraw and send in $1 JUSD + 1 wei of JUSD, which will be rounded to two shares. Profit: around a dollar.
5. Repeat steps 3 and 4 for as long as you want.

Unless someone has challenges regarding the validity of this example, I will rest my case here and leave it to the head of judging. The fact that the issue exists and drains the contract in a version of the codebase that the LW has agreed fixes share inflation proves that this issue is distinct and valid.

IAm0x52

> 1. Deposit $1000. 1000 shares will be minted, for a share price of $1. I will pay $1 in deposit fees for this.

This is again mistaken. Using the changed code our initial index would be:

```
1e18 * (1 + 0) / (0 + 1e3) = 1e15
```

This would mean that the following is minted:

```
1000e6 * 1e18 / 1e15 = 1,000,000e6
```

After deposit our index is now:

```
1e18 * (1 + 1000e6) / (1,000,000e6 + 1e3) = ~1e15
```

Now lets finish the example:

2. Let a bunch of other people deposit
3. Deposit $2. You will receive 2000e6 shares and pay $0.002 in deposit fees
4. Withdraw and send in 1,000,001 wei of JUSD. Which will be rounded to 1,000,001,000 shares. Profit: $0 - Loss: $0.002 (deposit fees)
5. Repeat steps 3 and 4 until you run out of money.

detectiveking123

@IAm0x52

Funny, that made me laugh.

But, apologies; I forgot the order of operations was the other way around. You should send into the contract first.

1. Send in $1000 into the contract. Then deposit $1000. `getIndex()` will return: `1e18 * (1 + 1e9) / (0 + 1e3) ~ 1e6 * 1e18`. As a result, 1000 shares will be minted, and the new share price will be $2. I will pay $1 in deposit fees for this.
2. Let a bunch of other people deposit.
3. Deposit $4 in. I will receive two shares for this. I will pay 0.4 cents in deposit fees for this.
4. Withdraw and send in $2 JUSD + 1 wei of JUSD, which will be rounded to two shares. Profit: around two dollars.
5. Repeat steps 3 and 4 for as long as you want, and make infinite money.

Also happy to just provide a PoC on the post-audit code if it resolves this discussion.

IAm0x52

The problem you run into is that now when another person deposits, they will mint with an index of:

```
1e18 * (1 + 2e9) / (1000 + 1e3) = 5e5 * 1e18 (half the original index)
```

Which will cause the attacker to immediately lose 1000 USDC because supply has the offset enabled. This is why we use the virtual offset because it will prevent all gains by counteracting the inflation. The more people that deposit the more it depresses the exchange rate (and the more the attacker loses) so the users who deposited right away will be able to withdraw and profit from the attacker and now he has lost all his money.

This is the exact purpose of the virtual offset and the reason it is implemented to break inflation attacks. The more they inflate the more they lose to others' deposits, causing a vicious cycle that causes massive loss to the attacker.

detectiveking123

@IAm0x52 check your math there

You said: `1e18 * (1 + 2e9) / (1000 + 1e3) = 5e5 * 1e18 (half the original index)` but this is incorrect

But, despite the math being wrong, I do get your point. The attacker can mitigate this pretty easily though. Just send in $1000initiallyanddepositalargeramount($5000 let's say, so you yourself acquire most of the "cheap" shares). The attacker will then put around $200 of capital at risk for the future ability to drain all deposits.

Though, even if the attacker had to put up the full $1000 at risk to drain all future deposits, it would be worth and a valid attack.

JoscelynFarr

> @IAm0x52
>
> Funny, that made me laugh.
>
> But, apologies; I forgot the order of operations was the other way around. You should send into the contract first.
>
> 1. Send in $1000 into the contract. Then deposit $1000. `getIndex()` will return: `1e18 * (1 + 1e9) / (0 + 1e3) ~ 1e6 * 1e18`. As a result, 1000 shares will be minted, and the new share price will be $2. I will pay $1 in deposit fees for this.
> 2. Let a bunch of other people deposit.
> 3. Deposit $4 in. I will receive two shares for this. I will pay 0.4 cents in deposit fees for this.
> 4. Withdraw and send in $2 JUSD + 1 wei of JUSD, which will be rounded to two shares. Profit: around two dollars.
> 5. Repeat steps 3 and 4 for as long as you want, and make infinite money.
>
> Also happy to just provide a PoC on the post-audit code if it resolves this discussion.

Hey could you provide a PoC on the posit-audit, thanks.

IAm0x52

@JoscelynFarr

Ah I see. The offset being used in the production code is not high enough. In fact it actually doesn't fix the inflation issue at all (either #54 or #57). I will recommend a higher offset. 1e3 can be broken with a donation of 1e9 which is an attainable number. Instead an offset of 1e9 would be more fitting and will break any chance of inflation.

It was my error to approve such an offset. 1e3 does not remediate the possibility of inflation.

detectiveking123

@IAm0x52 It did fix the inflation issue, at least 99.9% of it, but it fixes 0% of the rounding issue. There are degrees to fixing things. Consider that this was a completely reasonable fix to #54, but if you had not known about #57, you would have never proposed a different fix.

@Czar102 will leave the decision up to you now.

JoscelynFarr

Hey @detectiveking123

Could you provide a PoC with the latest codebase? Thank you so much.

detectiveking123

@JoscelynFarr Sure, give me 1-2 days of time since it's a weekday. It should look pretty similar to the PoC in #57 but I'll have to modify it a bit.

IAm0x52

Yeah I'd like to second that request. I cannot get the repeated deposit and withdrawal working on the new code (or on the old code either). In order to get 2 shares of earnUSDC you have to deposit 2 USDC to get 2 EarnUSDC. Then when I withdraw 1 JUSD + 1 wei it pays 2 USDC but now the attackers EarnUSDC is also 0. So I am unable to get any additional USDC out of the vault. I see how your POC works as the primary deposit but I cannot get the repeated deposits and withdrawals to work. Additionally that only seems to work a single time per account, as further withdrawals revert with the message "lockedEarnUSDCAmount is bigger than earnUSDCBalance"

detectiveking123

@IAm0x52 you're not supposed to get additional USDC out of the vault. The profit is in your JOJODealer credit (reread the code a bit if you want to understand why).

See this line of the PoC:

```
(,uint secondaryCredit,,,) = jojoDealer.getCreditOf(alice);
console.log(secondaryCredit);
```

I will look at this further first thing after work tomorrow.

detectiveking123

Also @IAm0x52 yes optimally you should use a new EOA to do each deposit/withdraw iteration

detectiveking123

@IAm0x52 @JoscelynFarr

I succeeded with creating the PoC. For your convenience, I just forked the production Github repo and added my PoC there.

Please run `forge test --match-test "testExploit" -vv`

Link: https://github.com/detectiveking123/smart-contract-EVM

I will also post the PoC script here for other people's convenience, though it will not work without some of the additional setup:

```
function testExploit() public {
    jusd.mint(address(fundingRateArbitrage), 50000e6);
    // net value starts out at 0 :)
    console.log("Initial net value");
    console.log(fundingRateArbitrage.getNetValue());

    vm.startPrank(Owner);
    fundingRateArbitrage.setMaxNetValue(10000000e6);
    fundingRateArbitrage.setDefaultQuota(10000000e6);
    vm.stopPrank();

    initAlice();
    // Alice transfers first
    USDC.transfer(address(fundingRateArbitrage), 1000e6);
    // Alice deposits in
    fundingRateArbitrage.deposit(1000e6);
    // (Alice can feel free to deposit multiple times after this if
    // she wants to mitigate the amount she risks losing up front,
    // but we won't for simplicity in this PoC)
    vm.stopPrank();

    // Share price is a bit over $1
    console.log("Share price");
    console.log(fundingRateArbitrage.getIndex());
    // Someone else makes a deposit
    initBob();
    fundingRateArbitrage.deposit(1000e6);
    vm.stopPrank();

    // Let's assume Carol and Dave are EOAs that belong to Alice
    initCarol();
    // Carol will get 1 share back after depositing $1.01
    fundingRateArbitrage.deposit(1_010_000);
    // Here is where the rounding issue comes in
    // Will just withdraw a full cent because I'm lazy
    fundingRateArbitrage.requestWithdraw(10_000);
    vm.stopPrank();

    // Just want to show it is possible to repeat the rounding issue
    // over and over again to drain the contract
    initDave();
    // Dave will get 1 share back after depositing $1.01
    fundingRateArbitrage.deposit(1_010_000);
    fundingRateArbitrage.requestWithdraw(10_000);
    vm.stopPrank();

    vm.startPrank(Owner);
    uint256[] memory requestIds = new uint256[](2);
    requestIds[0] = 0;
    requestIds[1] = 1;
    fundingRateArbitrage.permitWithdrawRequests(requestIds);
    vm.stopPrank();

    // Carol is back to her initial balance, but now has a bunch of extra JUSD
    ↪  deposited for her into jojodealer, which is her profit!
    console.log("Carol usdc balance");
    console.log(USDC.balanceOf(carol));
    (, uint secondaryCredit, , , ) = jojoDealer.getCreditOf(carol);
    console.log("Carol JOJO dealer credit");
    console.log(secondaryCredit);

    // Dave is also up money
    console.log("Dave usdc balance");
    console.log(USDC.balanceOf(dave));
    (, uint daveSecondaryCredit, , , ) = jojoDealer.getCreditOf(dave);
    console.log("Dave JOJO dealer credit");
    console.log(daveSecondaryCredit);
}
```

Czar102

I think it is clear that issue #57 is separate as a different rounding can be fixed in order to mitigate it. There are some similarities in the exploit path, but the rounding and fixes are entirely different, in separate parts of code.

I stand by my decision to separate #57 from #54.

IIIIIII000

@Czar102 doesn't the admin having to call permitWithdrawRequests() change the severity of 57?

detectiveking123

@IIIIIII000 Looking at the code, it is not so clear cut as "admin has to call permitWithdrawRequests". I believe this is a high -- there are a few points I'd like to make here:

1. If the attacker deposits $1000 at some point and withdraws $1001 a day later unfairly, it doesn't seem noticeable at all (the price of one share is likely barely noticeable to the admin until enough damage has been done). We shouldn't assume an omnipotent admin (otherwise they could just front-run every single hack with some type of withdraw all) but rather one who acts reasonably well. I think in this case you would easily get past an admin who is acting reasonably well.
2. The malicious attacker is not the only one who can run this exploit / who this exploit affects. Consider an ordinary user who notices they can send in a few dollars less of JUSD and receive out the same amount of USDC. They're innocent and haven't done anything wrong, but the admin now has to choose between blocking their request (which leads to a loss of funds for them -- they lose their share in the vault and gain nothing back) or giving them a few extra dollars. It is like choosing between one way to cause a loss of funds or another. It is likely that with the current way the rounding works, even if all malicious requests are blocked (but how do you even know if a request is malicious or not?), regular users would just 'accidentally' drain the vault over time, leaving nothing for the users at the end who didn't withdraw. Overall, this rounding error just completely breaks the withdraw functionality and causes a loss of funds no matter what choice you make here.

IIIIIII000

The rules for Medium state `Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.` and $1 out of $1000 is only 0.1% so that doesn't sound like anything but a small, finite amount. In order to get more out of the protocol, one would have to submit many, many more of these transactions and at that point, I think it's relevant whether the admin is complicit. I agree there's an issue, but am skeptical that the amounts and the preconditions lead to anything more than Medium, and possibly even as little as a Low, depending on how Sherlock judges things, so I'd like to hear reasoning provided by the Sherlock team, so things are clear for future cases like this (since it doesn't seem like a one-off issue that'll never be seen again).

detectiveking123

@IIIIIII000 This depends on which version of the code you're talking about. **In the original version of the code, there can be much more lost than 0.1% lost per withdraw request**. But in the updated, post-audit version of the code, it is around 0.1% per withdraw request. Also consider that these withdraws can be done as many times as one wants.

This is a fundamental issue that breaks the withdraw functionality though; the calculations are just incorrect, which doesn't affect just the exploiter, but also regular users. **As I stated above, even a well-acting admin is forced into a decision between one type of user fund loss or another**.

IIIIIII000

The only thing that matters for these bugs is the original version of the code during the time of the audit. Can you outline what the maximum percentage lost is (just a ballpark estimate of the order of magnitude), so the Sherlock team can answer the question of how the admin having to approve it affects the severity?

detectiveking123

@IIIIIII000 Up to `100%` of the initial capital amount per withdraw request (so the attacker can choose the amount to drain per withdraw).

But again, the withdraw functionality is broken to the point where there can be a loss of user funds regardless of what the admin does.

IIIIIII000

Thanks detectiveking123. @Czar102 can you answer how the admin constraint affects severity (the reasoning based on the rules, not just the final severity answer), so we can properly submit and judge cases like this in the future?

Czar102

I think the point made by @detectiveking123 is an accurate view – an admin doesn't make mistakes when it comes to their interactions, but when it comes to accepting values determined by an in-scope code, I think it's reasonable that a relatively small rounding error may go unnoticed. It seems this "signing off" of an admin on withdrawals is only an external sanity check, and we obviously can't assume it will catch any small discrepancy.

I'd like to note that we have some README questions in works that will allow Watsons to answer the question "What checks are made when calling admin function xyz?" to be able to define the scope of issues with admin interactions more precisely.

JoscelynFarr

Have already fix in here: https://github.com/sherlock-audit/2023-12-jojo-exchange-update-judging/issues/57#issuecomment-1949761945

IAm0x52

> @IllIllI000 Looking at the code, it is not so clear cut as "admin has to call permitWithdrawRequests". I believe this is a high -- there are a few points I'd like to make here:
>
> ```
> 1. If the attacker deposits $1000 at some point and withdraws $1001 a day
>  ↪  later unfairly, it doesn't seem noticeable at all (the price of one
>  ↪  share is likely barely noticeable to the admin until enough damage has
>  ↪  been done). We shouldn't assume an omnipotent admin (otherwise they
>  ↪  could just front-run every single hack with some type of withdraw all)
>  ↪  but rather one who acts reasonably well. I think in this case you would
>  ↪  easily get past an admin who is acting reasonably well.
>
> 2. The malicious attacker is not the only one who can run this exploit /
>  ↪  who this exploit affects. Consider an ordinary user who notices they
>  ↪  can send in a few dollars less of JUSD and receive out the same amount
>  ↪  of USDC. They're innocent and haven't done anything wrong, but the
>  ↪  admin now has to choose between blocking their request (which leads to
>  ↪  a loss of funds for them -- they lose their share in the vault and gain
>  ↪  nothing back) or giving them a few extra dollars. It is like choosing
>  ↪  between one way to cause a loss of funds or another. It is likely that
>  ↪  with the current way the rounding works, even if all malicious requests
>  ↪  are blocked (but how do you even know if a request is malicious or
>  ↪  not?), regular users would just accidentally drain the vault over time,
>  ↪  leaving nothing for the users at the end who didnt withdraw. Overall,
>  ↪  this rounding error just completely breaks the withdraw functionality
>  ↪  and causes a loss of funds no matter what choice you make here.
> ```

This argument doesn't make much sense to me. As pointed out by @IllIllI000 with permitWithdrawRequests() there is a delay on every withdrawal. Inflation is ridiculously easy to see when it happens and both attacks are only is effective with inflation. Lots of eyes are on the vault at launch and anyone could escalate to admin before any funds leave. Admin can also remove all funds from the contract so you are incorrect in saying that it always causes loss of funds, swap to remove USDC or refundJUSD to remove JUSD, then redistribute funds back to appropriate parties. IMO seems very unlikely either would lead to loss even before fixes. Seems judgment is firm but looks like both #54 and #57 should be low, due to this obvious constraint we've overlooked.

Disclaimers

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

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