SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	100x
Prepared by:	Sherlock
Lead Security Expert:	panprog
Dates Audited:	February 2 - February 13, 2024
Prepared on:	March 5, 2024

Introduction

100x on BLAST. Orderbook perpetual DEX built on @Blast_L2. Going live with Blast mainnet. Trade pre-launch perps + move contracts soon.

Scope

Repository: rysk-finance/rysk-ciao-protocol

Branch: sherlock-audit

Commit: c4a8bf14c3527a2ca52c2b1bf9b5742a1f7aac2d

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
7	3

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: `executeWithdrawal` can be called by user after withdrawal timeout to withdraw full balance regardless of unrealized loss creating large bad debt

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/6

Found by

Chinmay, KupiaSec, T1MOH, carrotsmuggler, ck, deadrxsezzz, hash, panprog

Summary

User can initiate withdrawal on-chain via calling `Ciao.requestWithdrawal`. If the protocol's off-chain app doesn't process this request in time (`minimumWithdrawalWaitTime`), user can proceed to call `Ciao.executeWithdrawal` to finish the withdrawal.

The problem is that this function doesn't check any unrealized loss user has, allowing the user to withdraw full balance. If the user has large unrealized loss, this allows to withdraw full balance creating large bad debt for the user and loss of funds for the other protocol users (they won't be able to withdraw everything they've deposited due to bad debt).

Vulnerability Detail

When user calls `Ciao.executeWithdrawal` function, it only checks that the quantity matches the requested quantity and `minimumWithdrawalWaitTime` has passed:

```
} else if (msg.sender == account) {
    // otherwise check the withdrawal receipt validity
    Structs.WithdrawalReceipt memory receipt = withdrawalReceipts[
        subAccount
    ][asset];
    // check the quantity being withdrawn is equal to the quantity requested
    if (quantityE18 != receipt.quantity) revert Errors.WithdrawQuantityInvalid();
    // check the minimum withdrawal wait time has passed
    if (
        receipt.requestTimestamp + minimumWithdrawalWaitTime >
        block.timestamp
    ) revert Errors.MinimumWaitTimeNotPassed();
    _withdraw(account, subAccount, quantityE18, asset);
} else revert Errors.SenderInvalid();
```

Additionally, the `_withdraw` internal function simply reduces user balance without any account health checks, the only check is that withdrawn quantity should not be greater than the user's balance (which doesn't include any unrealized profit or loss):

```
function _withdraw(
    address account,
    address subAccount,
    uint256 quantity,
    address asset
) internal {
    // The account has the full quantity withdrawn from balance
    _changeBalance(subAccount, asset, -int256(quantity));
    // if the balance becomes zero then remove the asset from the set
    if (balances[subAccount][asset] == 0) {
        subAccountAssets[subAccount].remove(asset);
    }
    uint256 quantityRealDecimals = Commons.convertFromE18(
        quantity,
        ERC20(asset).decimals()
    );
    // clear the withdrawal receipt
    delete withdrawalReceipts[subAccount][asset];
    // Transfer asset to sender
    SafeTransferLib.safeTransfer(
        ERC20(asset),
        account,
        quantityRealDecimals
    );
}
```

This means that any user with any unrealized loss can cause bad debt and loss of funds for the other users if off-chain app doesn't do anything for user's withdrawal request, this can happen for a variety of reasons, including:

- user account in not healthy after withdrawal - off-chain app will check this and won't execute the withdrawal, but the user can still force withdraw after withdrawal wait time passes, because the withdrawal request is not cleared;
- off-chain app is down for whatever reason and user simply withdraws whatever he requests ignoring any unrealized loss.

Impact

Any user can withdraw more than he should be allowed due to absence of account health check during on-chain withdrawal execution

Code Snippet

`Ciao.executeWithdrawal` doesn't check account health: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L296-L309

`_withdraw` doesn't check account health either: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L346-L370

Tool used

Manual Review

Recommendation

Re-consider the withdrawal flow. Either use the latest submitted price to check account health, even if it's outdated, or remove the user execution of withdrawal request completely.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Escalate

Some duplicates of this incorrectly describe the issue as simply doing withdrawal as if there is no health check at all, although the health check is done in the off-chain app, so these issues are invalid. The problem described here is in the fact that the user can withdraw himself after the on-chain withdrawal request and withdrawal timeout has passed (and off-chain app didn't pick up the withdrawal request).

The issues which do not mention this and are thus invalid are: #3, #21, #23 #24 provides a very strange scenario which seems invalid, although it does mention the core issue (direct user withdrawal after timeout and lack of health check).

#8 is a different issue, because the scenario (and the fix) are different: this one is still possible even when withdrawal is executed via off-chain app, so user doesn't have to do direct withdrawal for the issue to happen, thus it should be a separate issue. This is escalated separately.

sherlock-admin2

> Escalate
>
> Some duplicates of this incorrectly describe the issue as simply doing withdrawal as if there is no health check at all, although the health check is done in the off-chain app, so these issues are invalid. The problem described here is in the fact that the user can withdraw himself after the on-chain withdrawal request and withdrawal timeout has passed (and off-chain app didn't pick up the withdrawal request).
>
> The issues which do not mention this and are thus invalid are: #3, #21, #23 #24 provides a very strange scenario which seems invalid, although it does mention the core issue (direct user withdrawal after timeout and lack of health check).
>
> #8 is a different issue, because the scenario (and the fix) are different: this one is still possible even when withdrawal is executed via off-chain app, so user doesn't have to do direct withdrawal for the issue to happen, thus it should be a separate issue. This is escalated separately.

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.

chinmay-farkya

> Escalate
>
> Some duplicates of this incorrectly describe the issue as simply doing withdrawal as if there is no health check at all, although the health check is done in the off-chain app, so these issues are invalid. The problem described here is in the fact that the user can withdraw himself after the on-chain withdrawal request and withdrawal timeout has passed (and off-chain app didn't pick up the withdrawal request).
>
> The issues which do not mention this and are thus invalid are: #3, #21, #23 #24 provides a very strange scenario which seems invalid, although it does mention the core issue (direct user withdrawal after timeout and lack of health check).
>
> #8 is a different issue, because the scenario (and the fix) are different: this one is still possible even when withdrawal is executed via off-chain app, so user doesn't have to do direct withdrawal for the issue to happen, thus it should be a separate issue. This is escalated separately.

I would like to add that #24 issue description is factually incorrect. It is based on use of frontrunning to avoid bad debt, but BLAST (or arbitrum acc to contest readme) does not allow frontrunning.

carrotsmuggler2

#23 Mentions that withdrawals can be taken out via the contract directly, bypassing the off-chain checks.

chinmay-farkya

> #23 Mentions that withdrawals can be taken out via the contract directly, bypassing the off-chain checks.

I think #23 instead asserts as if any withdrawals can be taken out without any health checks. And it fails to mention the specific case where this bug actually happens : if the off-chain is not active for a minWaitTime.

Thus #23 is factually incorrect => invalid.

carrotsmuggler2

#23 states `lets users take their collateral out of the system either via the order dispatch or via the contract directly`. The mainissue is that the contract allows direct withdrawals,bypassing offchain checks which is pointed out.

chinmay-farkya

> either via the order dispatch or via the contract directly

And this is not true. Not possible through the order dispatch => factually incorrect.

> The main issue is that the contract allows direct withdrawals ,bypassing offchain checks which is pointed out.

I think this is again false. off-chain checks are only missed in this one particular case of minwaitTime passing, and that is not mentioned in #23.

carrotsmuggler2

The issue clearly states that withdrawals can be done via the contract. Yes there is an incorrect bit regarding order dispatch which wasnt clarified in the docs. But the issue description still cover the main attack vector

iamckn

In response to the challenges to #24, the fact that a user can withdraw collateral at any time without health checks has been identified.

As to the fact that frontrunning is not possible on BLAST/Arbitrum, this doesn't require the user to monitor a mempool. All they need to do is monitor fluctuations in token prices they have positions open for. They would then be able to withdraw their collateral without the debt being updated because as I said in the issue PnL values aren't updated during withdrawals. The issue should therefore remain a valid duplicate according to duplication rules.

Czar102

@cvetanovv would like to get your input on the duplication status of these issues.

cvetanovv

It's really hard for me to decide. Even the Sherlock documentation might tally it differently. On the one hand: A -> vulnerability B -> Very good report and describe everything C -> Just identifying the vulnerability Then they can be grouped together.

On the other hand:

> There is a submission `D` which identifies the core issue but does not clearly describe the impact or an attack path. Then `D` is considered **low**. - Reference.

In the above example, I think `D` are the examples from the panprog escalation.

So rather I think the escalation should be accepted, because on their own if this report did not exist they would be invalid because they say there is no health check but actually there is a health check by the off-chain system.

deadrosesxyz

@cvetanovv I think there has been a mistake:

> Some duplicates of this incorrectly describe the issue as simply doing withdrawal as if there is no health check at all, although the health check is done in the off-chain app, so these issues are invalid. The problem described here is in the fact that the user can withdraw himself after the on-chain withdrawal request and withdrawal timeout has passed (and off-chain app didn't pick up the withdrawal request).

Users can directly make a withdraw through the Ciao contract. There's no off-chain mechanism which monitors it, which can prevent it from happening. The user directly communicates with the contract both to create the request and then execute it.

> on their own if this report did not exist they would be invalid because they say there is no health check but actually there is a health check by the off-chain system.

This is factually incorrect.

chinmay-farkya

> Users can directly make a withdraw through the Ciao contract. There's no off-chain mechanism which monitors it, which can prevent it from happening. The user directly communicates with the contract both to create the request and then execute it.

This is not true. There is a health check in 2 cases by the off-chain app. 1 - user requests withdrawal off-chain and off-chain app calls executeWithdrawal, only if the resultant health is not bad. 2 - user requests withdrawal on-chain on ciao and off-chain app calls executeWithdrawal, only if the resultant health is not bad.

This is clearly mentioned in the docs.

the real issue is only with the 3rd case - user requests on-chain and executes himself on-chain. But that requires the off-chain app to not be working because otherwise any on-chain withdrawal requests are also caught by off-chain app and executed itself (with a health check) : this is done using even emissions as mentioned in docs. The off-chain system monitors on-chain requests and picks up event data and pushes executeWithdrawal calls on its own using that data. And this only arises ie. user can directly execute only when min wait time has passed, this 3rd case and its intricacies are never mentioned in those invalid duplicates (see panprog's escalation for issue numbers).

Those issues fail to find the root cause and instead describe a factually incorrect issue, and should not be duped with this real issue.

deadrosesxyz

The issues clearly describe the users are directly doing the withdrawals themselves without integrating with the off-chain application.

> But that requires the off-chain app to not be working because otherwise any on-chain withdrawal requests are also caught by off-chain app and executed itself (with a health check) : this is done using even emissions as mentioned in docs.

Can you please point to me how after a user a user creates an on-chain request which would make his account unhealthy, the off-chain app can prevent it? It can't. The issues correctly identify the problem.

kjr217

This issue is valid

panprog

Fix Review: Fixed. User is no longer able to call `executeWithdrawal`

sherlock-admin

The Lead Senior Watson signed off on the fix.

Czar102

@panprog any comments? It seems @deadrosesxyz made some counterarguments that haven't been addressed.

Would also like @cvetanovv to share his thoughts.

cvetanovv

@deadrosesxyz:

> Can you please point to me how after a user a user creates an on-chain request which would make his account unhealthy, the off-chain app can prevent it? It can't. The issues correctly identify the problem.

@kjr217 Can you confirm that? If this is true then we should reject the escalation and keep the reports duplicated.

kjr217

This finding is valid, how would the system be able to stop this offchain action. Ive addressed this on multiple occasions, if I made a fix for it, it means I think the finding is valid, anything else is just arguing semantics.

panprog

> Can you please point to me how after a user a user creates an on-chain request which would make his account unhealthy, the off-chain app can prevent it? It can't. The issues correctly identify the problem.

It can - just create `ingresso` transaction with `withdrawal = 1 wei` - it will withdraw tiny amount while resetting the withdrawal receipt. When offchain app reacts to user's on-chain withdrawal request, it can execute withdraw with any amount, not just what the user specified. So it's possible that in case of unhealthy state after withdrawal, the offchain app may simply execute the withdrawal with minimum quantity possible just to clear the withdrawal request so that user can't withdraw on-chain himself.

kjr217

Whilst true, I don't see that as intended, more as a hack tbh. Also if the system is not able to commit messages for whatever reason but is still working then there is a definite issue.

panprog

> Whilst true, I don't see that as intended, more as a hack tbh. Also if the system is not able to commit messages for whatever reason but is still working then there is a definite issue.

Yes, sure. I'm just giving more details into why I think some issues are invalid: they simply assume that any withdrawal request which makes user position unhealthy will execute since there is no on-chain health check, but this assumption is incorrect: offchain app can simply ignore such off-chain requests and "clear" on-chain requests via 1 wei withdrawals. So it is not enough to just say that any withdrawal into unhealthy position will execute and make user position unhealthy: this is invalid.

Czar102

I don't think it's a must to mention a hacky way for the offchain service to reset the withdrawal.

Planning to reject the escalation and leave the duplicates as they are.

Czar102

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: rejected

Issue H-2: `OrderDispatch._matchOrder` incorrectly reduces `order.taker.quantity` by filled quantity for each matched maker order instead of just once

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/7

Found by

Chinmay, hash, panprog

Summary

`OrderDispatch._matchOrder` function is used to match one taker order with 1 or more maker orders. To reuse order signatures, `Crucible` stores the filled quantity of each order, so the real "active" order quantity = order quantity - order filled quantity.

The problem is that taker order quantity is reduced by filled quantity *inside* the maker orders loop, meaning that taker order quantity is reduced multiple times instead of just once:

```
    for (uint j; j < makerLength; j++) {
...
        order.taker.quantity -= crucible.filledQuantitys(order.takerDigest);
        order.maker.quantity -= crucible.filledQuantitys(order.makerDigest);
...
        // handle filled quantitys
        crucible.updateFilledQuantity(
            order.takerDigest,
            order.makerDigest,
            uint128(baseQuantity)
        );
        // emit event to show order matched
        emit Events.OrderMatched();
    }
```

This means that accounting will be completely wrong if taker order is matched with more than 1 maker order (which is very likely for large taker orders).

Vulnerability Detail

`OrderDispatch._matchOrder` function is called from force swap and from match order actions sent by off-chain app. The problem described above happens when at least 3 maker orders are matched with 1 taker order.

Here is an example of what happens with 3 maker orders:

1. Taker = 8 (filled = 0), maker1 = 1, maker2 = 2, maker3 = 5
2. Matching with maker1: taker = 8 - 0. baseQuantity = min(8,1) = 1. filled (taker) = 1
3. Matching with maker2: taker = 8 - 1 = 7. baseQuantity = min(7,2) = 2. filled (taker) = 3
4. Matching with maker3: taker = 7 - 3 = 4. baseQuantity = min(4,5) = 4. filled (taker) = 7

In the end - taker order is filled for a quantity = 7 instead of 8, maker3 is also not fully filled. In such scenario the result is just unexpected (taker order will remain in the orderbook instead of filling completely). However, the following actions which don't expect these user position sizes might do something unexpected. For example, if the following action in the same transaction is withdrawal, and the user is left with a larger position (closing smaller than expected), since health check is only off-chain, and off-chain app didn't expect such user position, on-chain withdrawal will be allowed, even if the user becomes unhealthy/liquidatable after it.

Also, the same code in many scenarios will simply revert. For example:

1. Taker = 11 (filled = 0), maker1 = 5, maker2 = 5, maker3 = 1
2. Matching with maker1: taker = 11 - 0. baseQuantity = min(11,5) = 5. filled (taker) = 5
3. Matching with maker2: taker = 11 - 5 = 6. baseQuantity = min(6,5) = 5. filled (taker) = 10
4. Matching with maker3: taker = 6 - 10 => reverts

This is much more common and much more severe issue, which will basically cause all transactions from off-chain app to `ingresso` to revert, both matching and all the others together with them, meaning the protocol will fail to function completely.

Impact

Any time a taker order is matched with 3+ makers, either the accounting is broken or the matching transaction reverts (and will keep reverting when off-chain app tries to execute it again and again).

If the accounting is broken, the taker and the last maker(s) will have unexpected positions and moreover, since withdrawal on-chain logic doesn't check account health, if order match and withdrawals are combined in the same transaction, due to unexpected positions, the withdrawal will succeed, but the users might remain with larger than expected position causing them to be immediately liquidated or cause bad debt.

Code Snippet

`OrderDispatch._matchOrder` reduces taker order quantity by filled quantity inside the loop for each maker order instead of once:
https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/OrderDispatch.sol#L475-L476

Tool used

Manual Review

Recommendation

Move reduction of `order.taker.quantity` by filled quantities to before the makers loop, reduce `order.taker.quantity` by `baseQuantity` at the same time with increasing filled quantities.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Fix Review: Fixed.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue H-3: Off-chain health check for withdrawal and the other operations are not enough, lack of on-chain health check during withdrawal can cause user position to be unhealthy or in bad debt

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/8

Found by

panprog

Summary

The protocol logic requires off-chain app to check the subaccount health before executing on-chain withdrawals. However, on-chain withdrawal logic ignores unrealized profit and loss as it doesn't check user health. Even though off-chain logic **should** prevent situations when user becomes unhealthy after withdrawal, there are certain scenarios possible when incorrect withdrawals can still happen. That is, if account state changes between offchain calculation and on-chain transaction execution, which can happen for a variety of reasons outside of the offchain app control (pending withdrawal executed on-chain by the user, on-chain permissionless liquidation or some accounting bug happening just before the `ingresso` withdrawal action).

The other operations (such as MatchOrder and ForceSwap) do not check account health either and are vulnerable as well, but the withdrawal operation is the most severe one.

Vulnerability Detail

`OrderDispatcher` action to withdraw calls `Ciao.executeWithdrawal` function, which only checks that quantity to withdraw is not larger than user balance before proceeding with withdrawal:

```
if (quantityE18 == 0 || quantityE18 > balances[subAccount][asset]) revert
 ↪  Errors.WithdrawQuantityInvalid();
// if the caller is the orderDispatch then execute the withdrawal normally
if (msg.sender == _orderDispatch()) {
    _withdraw(account, subAccount, quantityE18, asset);
} else if (msg.sender == account) {
```

The `_withdraw` internal function simply reduces user balance without any account health checks:

```
function _withdraw(
    address account,
    address subAccount,
    uint256 quantity,
    address asset
) internal {
    // The account has the full quantity withdrawn from balance
    _changeBalance(subAccount, asset, -int256(quantity));
    // if the balance becomes zero then remove the asset from the set
    if (balances[subAccount][asset] == 0) {
        subAccountAssets[subAccount].remove(asset);
    }
    uint256 quantityRealDecimals = Commons.convertFromE18(
        quantity,
        ERC20(asset).decimals()
    );
    // clear the withdrawal receipt
    delete withdrawalReceipts[subAccount][asset];
    // Transfer asset to sender
    SafeTransferLib.safeTransfer(
        ERC20(asset),
        account,
        quantityRealDecimals
    );
}
```

This means that off-chain app must ensure user account is healthy after the action, as there is no such on-chain check. However, it's still possible that user account becomes unhealthy or even in bad debt after withdrawal even if off-chain app makes sure it is healthy:

- When several actions are chained together in the same `ingresso` call, intermediate results between these actions might be tricky to calculate, so if for whatever reason the calculations are incorrect, the withdrawal might be allowed off-chain while account will become unhealthy on-chain
- Any on-chain bug (such as the other reported bug with incorrect taker filled quantities applied multiple times when matching with multiple makers) which provides unexpected user positions will become more severe, because from a harmless "causes unexpected position" bug it becomes "causes bad debt and loss of funds" bug.
- Any unexpected transaction, which comes before the `ingresso` transaction, which modifies the user state unexpectedly, which off-chain app didn't expect and thus didn't include in the user account health calculations. Currently, this

can be on-chain liquidation if premissionless liquidations are on, and also `executeWithdrawal` action by the user (if request was pending and not executed by the offchain app with a certain timeout, then user requested off-chain withdrawal, and then front-runs `ingresso` withdrawal transaction with user-initiated `executeWithdrawal` transaction, withdrawing twice).

Impact

It's possible that user account becomes unhealthy (liquidatable) or in a bad debt in certain situations. Some harmless on-chain bugs can become more severe as unexpected user state will cause liquidations and/or bad debt and loss of funds for all users of the protocol.

Code Snippet

`OrderDispatch.withdraw` doesn't do any checks for account health: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/OrderDispatch.sol#L171-L176

`Ciao.executeWithdrawal` doesn't check account health: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L296-L309

`Ciao._withdraw` doesn't check account health either: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L346-L370

Tool used

Manual Review

Recommendation

Do on-chain account health check after the withdrawal, because off-chain app can not ensure that account is healthy after withdrawal in 100% of cases due to unepected things which are out of off-chain app control.

Additionally, do the on-chain account health check after MatchOrder and ForceSwap operations.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Escalate

This issue is different from #6 and its dups, including the fact that the fix for #6 doesn't fix this one! This issue should be a separate high.

#6 is about the fact that `executeWithdrawal` can be called by the user after a timeout without interacting with off-chain app, and can thus create bad debt for his account. This was fixed by removing the ability for the user to call this function directly (only the off-chain app can call it now).

This issue is about the fact that off-chain app can verify that withdrawal doesn't make account unhealthy, but the user account state changes between off-chain app submitting ingresso withdrawal transaction and actual transaction execution (e.g.: account permissionlessly liquidates another account, receiving the position from liquidatee just before ingresso withdrawal transaction, which will now make the account unhealthy due to assumed new position from liquidation).

This issue still remains after the fix of #6, because it's caused by off-chain withdrawal action, not by on-chain withdrawal execution by the user. As such, this should be a separate issue.

Additionally, the same issue happens for `MatchOrder` and `ForceSwap` actions (no on-chain health checks, which can make account unhealthy due to the same issue - something happening with the account state before the ingresso transaction is executed on-chain)

sherlock-admin2

> Escalate
>
> This issue is different from #6 and its dups, including the fact that the fix for #6 doesn't fix this one! This issue should be a separate high.
>
> #6 is about the fact that `executeWithdrawal` can be called by the user after a timeout without interacting with off-chain app, and can thus create bad debt for his account. This was fixed by removing the ability for the user to call this function directly (only the off-chain app can call it now).
>
> This issue is about the fact that off-chain app can verify that withdrawal doesn't make account unhealthy, but the user account state changes between off-chain app submitting ingresso withdrawal transaction and actual transaction execution (e.g.: account permissionlessly liquidates another account, receiving the position from liquidatee just before ingresso withdrawal transaction, which will now make the account unhealthy due to assumed new position from liquidation).

> This issue still remains after the fix of #6, because it's caused by off-chain withdrawal action, not by on-chain withdrawal execution by the user. As such, this should be a separate issue.
>
> Additionally, the same issue happens for `MatchOrder` and `ForceSwap` actions (no on-chain health checks, which can make account unhealthy due to the same issue - something happening with the account state before the ingresso transaction is executed on-chain)

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.

Czar102

@cvetanovv could you let me know what do you think?

cvetanovv

I agree with the escalation. I think we can separate it from the others. The description is different from the others, and the fact that the fix in the other reports doesn't fix this one makes me think we can separate it from the others according to Sherlock judging rules.

panprog

Fix Review: Fixed. No on-chain actions (other than deposit) are allowed, so nothing can front-run the withdrawal transaction.

sherlock-admin

The Lead Senior Watson signed off on the fix.

10xhash

> I agree with the escalation. I think we can separate it from the others. The description is different from the others, and the fact that the fix in the other reports doesn't fix this one makes me think we can separate it from the others according to Sherlock judging rules.

The fix in other reports (#45,#33,#21 etc.) all mention performing on-chain health checks following withdrawal which is same as the recommended fix.

cvetanovv

> The fix in other reports (#45,#33,#21 etc.) all mention performing on-chain health checks following withdrawal which is same as the recommended fix.

None of these reports mention `MatchOrder()` and `ForceSwap()`.

panprog

> The fix in other reports (#45,#33,#21 etc.) all mention performing on-chain health checks following withdrawal which is same as the recommended fix.

The *recommended* fix is the same, but the circumstances of how it can be achieved and attack path (scenario) is completely different. The fact that health isn't checked on-chain is irrelevant if this is done off-chain, the core reason described in this issue is the ability to do something, that allows to do something wrong even when the off-chain check is correct. Final fixes for both #6 and for this issue are not adding on-chain health check, but removing ability to abuse it. So the core reason can be considered "ability to call withdraw by user" for #6 and core reason for this issue - "ability to front-run offchain-initiated withdrawal with some other action", so it's different. In this context, lack of on-chain health check for withdrawal is intended behaviour, not the core reason.

10xhash

I agree that both issues take different path. But for the highest severity both exploit the same issue ie. lack of on-chain health checks for withdrawals, which falls short in both of the attack paths. Since `ability to call withdraw by user` is a high level decision, it would be considered an intended functionality and not as a core-issue for #6. This attack path comes from breaking the assumption that `off-chain checks performed are always based on the correct state` ie. front-running with tx's, which is also described in #9. I am not sure on how this should be judged and wanted to point out the similarities between these issues when making the judgement.

Czar102

I agree with the point in the escalation. Planning to make this a separate High severity issue.

Czar102

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

Issue M-1: User can DOS its own `ingresso` transactions by depositing via off-chain app and front-running its transaction with removed transfer allowance

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/9

The protocol has acknowledged this issue.

Found by

panprog

Summary

User can `ERC20.approve` deposit amount to ciao address, create deposit request via off-chain app, but then front-running `ingresso` transaction with removal or reduction of approved amount to ciao address. This will cause the deposit and whole `ingresso` transaction to revert. This can be repeated again (simply increase approved amount again, possibly re-creating off-chain deposit request again).

Such user behavior will cause lots of reverted transactions (with wasted/lost gas by operator), disruption of the protocol service to the other users (due to failed transactions) and moreover will allow user to selectively censor `ingresso` transactions (allowing what he "likes" to proceed and what he doesn't like to revert).

Vulnerability Detail

`Ciao.deposit` uses ERC20 `transferFrom` to transfer funds from the user:

```
// Pull resources from sender to this contract
SafeTransferLib.safeTransferFrom(
    ERC20(asset),
    account,
    address(this),
    quantity
);
```

This is also called from `OrderDispatch.deposit` action:

```
ICiao(Commons.ciao(address(addressManifest))).deposit(
    depo.account,
    depo.subAccountId,
    depo.quantity,
```

SHERLOCK

```
    depo.asset
);
```

This means that user must first approve Ciao address to spend this amount of asset. This should be checked by off-chain app before it sends `ingresso` transaction with deposit.

However, the user can front-run this transaction and reduce approved amount to cause `ingresso` transaction revert. This will waste operator's gas, and since `ingresso` might include the other transactions as well, this will also cause disruption to protocol service.

Alternatively, especially in L2 networks with no mempool - just figure out the time for off-chain to react to deposit request, and submit approve 0 transaction shortly before this reaction time to ensure `ingresso` transaction executes after approve 0 transaction to make it revert.

Impact

Operator will waste gas on reverted transactions, user can disrupt protocol service and can also selectively censor transactions/actions if they're batched together in 1 transaction with user's deposit action, allowing user to do all kinds of malicious things.

Code Snippet

`OrderDispatch.deposit` calls `Ciao.deposit`: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/OrderDispatch.sol#L133-L138

`Ciao.deposit` uses safeTransfer which requires user approval to ciao address: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L226-L231

Tool used

Manual Review

Recommendation

Consider depositing min(approved, requested) amount (and skipping deposit if approved == 0) when doing it via `ingresso`, which will avoid any reverts during deposits.

Discussion

T1MOH593

Escalate

Duplicate of #11. This issues says how 1 action can revert the whole batch, #11 says that 1 failed action will cause the whole batch to revert.

sherlock-admin2

> Escalate
>
> Duplicate of #11. This issues says how 1 action can revert the whole batch, #11 says that 1 failed action will cause the whole batch to revert.

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.

cvetanovv

Escalate

Escalating on behalf of Chinmay: Invalid because Blast uses a FIFO queue for transactions, frontrunning is not possible at all. “just figure out the time for off-chain to react to deposit request” is a bad assumption, because that is only known to the operator and off-chain admin. There are many other ways that we can block ingresso by frontrunning, but the frontrunning assumption here is impractical otherwise any L2-based design would not be able to function. Historically also, any such issues have been deemed invalid.

sherlock-admin2

> Escalate
>
> Escalating on behalf of Chinmay: Invalid because Blast uses a FIFO queue for transactions, frontrunning is not possible at all. “just figure out the time for off-chain to react to deposit request” is a bad assumption, because that is only known to the operator and off-chain admin. There are many other ways that we can block ingresso by frontrunning, but the frontrunning assumption here is impractical otherwise any L2-based design would not be able to function. Historically also, any such issues have been deemed invalid.

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.

carrotsmuggler2

Escalate

Off chain checks execution before ingresso is called according to the docs.

https://100x.gitbook.io/ciao-protocol-technical-documentation-1/technical-documentation/orderdispatch#actions

Also, Blast is FIFO. Users blocking their own transaction going through is not an issue. no loss of funds, no impact

`reverting the action will desync this state` is a baseless assumption, surely the offchain will check for transaction confirmation before updating state. This is standard in every off-chain integrated apps, like CEXs

sherlock-admin2

> Escalate
>
> Off chain checks execution before ingresso is called according to the docs.
>
> https://100x.gitbook.io/ciao-protocol-technical-documentation-1/technical-documentation/orderdispatch#actions
>
> Also, Blast is FIFO. Users blocking their own transaction going through is not an issue. no loss of funds, no impact
>
> `reverting the action will desync this state` is a baseless assumption, surely the offchain will check for transaction confirmation before updating state. This is standard in every off-chain integrated apps, like CEXs

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.

panprog

Escalate

This is a valid medium.

> Duplicate of #11. This issues says how 1 action can revert the whole batch, #11 says that 1 failed action will cause the whole batch to revert.

These are different issues. #11 can happen because of different things (for example, deposit before liquidation, or the bug reported in the other issue), not just this one, and batching multiple actions which all revert because of one action is the main problem of that.

This issue, on the other hand, is about a specific case of user forcing offchain app `ingresso` transaction to revert, thus losing gas for the offchain app keeper.

> Invalid because Blast uses a FIFO queue for transactions, frontrunning is not possible at all. “just figure out the time for off-chain to react to deposit request” is a bad assumption, because that is only known to the operator and off-chain admin.

You don't need to know time when off-chain app submits transactions, it's pretty obvious it will submit them ASAP after receiving, so you basically submit to offchain via some API and submit your transaction after a small timeout or even at the same time. It's easily possible. The point is not in that your front-run will 100% success, the point is that **it is possible** to do it this way (even if in 1% or 10% cases) and cause these reverted transactions. #11 then multiplies the impact, because multiple other actions might revert along with user's deposit.

> Users blocking their own transaction going through is not an issue. no loss of funds, no impact

First, the offchain app keeper which submits `ingresso` transactions will keep submitting reverted transactions, losing transaction fees in the process. Second, due to #11, user might be able to DOS many other actions (his own, or the other users), because they will be batched together with deposit and all of them will revert. Third, sponsor considers very important to have offchain app state and onchain state to match, and reverting the action will desync this state, causing issues to the app.

sherlock-admin2

> Escalate
>
> This is a valid medium.
>
> > Duplicate of #11. This issues says how 1 action can revert the whole batch, #11 says that 1 failed action will cause the whole batch to revert.
>
> These are different issues. #11 can happen because of different things (for example, deposit before liquidation, or the bug reported in the other issue), not just this one, and batching multiple actions which all revert because of one action is the main problem of that.
>
> This issue, on the other hand, is about a specific case of user forcing offchain app `ingresso` transaction to revert, thus losing gas for the

offchain app keeper.

> > Invalid because Blast uses a FIFO queue for transactions, frontrunning is not possible at all. “just figure out the time for off-chain to react to deposit request” is a bad assumption, because that is only known to the operator and off-chain admin.
>
> You don't need to know time when off-chain app submits transactions, it's pretty obvious it will submit them ASAP after receiving, so you basically submit to offchain via some API and submit your transaction after a small timeout or even at the same time. It's easily possible. The point is not in that your front-run will 100% success, the point is that **it is possible** to do it this way (even if in 1% or 10% cases) and cause these reverted transactions. #11 then multiplies the impact, because multiple other actions might revert along with user's deposit.
>
> > Users blocking their own transaction going through is not an issue. no loss of funds, no impact
>
> First, the offchain app keeper which submits `ingresso` transactions will keep submitting reverted transactions, losing transaction fees in the process. Second, due to #11, user might be able to DOS many other actions (his own, or the other users), because they will be batched together with deposit and all of them will revert. Third, sponsor considers very important to have offchain app state and onchain state to match, and reverting the action will desync this state, causing issues to the app.

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.

chinmay-farkya

> it will submit them ASAP after receiving

If we consider this to be true, then the attacker does not have time to place such a deposit transaction on-chain.

Suppose a user places some deposit order to the off-chain engine. The off-chain engine will select a batch of orders from the pending ones and then create the payloads for them, while checking that the account has given sufficient approval (see docs).

This means the off-chain engine submits a deposit order payload if there is approval at the time, and then immediately pushes the deposit on-chain. Now this ingresso call cannot be frontrun. The possibility that attacker's transaction goes through between the off-chain order validation process and the payloads pushed on-chain is very thin (impossible), because these actions will most probably be bundled (check approval => add to a list of orders => call ingresso, these 3 actions will most probably be bundled and performed in a few ms so the attacker cannot sneak his txn in between this time). So this and #11 both should be low/invalid.

panprog

@chinmay-farkya I'm not sure if you've ever tried actually doing this on-chain, but it's easily possible. I have a lot of first-hand experience in doing exactly this in L2 networks, so technically this is well possible. Timing looks something like this: when you submit transaction to L2 sequencer, it's added to the queue, then your transaction appear on-chain after about 1 second or a bit more depending on current queue size and your server physical proximity to sequencer.

So this brings us to timings like this: T=0.1 sec -> user submits approve(0) transaction to sequencer T=0.2 sec -> user submits deposit(10) transaction to off-chain API T=0.3 sec -> off-chain app verifies that user has approved 10 (note: user's approve(0) transaction is not visible to anyone yet) T=0.3 sec -> off-chain app submits `ingresso` transaction with deposit to sequencer T=1.2 sec -> approve(0) transaction executes and appears on-chain T=1.4 sec -> `ingresso` deposit transaction executes and reverts (note: offchain app didn't have a chance to know that transaction may fail)

Timings might be different. For example, offchain app can take 1-2 seconds to process before submitting, this is easy to analyze by doing test transactions and verifying timings, then you can modify the scenario above to first send deposit transaction to offchain app, then after 1 second submit your approve(0) transaction on-chain, so that the app doesn't see the transaction yet, but when it submits transaction, it's already behind approve(0) in the queue.

carrotsmuggler2

I still dont understand how the core functionality is broken, or user funds are lost, which are the requirements for a sherlock medium. A user front-runs their own deposit. How does this affect the protocol or other users?

panprog

> I still dont understand how the core functionality is broken, or user funds are lost, which are the requirements for a sherlock medium. A user front-runs their own deposit. How does this affect the protocol or other users?

I already said that above:

1. Transaction fees wasted by offchain app operator
2. Actions of the other users are DOS'd if this is done in scale - basically all or most `ingresso` transactions will revert, so not just user's action, but all the

other actions together with it

3. Desync of off-chain app state and on-chain state, as sponsor considers very important to have them in sync (off-chain app's internal state is set before ingresso transaction executes), so that all ingresso transactions must not revert, this by itself already breaks their assumptions and creates issues.

kjr217

The finding is valid, however i will say that the timing between the deposit at offchain and the time to commit is irrelevant, teh offchain engine will be able to see this and remove it from the ingresso payload. The concern in my opinion is a user being able to catch the committer tx after the final approval check gets done by the offchain keeper and it actually hits chain, if a user can get in there then there is an issue.

chinmay-farkya

> however i will say that the timing between the deposit at offchain and the time to commit is irrelevant, teh offchain engine will be able to see this and remove it from the ingresso payload

If this is true, then I think this one, #11 and #10 should be invalid because these kind of orders can be removed from the payload by the off-chain system.

panprog

> The finding is valid, however i will say that the timing between the deposit at offchain and the time to commit is irrelevant, teh offchain engine will be able to see this and remove it from the ingresso payload.

I already gave a technical scenario (above) of how this could easily happen. Basically, there is about 1 second+ timeframe when the transaction is in the sequencer queue and nobody sees it yet, if the offchain app operator submits ingresso transaction at this time interval, approval will 100% happen before the ingresso and offchain app can't do anything about it. I've been testing and doing exactly things like this in optimism and arbitrum mainnet so I can assure that it's much easier to do than it maybe looks like. Even if the operator specifically works against such strategies by randomizing the time it posts the transaction, this is still doable, although at a lower probability. But if the time is not randomized, it's very predictable and very easy to post transaction in such way that offchain app doesn't have a chance to see your approve transaction when it submits the deposit transaction.

Still, I believe this is an issue regardless of technical implementation, because it **can** happen. And offchain app can not remove it from the payload, because the transaction is already submitted at the time user's approval transaction appears on-chain.

kjr217

I agree, this is why we are fixing by having on chain deposits with event listening instead of off chain deposits and as a result adding deposit counts

Czar102

I believe there are strategies for the off-chain service to make this attack statistically unviable, and the submission is based on an assumption of how inefficiently these off-chain services may function.

Also, there is no loss of funds except for a little gas, which is not a valid Medium on Sherlock, especially when connected to the fact that the attacker needs to frontrun by guessing.

Planning to accept only the second escalation and invalidate the issue.

Czar102

That is, unless it was explicitly mentioned that this transaction must not revert.

kjr217

This transaction must not revert

panprog

> That is, unless it was explicitly mentioned that this transaction must not revert.

The exact wording from the docs is

> the ingresso function should ideally never fail as the off-chain system conducts all necessary checks before pushing up the action and rejects any invalid actions before they get to the ingresso. https://100x.gitbook.io/ciao-protocol-technical-documentation-1/technical-documentation/orderdispatch#actions

Further communication with sponsor I think indicates that revert of the `ingresso` transaction is not acceptable risk by the protocol. That's why I agree that #11 can be considered invalid, because `ingresso` transaction must never revert. But any issue describing how `ingresso` transaction might revert should be a valid medium, including this one.

Czar102

After some discussion with the team on the interpretation of the exact wording in the docs, this issue should be considered valid given that the docs stated said that this transaction mustn't revert.

Planning to reject the escalation and leave the issue as is.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- cvetanovv: rejected
- T1MOH593: rejected
- carrotsmuggler2: rejected
- panprog: rejected

Issue M-2: User can DOS liquidation by front-running with a small deposit to cause the account to become `initial margin healthy` after liquidation

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/10

Found by

panprog

Summary

When user is liquidated, the system makes sure it's not "over-liquidated":

```
// check liquidatee's initial health is zero or below. If above, they have been
↪  liquidated for too much
if (_furnace().getSubAccountHealth(liquidateeSubAccount, true) > 0) revert
↪  Errors.LiquidatedTooMuch();
```

A rational liquidator will always liquidate maximum amount to be at exact 0 or just below 0 for liquidatee account health.

This means that a tiny deposit by the user which front-runs the liquidation will cause liquidation to revert due to the line above, keeping user account healthy when it should be liquidated for a tiny cost. This allows user to DOS liquidator and keep unhealthy account active for arbitrary time, potentially letting it drop down into bad debt and loss of funds for the other users of the protocol.

Vulnerability Detail

Scenario for the user:

- Open 2 opposite positions with minimum collateral
- Wait until either position is unhealthy
- Watch for mempool and front-run liquidation transaction with user's tiny deposit transaction
- OR alternatively (especially for L2 chains without mempool) - simply submit multiple small deposit transactions to ensure all liquidation transactions are front-run by user's deposit transactions

Impact

Liquidator(s) (or operator) will waste gas on reverted transactions, user can avoid liquidation and keep unhealthy position active for rather low cost for arbitrary time, which can easily cause bad debt and loss of funds for the other users in the system.

Code Snippet

`Liquidation.liquidateSubAccount` requires that liquidatee is not over-liquidated: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Liquidation.sol#L135-L136

Tool used

Manual Review

Recommendation

Consider a minimum limit on the USD value of the deposit - like for example min of $10 deposit. This will render this strategy useless (while liquidator's revert is still possible due to this, it's not possible to apply it many times and is not cheap).

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Escalate

This should be valid, because it allows the user to keep unhealthy position from liquidation by depositing tiny amounts (a few wei), increasing bad debt risk.

sherlock-admin2

> Escalate
>
> This should be valid, because it allows the user to keep unhealthy position from liquidation by depositing tiny amounts (a few wei), increasing bad debt risk.

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.

chinmay-farkya

My comments on this : I too found this and other issues where frontrunning can be used to DOS, there are actually 6 such cases.

But these issues are not valid for this contest because frontrunning is not a real threat on Blast which is FIFO. The base assumption is invalid, so I think this should remain invalid. The contest readme should take precedence.

panprog

@chinmay-farkya This issue doesn't really require "front-running" as in "watch mempool and submit transaction with higher fees to front-run", but rather you can just spam submit many small deposits with random small time intervals between them - this will make sure that any liquidation which is submitted will have tiny deposit just before it, effectively front-running all attempts to liquidate. Yes, this will cost quite some gas fees, but compared to possible protocol loss this might be small amount. It also mentions Arbitrum, which might make it possible to frontrun as well, so such L2s are not invincible to front-running, even if it's not classic front-running.

chinmay-farkya

@panprog

> you can just spam submit many small deposits with random small time intervals between them

This amount of deposit transactions will require significant gas, because when the liquidation is going to be pushed on-chain needs to be precisely predicted.

Even if liquidation fails due to such a deposit, then the attacker will need to do it repeatedly unlimited times because the off-chain engine will keep pushing the liquidation in next batch of orders, so liquidation can not really be prevented.

Also, I think it would be very difficult to predict when a liq is going to be pushed.

> this will make sure that any liquidation which is submitted will have tiny deposit just before it

This does not affect "any" liquidation, it only affects such a liquidation where the health was going to zero ie. full liq and now it goes > 0 because of the extra deposit, this is really a small subset of liquidations. Other partial liquidations by anyone will still go through for the same position and subaccount. If a user sees such a problem, anyone can solve this by requesting some partial liquidations to the off-chain engine (which is feasible in normal operation => many users wanting to liquidate unhealthy accounts).

In terms of the operator's gas being wasted, the amount of gas wasted by spam deposits will be comparable because of the unpredictability.

> It also mentions Arbitrum, which might make it possible to frontrun as well

I agree that L2s are not invincible to this kind of frontrunning, but the fund loss is very low and attacker's gain is nothing in this specific case. Instead the attacker will also lose gas fees and deposit amount.

Unpredictability of liq order matching as well as unpredictability of pushing the payload to sequencer + numerous spam requests might be required which might cost greater than the protocol's gas losses on L2 + partial liq still possible, I think this should be a low.

chinmay-farkya

I will add a point here, the "spam" requests will also require repeated approval transactions to also frontrun the liquidation (added to the low possibility based on points above), because if a deposit request goes through on-chain before the off-chain system selects and prepares orders, the approval will get used and the user will need to again frontrun with an approval and then with a deposit, micromanaging time such that the sequence be : previous spam deposit => approve => off-chain system checks approval and selects order to be pushed on-chain => another frontrun using spam deposit => liq payload pushed n-chain

I think predicting liq orders in such a way does not have much possibility, also note that the approval amounts and deposit amounts will also need to be changed in every frontrun because the liq order parameters might change with time.

the chances of this being done is extremely low, and partial liquidations will anyways execute all through this time.

panprog

> because if a deposit request goes through on-chain before the off-chain system selects and prepares orders, the approval will get used and the user will need to again frontrun with an approval and then with a deposit

I think you misunderstand the process. It doesn't require approvals, you simply approve max amount beforehand, and then spam deposit small amounts. The order of events will look like this:

- 0.0s On-chain balance = 10e18, maintenance health = -5e18
- 0.1s user submits deposit(10) [1]
- 0.2s off-chain app sees maintenance health and submits liquidation to bring to initial health = -3 [2]
- 0.5s user submits deposit(10) [3]

- 1.0s user submits deposit(10) [4]
- 1.1s deposit(10) [1] executes on-chain, on-chain balance is now 10e18+10
- 1.2s `ingresso` liquidation [2] executes on-chain and reverts, because initial health is now +7 due to user deposit
- 1.2s offchain app re-submits liquidation transaction to liquidate up to initial health = -3 again (with a slightly larger liquidation size to take into account +10 deposit) [5]
- 1.5s user submits deposit(10) [6]
- 1.6s deposit(10) [3] executes on-chain, on-chain balance is now 10e18+20 ...

This way offchain app will be always behind the user's actual on-chain balance, because tiny deposits will keep coming before the ingresso transaction.

chinmay-farkya

@panprog Oh but now I see another problem which I didn't realize before : from the docs

> liquidation can only be done if the health of the account is below maintenance margin

and here, the issue says that user has to bring initial health > 0 because it checks for getSubAccountHealth using isInitial set to true.

Now consider this : user's maintenance margin is < 0, off-chain engine pushes liq order, now even if this gets frontrunned by a deposit, the spam deposit cannot be small 1 wei (as mentioned in original sisue) because it has to push the "initial margin health" > 0 in order to revert due to over-liquidation, which is way below 0 when liq can actually be pushed.

We know that initial margin is above maintenance margin. So in retrospect because of the deposit (which has to be significant amount and not 1 wei), the user position is now healthy above initial margins which is actually good because user has paid significant funds for the margins to be healthy. Now even though this liq order can revert, the off-chain engine will not attempt liquidation anytime soon again because it is doing so only when maintenance margin < 0.

the movement of the position health from initial margin > 0 to maintenance margin < 0 will take the normal route of position health, which is when liq is pushed again and user can again prevent liq by - "being healthy with initial margin".

So user is blocking liquidation by - "being healthy above and beyond the maintenance margin" which does not benefit the attacker and instead benefits the protocol as all such positions will remain healthy. The assertion in original issue that 1 wei of deposit causes DOS is wrong.

So liq is not really DOS'ed because the position becomes anyways healthy far beyond the liq limits (required initial margin health is designed to be > required maintenance margin health), so this issue is invalid.

In summary : preventing liq by being healthy way above limits is not really a problem.

Czar102

If the above comment is true, planning to reject the escalation and leave the issue as is.

Czar102

Also, it seems the report assumed a concrete suboptimal behavior of the liquidator, which makes it invalid. There are strategies for the liquidator which allow them to liquidate the majority of a position without being impacted by any potential behavior of the user.

panprog

> Now consider this : user's maintenance margin is < 0, off-chain engine pushes liq order, now even if this gets frontrunned by a deposit, the spam deposit cannot be small 1 wei (as mentioned in original sisue) because it has to push the "initial margin health" > 0 in order to revert due to over-liquidation, which is way below 0 when liq can actually be pushed.

In real-life - user's maintenance margin will move in large steps. For example, user has position of 10 ETH open. If ETH price moves from $1000 to $999.9, then user's health will drop by 10 * $0.1 = $1 = 1e18. Situations where user's maintenance health is exactly 1 wei is next to impossible, most of the time it will move in large steps, say +5e17 -> -5e17, allowing the user to deposit small amounts to revert liquidations.

As for the:

> the spam deposit cannot be small 1 wei (as mentioned in original sisue) because it has to push the "initial margin health" > 0 in order to revert due to over-liquidation, which is way below 0 when liq can actually be pushed.

This is not true. Here is some example of how small deposit will revert liquidation:

- Maintenance margin = 0.95, initial margin = 0.9
- User has balance of 4 USDC and position of 0.1 ETH with entry and current ETH = $1000 (position PNL=0)
- Mantenance health = 4 + 0.1*(1000*0.95-1000) = -1
- Initial health = 4 + 0.1*(1000*0.9-1000) = -6

- If liquidation occurs at a price of 1000, optimal liquidation quantity = 0.06. After liquidation, liquidatee
- Mantenance health = 4 + 0.04*(1000*0.95-1000) = 2
- Initial health = 4 + 0.04*(1000*0.9-1000) = 0

Now if we frontrun such liquidation (of 0.06) with 1 wei deposit, our maintenance health is still negative before liquidation (-1e18+1), but after liquidation, initial health = 1 wei, thus reverts.

panprog

> Also, it seems the report assumed a concrete suboptimal behavior of the liquidator, which makes it invalid. There are strategies for the liquidator which allow them to liquidate the majority of a position without being impacted by any potential behavior of the user.

Yes, the issue happens only if liquidator liquidates optimally (maximum possible amount), which is very logical thing to do. If liquidator liquidates smaller amounts, then user will have to front-run with the deposit which makes user account maintenance healthy (so a large deposit), which will revert liquidation but is normally intended behaviour for the liquidations in most protocols.

However, in this specific protocol, it's extremely important for the `ingresso` transaction to never revert (as stated in the docs). Both these actions (spam deposits or simply depositing something to make account healthy just before liquidation) will make `ingresso` liquidation transaction revert, which is not acceptable risk according to developers:

> the ingresso function should ideally never fail as the off-chain system conducts all necessary checks before pushing up the action and rejects any invalid actions before they get to the ingresso. https://100x.gitbook.io/ciao-protocol-technical-documentation-1/technical-documentation/orderdispatch#actions

So this issue provides a way to make liquidations revert by frontrunning small deposits, and at the same time provides a way to revert `ingresso` transactions, which is an issue by itself.

chinmay-farkya

In your example,

> User has balance of 4 USDC and position of 0.1 ETH with entry and current ETH = $1000 (position PNL=0)

I doubt that such a position can even be opened because initial health is already < 0 here. If anything, the user needs to have other balances or positions that make up the subaccount's overall initial health > 0 for this position to be opened.

from the 100x docs :

> but it also encourages liquidations to happen gradually, putting less stress on the system and avoiding liquidity issues.

They expect the liquidations to happen in parts, so the optimal quantity to be liquidated has to be calculated correctly by the liquidator/ off-chain engine.

from the 100x docs :

> assumes liquidator is able to sell or hedge assumed positions for profit (otherwise no incentive for them to liquidate)

this incentivizes the liquidator to correctly calculate liquidation quantities such that small health jumps will not move the positions from maintenance < 0 to initial > 0

also from the 100x docs :

> Assumes liquidator has calculated the correct liquidation size (will not floor size if the final state results in an over-liquidation, it will just revert)
>
> If liquidation occurs at a price of 1000, optimal liquidation quantity = 0.06. After liquidation, liquidatee

the optimal liq qauntity is not 0.06 here, in the docs it is mentioned that post-liquidation also the maintenance margin can still be < 0 , and the concept of preventing "over-liquidation" in itself is present to promote liq in parts. so the optimal quantity here will anywhere below 0.06, and it is the responsibility of the liquidator to ensure that small movement in position health (like a 1 wei deposit) cannot cause over-liquidation. Because if you think about the various parameters in determining health, this "over-liquidation" can also be caused by many other reasons (updated funding fees, updated prices, changing PnLs, pending deposits or withdrawals etc. if any of such Action gets executed in that batch before this liq order) => so it is the job of the liquidator to consider all these factors in determining liq quantity.

And even if the liq reverts once, the liquidator can then realize this and try liquidating smaller amounts => which will go through and liq is successful.

panprog

Similar to #9, this one shows how `ingresso` transaction might revert due to user's transaction just before the `ingresso`, and revert of `ingresso` must not happen (is not acceptable risk by the protocol), so this issue should be valid.

panprog

Fix Review: Fixed. If the liquidation transaction is front-run by the deposit, it will ignore liquidatee health now, so the liquidation will not revert.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Czar102

> Similar to #9, this one shows how `ingresso` transaction might revert due to user's transaction just before the `ingresso`, and revert of `ingresso` must not happen (is not acceptable risk by the protocol), so this issue should be valid.

If that's the only argument guarding the validity of this issue, planning to duplicate this with #9 (issue is that it's possible for one of the actions to revert and these are not independent).

Czar102

After some more thoughts, I believe this should be a unique Medium severity issue, not a duplicate of #9.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

Issue M-3: `feeRecepient` and `insurance` special addresses might cause incorrect accounting and can potentially cause bad debt

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/25

Found by

KupiaSec, T1MOH, hash, panprog

Summary

The protocol has 2 special addresses: `feeRecepient` and `insurance`, which receive fees from trading and liquidations. They receive their fees via `Ciao.incrementFee` function, which doesn't take debt into account. This means that if any of these accounts has debt in core collateral token for whatever reason, `incrementFee` will cause the account to have both asset balance and debt. If the balance is fully withdrawn, the core collateral asset is removed from the account and is no longer calculated in account health calculation. Such account might have core collateral debt which will be ignored and thus the account health calculations will be incorrect, allowing account to take more risk than it should, potentially causing bad debt and loss of funds for the other users.

Vulnerability Detail

The `Ciao.incrementFee` function simply increases recepient's asset balance without taking into account debt:

```
    function incrementFee(
        address asset,
        uint256 fee,
        address recipient
    ) external nonReentrant {
        // check that the caller is the order dispatch or liquidation
        _isBalanceUpdater();
        subAccountAssets[recipient].add(asset);
        _changeBalance(recipient, asset, int256(fee));
    }
...
    function _changeBalance(
        address subAccount,
        address asset,
        int256 change
    ) internal {
        int256 balanceBefore = int256(balances[subAccount][asset]);

        if (change > 0) {
            balances[subAccount][asset] += uint256(change);
        } else {
            balances[subAccount][asset] -= uint256(-change);
        }
```

This function is called for `feeRecepient` and `quote` asset in `OrderDispatch`, and `quote` asset can be core collateral asset:

```
ciao.incrementFee(
    product.quoteAsset,
    uint256(takerFee + makerFee),
    ciao.feeRecipient()
);
```

It is also called for `insurance` recepient for core collateral address in `Liquidation`:

```
ciao.incrementFee(
    ciao.coreCollateralAddress(),
    vars.liquidationFees,
    ciao.insurance()
);
```

If either of these addresses has core collateral debt at the time of `incrementFee` call - it will not decrease debt, but will simply increase balance. When `withdraw` is called for this account with full amount, core collateral asset will be removed from the assets list:

```
    function _withdraw(
        address account,
        address subAccount,
        uint256 quantity,
        address asset
    ) internal {
        // The account has the full quantity withdrawn from balance
        _changeBalance(subAccount, asset, -int256(quantity));
        // if the balance becomes zero then remove the asset from the set
        if (balances[subAccount][asset] == 0) {
            subAccountAssets[subAccount].remove(asset);
        }
...
```

This means account will have core collateral debt, but core collateral asset will be absent from the account's asset list. When calculating account health, it will ignore core collateral and will not count the debt:

```
        tempVars.spotAssets = _ciao().getSubAccountAssets(subAccount);
        tempVars.assetsLen = tempVars.spotAssets.length;
...
        for (uint i = 0; i < tempVars.assetsLen; i++) {
            address spotAssetAddress = tempVars.spotAssets[i];
            uint32 spotProductId = _productCatalogue()
                .baseAssetQuoteAssetSpotIds(
                    spotAssetAddress,
                    _ciao().coreCollateralAddress()
                );
            uint256 spotBalance = _ciao().balances(
                subAccount,
                spotAssetAddress
            );
            if (spotProductId == CORE_COLLATERAL_INDEX) {
                health +=
                    int256(spotBalance) -
                    int256(_ciao().coreCollateralDebt(subAccount));
                continue;
            }
```

Since `getSubAccountAssets` won't return core collateral asset, core collateral `balance - debt` will never be added to health.

Impact

Incorrect account health calculation for specific accounts, allowing accounts to take on any debt which is not calculated in the account health, potentially creating bad debt and loss of funds for all protocol users.

Code Snippet

`Ciao.incrementFee` doesn't account for core collateral debt, simply increasing asset balance: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Ciao.sol#L177-L186

Tool used

Manual Review

Recommendation

Check that `incrementFee` asset is core collateral, and settle core collateral balance instead of updating balance in such case.

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

cvetanovv

Escalate

Escalating on behalf of Chinmay: Invalid. The feerecipient and insuranec addresses are protocol owned addresses, these are not for trading. There is no way these are gonna be used for trading and thus cannot acquire core debt. These are trusted addresses.

sherlock-admin2

> Escalate
>
> Escalating on behalf of Chinmay: Invalid. The feerecipient and insuranec addresses are protocol owned addresses, these are not for trading. There is no way these are gonna be used for trading and thus cannot acquire core debt. These are trusted addresses.

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.

carrotsmuggler2

Escalated

Same as #26 and only affects protocol owned accounts. Unrealistic to think protocol owned treasury accounts will be used for large transactions. Should be low. No losses to other users.

sherlock-admin2

> Escalated
>
> Same as #26 and only affects protocol owned accounts. Unrealistic to think protocol owned treasury accounts will be used for large transactions. Should be low. No losses to other users.

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.

panprog

Docs don't mention that `feeRecepient` and `insurance` are trusted addresses, these can be any addresses. But I agree that it's a borderline issue and can be considered both ways.

chinmay-farkya

> Docs don't mention that `feeRecepient` and `insurance` are trusted addresses, these can be any addresses

Yes, but these are still protocol-owned addresses and the admin is trusted. Plus, these addresses have been mentioned int the docs to have a particular purpose of just collecting fees, these can not be expected to be involved in trading perps.

Both this and #26 should be invalid.

Czar102

@carrotsmuggler2 @chinmay-farkya can you show me where is it mentioned in the docs and what is the exact wording?

chinmay-farkya

@Czar102

> However, the protocol receives 25% of the profit that liquidators generate, and these fees are deposited into the insurance fund to protect protocol health moving forward

which means insurance is a protocol-owned address. here

> // fee recipient for receiving all fees, can withdraw like any other user to harvest fees

comment on Ciao.sol # L38, and this address is set by the protocol in initialize function at Ciao.sol #L65. Ciao also has a setFeeRecipient function where only the admin can set it.

Czar102

I believe this is a valid finding. The addresses are not mentioned as trusted in the README, and the protocol should be safe to set the fee recipient to any untrusted actor.

Planning to reject the escalation and leave the issue as is.

chinmay-farkya

> I believe this is a valid finding. The addresses are not mentioned as trusted in the README, and the protocol should be safe to set the fee recipient to any untrusted actor.
>
> Planning to reject the escalation and leave the issue as is.

I agree that it is not mentioned in the README, but based on the context of the protocol I'd say that the "protocol fees" and "insurance fund" cannot go to an untrusted actor, for the insurance address it is even mentioned in docs that -

> the protocol receives 25% of the profit that liquidators generate, and these fees are deposited into the insurance fund

I think we should consider common sense here, because some details are always left out in the README and the context of the protocol implies that these are going to be protocol-owned addresses.

panprog

Fix Review: Fixed. When fee receiver gets fee in core collateral asset, it's settled instead of simply increasing balance.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Czar102

I thought a lot about this issue, and I still think it should remain valid – even though these addresses are owned by protocol-owned addresses, they are not trusted. That means that these addresses can be owned e.g. by a single member of the team, who is not trusted by themselves (only as a member of a multisig).

To my knowledge, #47 also presents a way for this exploit to occur without the control of any of these two addresses.

Czar102

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- cvetanovv: rejected
- carrotsmuggler2: rejected

Issue M-4: Discontinued/retired products or spot assets can be stolen/exploited

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/27

Found by

KupiaSec, carrotsmuggler, hash, panprog

Summary

Discontinued/retired products or spot assets can be stolen/exploited

Vulnerability Detail

The protocol implements a `ProductCatalogue` contract which keeps track of the various products that can be traded. This limits the `deposit` function from depositing random assets into the protocol. A `productId` is set, along witht he base and quote asset addresses.

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/ProductCatalogue.sol#L44-L51

Issue is that there is no proper way to discontinue a product. The protocol implements a `changeProductTradeability` function to change the `isProductTradeable` value but this is not checked anywhere in the protocol.

This means if the product is to be discontinued, the developers can only make the following changes:

- Change the price of the product to 0 in the furnace
- Change the risk weight to 0 for the product

Lets assume a user has this asset in their account, and then the protocol, and the developers intend to delist it. The user will still have the product in their `subAccountAssets` set.

- During health checks, this product will still be weighed in, since it is present in the user's asset set.

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Furnace.sol#L144-L151

Thus to prevent the product from contributing to the health, either the `price` or the `riskWeight` of the product needs to be set to 0.

- However, if the price or the weight is set to 0, this lets on-chain liquidators claim the product at a price of 0 or much lower than intended. This is because of how the `_getLiquidationPrice` function is implemented.

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Liquidation.sol#L352-L360

All of these arguments are completely on-chain interactions. The off-chain engine is expected to maybe ignore product interactions from a separate blacklist as a best-case scenario.

Thus there is no way for the devs to set the protocol in such a way that the bad assets are ignored, and the users are able to withdraw them if necessary. Also, users are still allowed to deposit more of the same asset into their accounts, since the `deposit` via the contract only checks if the product exists, not if it is tradable or not. The productId is guaranteed to exist, since any product with a defined `baseAsset` cannot be modified.

```
if (products[productId].baseAsset != address(0)) revert
↪  Errors.ProductAlreadySet();
```

Since assets once supported by the protocol can be lost or sold off at very low rates, this is a medium severity issue.

Impact

Assets cannot be de-listed from the protocol due to missing `isProductTradeable` check.

Code Snippet

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Furnace.sol#L144-L151

Tool used

Foundry

Recommendation

Implement checks on `isProductTradeable` during health checks as well as liquidations and all on-chain product interactions. The value already exists but is unused. On-chain contracts need to check this since on-chain workflows are not expected to be aware of off-chain blacklists.

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/ProductCatalogue.sol#L76-L84

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Fix Review: Fixed. No more on-chain actions possible for the user, everything goes via offchain app which needs to check this status, so no on-chain check is required.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-5: Users might not be able to withdraw perp profits or even just collateral if not enough core collateral asset is deposited as collateral by all protocol users

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/32

The protocol has acknowledged this issue.

Found by

hash, panprog

Summary

Users can deposit any collateral to support their perp positions, however settlement is done in core collateral asset. The problem is the mismatch between the realized profit and loss (which is forced to be in core collateral asset) and unrealized profit and loss (which is theoretical and is not forced to be in core collateral asset). When a lot of deposits are in non-core collateral assets and then some profit is realized, transactions to withdraw this profit will revert, because system won't have enough core collateral asset and there is no way to "force" conversion of deposited assets into core collateral asset.

This means that it's easily possible that users can't withdraw core collateral asset due to protocol not having enough of it, which can happen by itself or crafted by malicious user.

Vulnerability Detail

Possible scenario of the issue happening by itself:

1. Alice deposits 10 BTC, Bob deposits 100 ETH, Charlie deposits 5000 USDC (USDC = core collateral asset)
2. Alice opens 1 BTC long position with Bob being the maker (1 BTC short position), at the price of BTC = $10000
3. BTC price rises to $20000. Alice now has unrealized profit of $10000, Bob has unrealized loss of $10000.
4. Alice closes her position against Charlie, realizing 10000 USDC profit.
5. Alice now has 1 BTC + 10000 USDC, Bob has 100 ETH and 1 BTC short position with unrealized loss of 10000 USDC, Charlie has 5000 USDC and 1 BTC long position (with 0 unrealized pnl).

6. Alice tries to withdraw 10000 USDC profit, but transaction reverts, because the protocol only has 5000 USDC.
7. Alice withdraws 5000 USDC. Now Charlie wants to withdraw some of its deposit, but any amount will revert, because protocol has no core collateral deposited at all.

The issue arises when some profit is realized in core collateral, but the loss remains unrealized thus deposited assets are not forced to be converted to core collateral asset.

Possible scenario of the issue being forced by malicious user:

1. Protocol has 1 BTC, 10 ETH and 10000 USDC deposited (USDC = core collateral asset)
2. Malicious user Alice wants to cause panic for protocol users
3. Alice uses 2 subaccounts, depositing 10 BTC into acc#1 and 10 BTC into acc#2
4. Alice uses acc#1 to open 100 BTC long position, and at the same time uses acc#2 to open 100 BTC short position (price = $10000)
5. Alice waits for BTC to rise or fall until either acc#1 or acc#2 is in 10000 USDC profit.
6. For example, BTC has dropped to $9900, acc#1 has 10000 USDC unrealized loss, acc#2 has 10000 USDC unrealized profit.
7. Alice closes and immediately reopens 100 BTC short position in acc#2, withdrawing 10000 USDC of realized profit.

After these steps, Alice can go wreck havoc by saying to users to try to withdraw their core collateral asset, and when transactions revert, this can cause panic and everybody withdrawing and creating negative publicity and reputation: it's always bad when users can not withdraw their assets unexpectedly.

Impact

Users might be unable to withdraw any deposited assets or realized profit in core collateral asset. This can happen by itself, or caused by malicious user.

Code Snippet

Health calculation includes any deposited asset: https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/libraries/MarginDirective.sol#L94-L100

This means that users can deposit non-core collateral asset to support their open positions, but the profit is realized in core collateral asset:

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/OrderDispatch.sol#L599-L607

This means that protocol might not have enough core collateral asset to pay out the realized profits, when the loss is not yet realized (and the loss can stay unrealized for a long time).

Tool used

Manual Review

Recommendation

The solution depends a lot on what the direction the protocol wants to go. Some possible solutions:

- Allow forced settlement of unrealized profit (something similar to auto deleveraging, only soft version to auto settle unrealized profit, converting part of collateral asset into core collateral asset to support unrealized loss when there is not enough core collateral in the system)
- Create some kind of "reserves" fund in core collateral asset, which will be used to cover shortfalls. It might be combined with insurance fund.

Issue M-6: Incorrect spread liquidation for assets with 1e18 spreadPenalty

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/48

Found by

KupiaSec, hash

Summary

Incorrect spread liquidation for assets with 1e18 spreadPenalty

Vulnerability Detail

Assets for which spread is not supported are given a maintanence weight of 1e18

```
function getSubAccountHealth(
    address subAccount,
    bool isInitial
) public view returns (int256 health) {

    ....

        uint256 spreadQuantity = spreadPenalties[spotAssetAddress]
            .maintenance == 1e18
            ? 0
            : _subAccountSpreadQuantity(spotBalance, perpPos);
```

Hence such positions from the view of liquidation must be treated as a seperate spot + perp position instead of a normal spread. But when liquidating, this condition is not checked.

```
function liquidateSubAccount(
    Structs.LiquidateSubAccount calldata txn
) external {

    ....

    // @audit condition checked to liquidate as spread is that user doesn't have
↪   any naked perps which doesn't account for spreadPenalty == 1e18

    if (txn.liquidationMode == uint8(LiquidationMode.SPREAD)) {
        // liquidating spread
        // must not have any naked perp positions open (those not part of a
↪   spread)
        // if naked perp positions exist, liquidate those first
        if (_userHasNakedPerps(liquidateeSubAccount)) revert
↪   Errors.LiquidateNakedPerpsFirst();
        _liquidateSpread(txn);
    } else if (txn.liquidationMode == uint8(LiquidationMode.SPOT)) {
```

Hence a huge spread penalty (1e18) is incorrectly applied on the liquidation which results in the spot asset being liquidated at 0 price and the perp short position being liquidated at 2x price.

```
function _getSpreadLiquidationPrice(
    address spotComponentAddress,
    uint256 oraclePrice,
    bool isSpot
) internal view returns (uint256 liquidationPrice) {
    uint64 spreadPenalty = _furnace()
        .getSpreadPenalty(spotComponentAddress)
        .maintenance;
    if (isSpot) {
        return oraclePrice.mul(1e18 - spreadPenalty);
    } else {
        // is short perp component, liquidate at a higher price
        return oraclePrice.mul(1e18 + spreadPenalty);
    }
```

This will cause the liquidatee to loose their funds at the gain of the liquidator

Impact

Loss of funds for the liquidatee

Code Snippet

Liquidating as spread only checks for naked perps existance https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Liquidation.sol#L112-L117

Naked perps doesn't factor spreadPenalty == 1e18 https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Liquidation.sol#L381-L424

Spread liquidation price calculation https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/Liquidation.sol#L363-L376

Tool used

Manual Review

Recommendation

Check whether spreadPenalty maintanence weight is 1e18 inside `liquidateSubAccount` and if yes disallow to liquidate as a spread

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Fix Review: Fixed.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Issue M-7: Attacker can make first time taker's loose their funds by inflating fees

Source: https://github.com/sherlock-audit/2024-01-100x-judging/issues/50

Found by

hash

Summary

Attacker can make first time taker's loose their funds by inflating fees

Vulnerability Detail

For first time taker buy orders, there is a `matchOrderFee` which is calculated in the `_matchSpotOrder` function as shown below:

```
        if (o.takerSide) {
            takerFee = BasicMath.mul(o.baseQuantity, product.takerFee);
            makerFee = BasicMath.mul(quoteQuantity, product.makerFee);

            if (o.isFirstTime) {
=>              takerFee += BasicMath.div(
                    txFees[MATCH_ORDER_TX_FEE_INDEX],
                    o.executionPrice
                );
            }
        }
```

Since `BasicMath.div` does division in e18, ie. mutliplies by e18 before the division, an attacker can make the taker pay a large amount of fees by keeping a really low execution price. This will hugely inflate the fees of the taker. To minimize the loss of the attacker for selling at a low price, the attacker can keep a low quantity

A likely scenario is the beginning, when there are no opposite-side orders currently on the orderbook

Impact

First time takers loose funds

Code Snippet

https://github.com/sherlock-audit/2024-01-100x/blob/main/rysk-ciao-protocol/src/contracts/OrderDispatch.sol#L540-L550

Tool used

Manual Review

Recommendation

Enforce price limit checks / collect the fees always in core collateral / use oracle price for conversion

Discussion

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/rysk-finance/rysk-ciao-protocol/pull/41.

panprog

Escalate

This should be low, because:

1. The probability of it happening is extremely low: there must be basically no orderbook, so that all orderbook can be controlled by single entity.
2. Taker buying base asset is very likely to have no such asset deposited, especially with next to empty orderbook, thus the inflated fee can not be taken from the user and will revert the transaction.
3. Off-chain app should do some basic checks, for example, allowing tiny quantities doesn't make much sense (as they will eat too much gas fees for too low value), so this can be also considered a trusted offchain app input error (too small quantity matched). And if quantity is not very low, this becomes very costly to the maker. It should also probably check the trading price range to be within some sane range.

sherlock-admin2

> Escalate
>
> This should be low, because:
>
> 1. The probability of it happening is extremely low: there must be basically no orderbook, so that all orderbook can be controlled by single entity.

2. Taker buying base asset is very likely to have no such asset deposited, especially with next to empty orderbook, thus the inflated fee can not be taken from the user and will revert the transaction.
3. Off-chain app should do some basic checks, for example, allowing tiny quantities doesn't make much sense (as they will eat too much gas fees for too low value), so this can be also considered a trusted offchain app input error (too small quantity matched). And if quantity is not very low, this becomes very costly to the maker. It should also probably check the trading price range to be within some sane range.

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.

10xhash

The probability is low but the impact if it happens is high which is why I have marked it as a medium. The issue is with the fees of a taker order being determined by a param in the maker order and the ideal way to fix this would be on-chain changes (changing dependence on makerOrder.executionPrice) and not off-chain enforcements.

Czar102

If the execution price is far off, wouldn't it mean that the order is executed at a very favorable price?

10xhash

Yes But it can cause the loss to be higher than the gain

Eg: first time taker fees = 1e18 maker execution price = 1 wei, quantity = 1e18 actual market price = 1e18 calculated taker first time fees = 1e36 , amount gained by taker as part of favorable trade = 1e18

Czar102

@cvetanovv @kjr217 @JoshDAO can you verify whether this scenario is plausable? What's the most trivial offchain check that would prevent this?

cvetanovv

I can't decide if it's low/medium. In general, I agree with the 10xhash report, but what may invalidate it is the off-chain app. Let the protocol say what they think since they are more informed about the off-chain system. If they didn't have in mind this situation I think this issue could remain a Medium.

Indeed the off-chain mechanism could be used as an excuse and invalidate most reports, but in this particular case, I don't think they considered it.

panprog

Fix Review: Fixed. These fees are now applied by settling core collateral. The user can incur core collateral debt due to it now, which will be force swapped from his other assets using market price.

sherlock-admin

The Lead Senior Watson signed off on the fix.

Czar102

It's difficult to me to wrap my head around what is the exact cause of this. Isn't it the `executionPrice` being highly off? @panprog @cvetanovv @10xhash

panprog

@Czar102 Yes, you're right that it's due to extremely off (very low) `executionPrice`. For example, current ETH price is $1000, but orderbook is almost empty except for:
maker1: wants to buy 1 ETH for 1e-18 (1 wei) USDC (bid price = 1e-18 (1 wei))
maker2: wants to sell 0.0001 ETH for 1e-18 (1 wei) USDC (ask price = 1e-14 (10000 wei))

If any users comes up to this order-
book and tries to buy, say, 1 ETH for 1000 USDC, then his taker order (price wanted =
$1000orlower)willbematchedwithmaker2forpartialfillof0.0001ETHatthepricewhichisextremelysmall($1e
14). Theoretically, the price is much much better than expected, but due to this issue the fees for such order will be 1e32 ETH.

So basically the condition for this issue to happen is empty orderbook so that maker can submit an order with the price which is extremely small and not get matched with some other existing maker order. Such maker gains nothing (loses his 0.0001 ETH for basically nothing - 1 wei USDC), but forces the taker to pay extremely high fees (which will most probably revert, as the user has to have sufficient amount of the asset he wants to ackquire to pay this inflated fee).

Czar102

Shouldn't the off-chain engine simply make sanity checks regarding the `executionPrice`? @panprog @10xhash

panprog

@Czar102

> Shouldn't the off-chain engine simply make sanity checks regarding the `executionPrice`?

Yes, that's also part of my reasoning, point 3 of the escalation. Limiting order prices to something like +-20% of the oracle price to prevent bad mistakes is a reasonable thing to do for offchain app, which will also fix this issue, although there is nothing like that mentioned in the docs I think.

Czar102

Planning to accept the escalation and consider the issue informational.

10xhash

> Planning to accept the escalation and consider the issue informational.

Since this issue is not known to the protocol team and have not been mentioned in the docs why is this being considered informational? In the absence of checks, as I have mentioned before the impact is high and the probability is low and I also think the ideal fix is changing the dependence on execution price since the protocol wants to charge a fixed fee in the quote token.

kjr217

Im unsure on this issue. I personally think it is valid, hence why we implemented a fix for it. We cant make any assumptions about oracle price vs execution price because any percentage surrounding it could be completely arbitrary and since its an orderbook price and a perp, limit prices are allowed to deviate from the oracle price, all that happens is the funding increases and people might get liquidated. We might have users who want to be filled no matter what and could submit a price that is really high or really low, so an arbitrary oracle price check is kind of bad UX. Fat fingering is an issue with any exchange.

So i think the assumption that we should validate the execution price is within the oracle price is a mistake and actually does not make sense for the system and thus the finding is valid, hence why i implemented a fix for it.

Czar102

@10xhash "informational' means that the issue is to inform about something. Here, the issue informs that there is an important off-chain check to be done for the execution price not to be vastly manipulated.

@kjr217 whenever there is a possibility of a reasonable way to prevent an issue by the trusted off-chain actor, we are assuming it is used to ensure the fairness of the audit contest.

If it has been communicated at the time of the audit that the execution price won't be checked against a plausible value, or this is suggested, I will consider this a valid issue.

For now, I'm maintaining the previous stance – planning to consider the issue informational and accept the escalation.

kjr217

@Czar102 I never stated anywhere that there is any check of the limit price against an oracle price, that would break the behaviour of the system. If anything it makes the system more unsafe because it means we have to be aware of the volatility of a given asset and if it is ever more volatile than that then the system breaks because users cant trade so introducing such a check is a security risk to the system as far as im concerned. Checking the limit price against some arbitrary index/oracle price is not something I will ever implement. Im unsure where the assumption that thats something we should do has come from.

10xhash

> @10xhash "informational' means that the issue is to inform about something. Here, the issue informs that there is an important off-chain check to be done for the execution price not to be vastly manipulated.
>
> @kjr217 whenever there is a possibility of a reasonable way to prevent an issue by the trusted off-chain actor, we are assuming it is used to ensure the fairness of the audit contest.
>
> If it has been communicated at the time of the audit that the execution price won't be checked against a plausible value, or this is suggested, I will consider this a valid issue.
>
> For now, I'm maintaining the previous stance – planning to consider the issue informational and accept the escalation.

This issue doesn't inform that the protocol should enforce off-chain price limit checks. It is entirely up to the protocol to choose whether to enforce this or not and the team has decided to not do so. The issue is the unwanted dependence on the maker's execution price for which better mitigations exist as I have mentioned before.

Czar102

I see. In that case, I'm planning to reject the escalation and leave the issue as is.

Czar102

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: rejected

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.