SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Symmetrical
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	August 30 - September 4, 2023
Prepared on:	October 1, 2023

Introduction

Reimagining bilateral OTC Derivatives by combining them with Intent-Based execution. Allowing permissionless leverage trading of any asset, with hyperefficient just-in-time liquidity.

Scope

Repository: SYMM-IO/symmio-core

Branch: main

Commit: 0dec905617d3c6355ce4393de3e77274b90e2eb4

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
5	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

panprog	bitsurfer	tvdung94
xiaoming90	nobody2018	

Issue H-1: `liquidatePartyA` requires signature which doesn't have nonce, making possible unfair liquidation and loss of funds for all parties

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/5

Found by

panprog, xiaoming90

`liquidationSig` provided by liquidator to `liquidatePartyA` doesn't have nonces of neither partyA nor partyB. This means that this signature is valid even if partyA and/or partyB do some actions before the liquidation with this signature happens. There are numerous scenarios possible, both happening by itself and caused by malicious parties, where this can lead to loss of funds for some or all parties involved.

Vulnerability Detail

Example Scenario:

1. PartyA: allocated=109, upnl=-99, cva+lf=10. To avoid liquidation, partyA tries to close 50% of its position, submitting requestToClosePosition
2. PartyA: allocated=109, upnl=-100, cva+lf=10. PartyA becomes liquidatable
3. Liquidator submits liquidating transaction (with upnl=-100), but partyB at the same time fulfills partyA request
4. partyB transaction executes first, partyA has: allocated = 59, upnl=-50, cva+lf=5 (not liquidatable anymore)
5. Liquidator transaction executes (with upnl=-100): 4.1. Available Balance = 59-5 - 100 = -46 4.2. Liquidation type = OVERDUE (46 > 5), deficit = 41 4.3. Corresponding partyB receives 50 - 50*46/100 = 27 4.4. PartyA allocated balance is set to 0 4.5. Liquidators receives 0 (because it's OVERDUE liquidation)

The result:

1. PartyA was solvent just before the liquidation, but was still liquidated, losing all funds
2. PartyB has received 27 instead of full 50 profit (even though partyA had enough funds)
3. Liquidator didn't receive any fee, although it should have received it as partyA had enough funds.

All 3 parties have lost funds. Protocol basically stole those funds.

Impact

Unfair liquidation for partyA, loss of funds for liquidator and partyB in many possible situations during partyA liquidation - happening by itself or caused by malicious actors.

Proof of Concept

Add this to any test, for example to `ClosePosition.behavior.ts`.

```
it("PartyA wrong upnl liquidation", async function () {
  const context: RunContext = this.context;

  this.user_allocated = decimal(119);
  this.hedger_allocated = decimal(1000);

  this.user = new User(this.context, this.context.signers.user);
  await this.user.setup();
  await this.user.setBalances(this.user_allocated, this.user_allocated,
↪   this.user_allocated);

  this.hedger = new Hedger(this.context, this.context.signers.hedger);
  await this.hedger.setup();
  await this.hedger.setBalances(this.hedger_allocated, this.hedger_allocated);

  this.liquidator = new User(this.context, this.context.signers.liquidator);
  await this.liquidator.setup();

  // open position (100 @ 10)
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .positionType(PositionType.LONG)
    .quantity(decimal(100))
    .price(decimal(10))
    .cva(decimal(6)).lf(decimal(4)).mm(decimal(10))
    .build()
  );
  await this.hedger.lockQuote(1, 0, decimal(1));
  await this.hedger.openPosition(1, limitOpenRequestBuilder().filledAmount(decim ⌋
↪   al(100)).openPrice(decimal(10)).price(decimal(10)).build());

  var info = await this.user.getBalanceInfo();
  console.log("partyA allocated: " + info.allocatedBalances / 1e18 + " locked: "
↪   + info.totalLocked/1e18 + " pendingLocked: " + info.totalPendingLocked /
↪   1e18);
  var info = await this.hedger.getBalanceInfo(this.user.getAddress());
```

```
 console.log("partyB allocated: " + info.allocatedBalances / 1e18 + " locked: "
↪  + info.totalLocked/1e18 + " pendingLocked: " + info.totalPendingLocked /
↪  1e18);

 // price goes to 9, so user is in a loss of -100 (less fee), locked balance =
↪  10, so liquidatable
 // user tries to close half of its position to save liquidation
 //await this.user.setBalances(this.user_allocated, this.user_allocated,
↪  this.user_allocated);
 await this.user.requestToClosePosition(
   1,
   limitCloseRequestBuilder().quantityToClose(decimal(50)).closePrice(decimal(9
↪  )).build(),
 );

 // liquidators obtains signature to liquidate
 var sig = await getDummyLiquidationSig("0x10", decimal(-100), [1],
↪  [decimal(9)], decimal(-100));
 // fix timestamp
 sig.timestamp = BigNumber.from(await sig.timestamp).add(5);

 // but before liquidation, partyB fullfills partyA close request
 await this.hedger.fillCloseRequest(
   1,
   limitFillCloseRequestBuilder()
     .filledAmount(decimal(50))
     .closedPrice(decimal(9))
     .build(),
 );

 var info = await this.user.getBalanceInfo();
 console.log("after closing: partyA allocated: " + info.allocatedBalances /
↪  1e18 + " locked: " + info.totalLocked/1e18 + " pendingLocked: " +
↪  info.totalPendingLocked / 1e18);
 var info = await this.hedger.getBalanceInfo(this.user.getAddress());
 console.log("after closing: partyB allocated: " + info.allocatedBalances /
↪  1e18 + " locked: " + info.totalLocked/1e18 + " pendingLocked: " +
↪  info.totalPendingLocked / 1e18);

 // liquidator starts liquidating partyA
 await context.liquidationFacet.connect(this.liquidator.signer).liquidatePartyA(
   this.user.signer.address,
   sig
 );
 await context.liquidationFacet.connect(this.liquidator.signer).setSymbolsPrice(
   this.user.signer.address,
   sig
 );
```

```
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePositi
↪   onsPartyA(
    this.user.signer.address,
    [1]
  );

  var info = await this.user.getBalanceInfo();
  console.log("after liquidation: partyA allocated: " + info.allocatedBalances /
↪   1e18 + " locked: " + info.totalLocked/1e18 + " pendingLocked: " +
↪   info.totalPendingLocked / 1e18);
  var info = await this.hedger.getBalanceInfo(this.user.getAddress());
  console.log("after liquidation: partyB allocated: " + info.allocatedBalances /
↪   1e18 + " locked: " + info.totalLocked/1e18 + " pendingLocked: " +
↪   info.totalPendingLocked / 1e18);

});
```

Console execution result:

```
partyA allocated: 109 locked: 20 pendingLocked: 0
partyB allocated: 1000 locked: 20 pendingLocked: 0
after closing: partyA allocated: 59 locked: 10 pendingLocked: 0
after closing: partyB allocated: 1050 locked: 10 pendingLocked: 0
after liquidation: partyA allocated: 0 locked: 0 pendingLocked: 0
after liquidation: partyB allocated: 1079.5 locked: 0 pendingLocked: 0
liquidator balance: 0
```

As can be noticed, partyA + partyB have 1109 total after closing, but only 1079.5 after liquidation.

If the `fillCloseRequest` is commented out, then liquidation produces correct result:

```
partyA allocated: 109 locked: 20 pendingLocked: 0
partyB allocated: 1000 locked: 20 pendingLocked: 0
after liquidation: partyA allocated: 0 locked: 0 pendingLocked: 0
after liquidation: partyB allocated: 1106 locked: 0 pendingLocked: 0
liquidator balance: 3
```

Code Snippet

Notice that there are no nonces of either party in the liquidation signature:
https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/libraries/LibMuon.sol#L54-L67

Tool used

Manual Review

Recommendation

Include partyA and partyB nonces in the liquidation signature.

Discussion

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/34

xiaoming9090

Verified. Fixed in PR 34.

Issue H-2: `liquidatePositionsPartyA` limits partyB loss to partyB allocated balance, which can lead to inflated partyB balance and loss of funds for protocol users

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/6

Found by

bitsurfer, panprog

If partyB has positions both in a high profit and in a high loss, so that total upnl is much smaller than individual positions upnl, then partyB can deallocate most (or even all) of its funds. During partyA liquidation, `liquidatePositionsPartyA` limits partyB loss to the **current** allocated balance of partyB. In this case there is a big difference of the final results depending on order of position liquidations:

1. If the 1st position is in a profit, then the profit is first applied to balanceB, and then position in loss correctly subtracts from it (there is enough partyB balance to fully cover position in loss).
2. If the 1st position is in a loss, then balance (which is smaller than position loss) is simply reduced to 0, but then the 2nd position (which is in a profit) is added in full, leading to inflated final balance of partyB.

If such liquidation happens (intentional or not), partyB will have much more funds than it should at the expense of the other users: protocol will owe users more funds than it has, which can lead to bank run and the last users being unable to withdraw.

Vulnerability Detail

Example Scenario:

Position 1: upnl = -960, cva+lf = 10, mm=10 Position 2: upnl = +1000, cva+lf = 10, mm=10 Total cva+lf = 20, total mm=20 PartyB has allocated balance = 0 (available = 0 + 1000 - 960 = 40 >= 20 - not liquidatable) Party A is liquidated:

1. Position 1 is liquidated. `amount` = 960 `amountToDeduct` = 0 (because partyB allocated balance = 0) partyB allocatedBalance remains 0
2. Position 2 is liquidated. `amount` = 1000 partyB allocatedBalance increases by 1000 (is set to 1000)

The result: PartyB should have a balance of less than 60 (upnl+cva), but it has 1000 instead!

Impact

In some situations during partyA liquidations, PartyB balance is increased significantly while no funds enter the protocol, meaning protocol debt to users increases and is greater than protocol funds. This can cause a bank run, since the last users to withdraw will be unable to do so.

Code Snippet

`amountToDeduct` is calculated as minimum between partyB loss and partyB balance. This amount is deducted from **current** partyB allocated balance. https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L163-L166

Tool used

Manual Review

Recommendation

Calculate total (signed) pnl for positions for each partyB before applying it: `amountToDeduct` should be for all positions combined for a particular partyB instead of separate positions.

Discussion

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/36

xiaoming9090

Verified. Fixed in PR 36

Issue M-1: Wrong calculation of solvency in `fillCloseRequest` prevents the position from being closed even if the user is solvent after position closure

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/9

Found by

panprog, xiaoming90

This is an issue 184 from previous contest, developers fixed it only partially (request to close), but fill close request remains the same. If partyA has requested to close position, and partyB has called `fillCloseRequest`, the transaction will revert even though partyA is solvent after closure in the following scenario:

- partyA is not solvent if the position is closed at the current market price
- `closePrice` provided by the partyB is better than market price
- partyA is solvent if the position is closed at the `closePrice` provided by partyB

This is unfair for partyA and can lead to liquidation and loss of funds for partyA, even though it could have had the position closed correctly to make it solvent.

Vulnerability Detail

`isSolventAfterClosePosition` verifies that both partyA and partyB are solvent at the market price, then verifies that both partyA and partyB are solvent at the `closePrice`. However, if both parties are solvent at `closePrice` - this should be sufficient, as it's the most fair way for both parties - let the position close if both parties are solvent after it. But the requirement to be solvent both at market and close price prevents the transaction from being completed successfully.

Impact

PartyA can be liquidated and lose funds instead of correct closure of the position due to failed position closure transaction.

Code Snippet

`isSolventAfterClosePosition` requires enough balance both at the market and close price: https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/libraries/LibSolvency.sol#L101-L133

Tool used

Manual Review

Recommendation

Require both parties to only be solvent at `closePrice` when the position is closed, there is no point in reverting if the position closure can make both parties solvent, even though one of it is not solvent before that.

Discussion

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/32

xiaoming9090

Verified. Fixed in PR 32

Issue M-2: Position closure might always revert in some cases due to allocatedBalances being unsigned, preventing the user from closing its positions

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/10

Found by

panprog

This is issue 193 from the previous audit contest. The liquidation part was fixed, but the main reason described here is still the same.

Allocated balances are stored as unsigned ints.

https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/storages/AccountStorage.sol#L37

https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/storages/AccountStorage.sol#L41

Total account value is calculated as `allocated balance + unrealized pnl`, for example for liquidation purposes it's calculated as:

https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/libraries/LibAccount.sol#L83-L85

In the other cases formula is similar. This means that in case of high unrealized profit, allocated balance might be valid to be negative. For example, a simple case is if account has opened position with unrealized profit of 1000 and locked balance of 100, allocated balance can theoretically be `-900` with account being safe from liquidation: `-900 + 1000 = 100 >= lockedBalance(100)`. But since balance is unsigned, transaction to deallocate such balance will revert. This might be expected behaviour.

However, there are more complex cases possible where unsigned allocated balance can unexpectedly revert different transactions and block users from doing important actions. For example, if user has 2 opposite positions opened of the same quantity (LONG and SHORT) and price has moved significantly, total user `upnl = 0` (because opposite positions hedge each other completely). But the user will be unable to close either of these positions, because closing any position will subract high loss from the position from either partyA or partyB, which will make allocated balance negative and revert the transaction:

Before closing position:

`Account Balance = Position 1 UPNL (+1000) + Position 2 UPNL (-1000) + Allocated Balance (100) = +100` (account solvent)

After closing position:

`Account Balance should be = Position 1 UPNL (+1000) + Allocated Balance (-900) = +100` (account solvent), but it will revert.

As such, both partyA and partyB will lose funds deposited into the protocol to keep these positions alive. It will also be impossible to finish liquidation of such positions if any account becomes liquidatable, effectively locking the funds for extended time (until price is in the range where positions can be closed without big loss).

Vulnerability Detail

The scenario when user unexpectedly can't close position and loses funds:

1. Open 2 large opposite positions between partyA and partyB such that one of them is in a high loss and the other in the same profit. Use minimal allocated balance both for partyA and partyB (because such opposite positions can't be liquidated by themselves as they perfectly hedge each other).
2. When price moves significantly, try to close either of these positions, both will revert.
3. Both partyA and partyB are stuck with the funds in the protocol which they can't take out until price goes back close to initial value.

There might be the other cases with unexpected reverts and users being unable to close their positions.

Impact

PartyA and PartyB funds can be unexpectedly locked in the protocol for extended time. PartyA or PartyB can allocate additional funds to be able to close positions, then deallocate and withdraw, however depending on situation, required funds to close such position might be orders of magnitude higher than what parties have or are willing to risk (due to leverage).

This is High severity, because the situation can happen by itself for users having multiple positions: naturally some of them will be in profit, some in loss, and total upnl can be much smaller than individual positions, allowing a low allocated balance and making it impossible to close some positions.

Code Snippet

Add this to any test, for example to `ClosePosition.behavior.ts`.

```
it("Unable to close positions due to unsigned allocatedBalances", async function
↪   () {
  const context: RunContext = this.context;
  this.user_allocated = decimal(82);
  this.hedger_allocated = decimal(80);

  this.user = new User(this.context, this.context.signers.user);
  await this.user.setup();
  await this.user.setBalances(this.user_allocated, this.user_allocated,
↪   this.user_allocated);

  this.hedger = new Hedger(this.context, this.context.signers.hedger);
  await this.hedger.setup();
  await this.hedger.setBalances(this.hedger_allocated, this.hedger_allocated);

  // open 2 opposite direction positions
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .quantity(decimal(100))
    .price(decimal(1))
    .cva(decimal(10)).lf(decimal(5)).mm(decimal(15))
    .build()
  );
  await this.hedger.lockQuote(1, 0, decimal(4, 17));
  await this.hedger.openPosition(1, limitOpenRequestBuilder().filledAmount(decim ⌋
↪   al(100)).openPrice(decimal(1)).price(decimal(1)).build());

  await this.user.sendQuote(limitQuoteRequestBuilder()
    .positionType(PositionType.SHORT)
    .quantity(decimal(100))
    .price(decimal(1))
    .cva(decimal(10)).lf(decimal(5)).mm(decimal(15))
    .build()
  );
  await this.hedger.lockQuote(2, 0, decimal(4, 17));
  await this.hedger.openPosition(2, limitOpenRequestBuilder().filledAmount(decim ⌋
↪   al(100)).openPrice(decimal(1)).price(decimal(1)).build());

  var info = await this.user.getBalanceInfo();
  console.log("partyA allocated: " + info.allocatedBalances / 1e18 + " locked: "
↪   + info.totalLocked/1e18 + " pendingLocked: " + info.totalPendingLocked /
↪   1e18);
  var info = await this.hedger.getBalanceInfo(this.user.getAddress());
  console.log("partyB allocated: " + info.allocatedBalances / 1e18 + " locked: "
↪   + info.totalLocked/1e18 + " pendingLocked: " + info.totalPendingLocked /
↪   1e18);

  // now the price doubles, with upnl still 0
  // try to close LONG position

  await this.user.requestToClosePosition(
    1,
    limitCloseRequestBuilder().quantityToClose(decimal(100)).closePrice(decimal(
↪   2)).build(),
  );

  await expect(this.hedger.fillCloseRequest(
    1,
    limitFillCloseRequestBuilder()
      .filledAmount(decimal(100))
      .closedPrice(decimal(2))
      .build(),
  )).to.be.revertedWith("LibSolvency: PartyB will be liquidatable");

  console.log("Attempt to close LONG position failed due to partyB being
↪   liquidatable");

  // try to close SHORT position
  await expect(this.user.requestToClosePosition(
    2,
    limitCloseRequestBuilder().quantityToClose(decimal(100)).closePrice(decimal(
↪   2)).build(),
    )).to.be.revertedWith("LibSolvency: partyA will be liquidatable");

  console.log("Attempt to close SHORT position failed due to partyA being
↪   liquidatable");

});
```

Tool used

Manual Review

Recommendation

Make allocated balances signed ints and change all appropriate parts where allocated balances are changed or verified. This will be more user-friendly (less limiting) and will prevent situations described in this bug report.

Discussion

nevillehuang

@MoonKnightDev, seems possible due to possibility of UPNL being negative causing allocated balance to fall below 0 and reverting since it is an uint. Could you shed some light on why this is disputed?

MoonKnightDev

> @MoonKnightDev, seems possible due to possibility of UPNL being negative causing allocated balance to fall below 0 and reverting since it is an uint. Could you shed some light on why this is disputed?

One of the recognized challenges with cross positions relates to the scenario described above. To navigate this predicament:

If PartyA wishes to close the winning position, PartyB must allocate funds to fulfill the close request. Should PartyB act maliciously, PartyA should first allocate funds to close the losing position and then proceed to close the winning one.

panprog

Escalate

I believe this should be valid, because it causes all kinds of nasty underflows, the example I provided is just one scenario. Another scenario is partyA having allocated balance = 0 and positions in loss and profit with different partyBs, and then if partyB tries to call `emergencyClosePosition`, it will revert due to underflow (partyA position is in loss, but partyA allocated balance = 0, so the loss can not be applied).

> One of the recognized challenges with cross positions relates to the scenario described above. To navigate this predicament:
>
> If PartyA wishes to close the winning position, PartyB must allocate funds to fulfill the close request. Should PartyB act maliciously, PartyA should first allocate funds to close the losing position and then proceed to close the winning one.

While it's possible to allocate and then close for partyA, this is still unexpected behavior for partyA: if it has position 1 in profit and position 2 in a loss, and it only wants to close position 1 (to realize profit), partyA will be unable to do so - all transactions will revert due to partyB allocatedBalance underflow when trying to apply position loss. So partyA **must** close position 2, then position 1.

Another point to consider is that partyA might simply have not enough funds to allocate, close position 2, close position 1. For example, partyA opens huge long and short positions of similar size when ETH = 1000, with leverage = 10: Position 1: long 100 ETH (entry price = 1000) (cva+lf+mm=10000) Position 2: short 99 ETH (entry price = 1000) (cva+lf+mm=10000) Now ETH price shoots up to 2000. Both PartyA and PartyB still have 20000 collateral (which is enough to not be liquidated, because partyB total upnl is just 1000). partyA wants to close Position 1. Since it requires to reduce partyB allocatedBalance by 100 * (2000-1000) = 100000, it reverts.

Now if partyA wants to close Position 1, it first has to close Position 2, but in order to close it, partyA must have allocated balance of 99 * (2000 - 1000) = 99000. But partyA only has 20000, it doesn't have access to such huge amount. So it's still stuck.

sherlock-admin2

> Escalate
>
> I believe this should be valid, because it causes all kinds of nasty underflows, the example I provided is just one scenario. Another scenario is partyA having allocated balance = 0 and positions in loss and profit with different partyBs, and then if partyB tries to call `emergencyClosePosition`, it will revert due to underflow (partyA position is in loss, but partyA allocated balance = 0, so the loss can not be applied).
>
> > One of the recognized challenges with cross positions relates to the scenario described above. To navigate this predicament:
> >
> > If PartyA wishes to close the winning position, PartyB must allocate funds to fulfill the close request. Should PartyB act maliciously, PartyA should first allocate funds to close the losing position and then proceed to close the winning one.
>
> While it's possible to allocate and then close for partyA, this is still unexpected behavior for partyA: if it has position 1 in profit and position 2 in a loss, and it only wants to close position 1 (to realize profit), partyA will be unable to do so - all transactions will revert due to partyB allocatedBalance underflow when trying to apply position loss. So partyA **must** close position 2, then position 1.
>
> Another point to consider is that partyA might simply have not enough funds to allocate, close position 2, close position 1. For example, partyA opens huge long and short positions of similar size when ETH = 1000, with leverage = 10: Position 1: long 100 ETH (entry price = 1000) (cva+lf+mm=10000) Position 2: short 99 ETH (entry price = 1000) (cva+lf+mm=10000) Now ETH price shoots up to 2000. Both PartyA and PartyB still have 20000 collateral (which is enough to not be liquidated, because partyB total upnl is just 1000). partyA wants to close Position 1. Since it requires to reduce partyB allocatedBalance by 100 * (2000-1000) = 100000, it reverts.
>
> Now if partyA wants to close Position 1, it first has to close Position 2, but in order to close it, partyA must have allocated balance of 99 * (2000 - 1000) = 99000. But partyA only has 20000, it doesn't have access to such huge amount. So it's still stuck.

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.

nevillehuang

> Escalate
>
> I believe this should be valid, because it causes all kinds of nasty underflows, the example I provided is just one scenario. Another scenario is partyA having allocated balance = 0 and positions in loss and profit with different partyBs, and then if partyB tries to call `emergencyClosePosition`, it will revert due to underflow (partyA position is in loss, but partyA allocated balance = 0, so the loss can not be applied).
>
> > One of the recognized challenges with cross positions relates to the scenario described above. To navigate this predicament: If PartyA wishes to close the winning position, PartyB must allocate funds to fulfill the close request. Should PartyB act maliciously, PartyA should first allocate funds to close the losing position and then proceed to close the winning one.
>
> While it's possible to allocate and then close for partyA, this is still unexpected behavior for partyA: if it has position 1 in profit and position 2 in a loss, and it only wants to close position 1 (to realize profit), partyA will be unable to do so - all transactions will revert due to partyB allocatedBalance underflow when trying to apply position loss. So partyA **must** close position 2, then position 1.
>
> Another point to consider is that partyA might simply have not enough funds to allocate, close position 2, close position 1. For example, partyA opens huge long and short positions of similar size when ETH = 1000, with leverage = 10: Position 1: long 100 ETH (entry price = 1000) (cva+lf+mm=10000) Position 2: short 99 ETH (entry price = 1000) (cva+lf+mm=10000) Now ETH price shoots up to 2000. Both PartyA and PartyB still have 20000 collateral (which is enough to not be liquidated, because partyB total upnl is just 1000). partyA wants to close Position 1. Since it requires to reduce partyB allocatedBalance by 100 * (2000-1000) = 100000, it reverts.
>
> Now if partyA wants to close Position 1, it first has to close Position 2, but in order to close it, partyA must have allocated balance of 99 * (2000 - 1000) = 99000. But partyA only has 20000, it doesn't have access to such huge amount. So it's still stuck.

This does seem valid unless it is expected behavior for cross positions, any thoughts @MoonKnightDev ?

Navid-Fkh

We recognize that this is a limitation in our system.

What are we doing about it?
At the moment, nothing. PartyB will remain trusted for this version of contracts. As a result, PartyB will ensure that adequate funds are allocated before executing

fillClosingPosition. Moreover, users might need to close another position (like the one in a loss, as per your example) before they can close a profitable one.

Why don't we allow allocatedBalance to go negative?
In the initial stages of designing the system, we chose to make the allocatedBalance unsigned to prevent further potential security threats. For instance, if "muon" were to be compromised, a user would only be able to deallocate up to their allocated amount now. However, if we allowed the allocatedBalance to go negative with a positive UPNL, this could lead to a much more significant challenge. We understand that there are some attack vectors with a compromised version of muon even now, but these are now far more complex, enabling our anomaly detection bots to identify and halt malicious activity more effectively.

Will the system always operate like this, even in subsequent versions?
No. Future versions will introduce mechanisms that allow users to convert their unrealized profits into realized ones, providing a more effective solution for such scenarios.

nevillehuang

> We recognize that this is a limitation in our system.
>
> **What are we doing about it?** At the moment, nothing. PartyB will remain trusted for this version of contracts. As a result, PartyB will ensure that adequate funds are allocated before executing fillClosingPosition. Moreover, users might need to close another position (like the one in a loss, as per your example) before they can close a profitable one.
>
> **Why don't we allow allocatedBalance to go negative?** In the initial stages of designing the system, we chose to make the allocatedBalance unsigned to prevent further potential security threats. For instance, if "muon" were to be compromised, a user would only be able to deallocate up to their allocated amount now. However, if we allowed the allocatedBalance to go negative with a positive UPNL, this could lead to a much more significant challenge. We understand that there are some attack vectors with a compromised version of muon even now, but these are now far more complex, enabling our anomaly detection bots to identify and halt malicious activity more effectively.
>
> **Will the system always operate like this, even in subsequent versions?** No. Future versions will introduce mechanisms that allow users to convert their unrealized profits into realized ones, providing a more effective solution for such scenarios.

Very valid design decisions and concerns. I am now leaning towards Medium severity as I believe this is still a risk that users should be made known of before interacting with the protocol, that is the potential inability to close profiting positions without first closing losing ones.

panprog

> **What are we doing about it?** At the moment, nothing. PartyB will remain trusted for this version of contracts. As a result, PartyB will ensure that adequate funds are allocated before executing fillClosingPosition. Moreover, users might need to close another position (like the one in a loss, as per your example) before they can close a profitable one.

There is still a scenario I've outlined in the escalation comment: `emergencyClosePosition` might always revert for partyB, because partyA might have multiple positions both in profit and in loss with different partyBs so that it doesn't have enough allocated balance to close that position. In this case partyB can't do anything about it and partyA doesn't have any incentive to allocate additional funds to let partyB do emergency close.

I understand sponsor's point and intentions, but it's nonetheless a problem which can pop up in different scenarios unexpectedly, and not all of them might be fixable by actions such as depositing additional funds or closing the other positions first. And making it possible to realize unrealized pnl might also not fix all the problems, like the emergency close (maybe something else too).

In the contest context, I still think this is a valid issue as it's a real problem, which was perhaps overlooked in the previous contest as it was a duplicate to very similar problem which only listed liquidation as impact, but the problem is more general than just liquidations.

As for the impact, I leave it up to sherlock to decide. I think it's high as the situation is quite easy to get into during normal trading flow (I've seen a lot of users opening opposite direction positions to "hedge", there might be trading strategies like statistical arbitrage - going long one asset and short another similar asset etc) and very unexpected for the users (and not always easily fixable). Also, it was a valid high in previous contest and is not fixed but marked "will fix".

hrishibhat

Result: Medium Unique Based on the points raised in the escalation the underlying issue still persists as a limitation and given the possible cases where the underflow can cause problems as mentioned above. Also, I agree with the Lead judge's points. Considering this issue a valid medium

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

Issue M-3: Position value can fall below minimum acceptable quote value when partially closing positions requested to be closed in full

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/12

Found by

panprog

This is issue 248 from previous audit contest, which was fixed incorrectly. When PartyA requests to close LIMIT position in full, but partyB closes it partially, the remaining open quote can be below `minAcceptableQuoteValue`, breaking important protocol invariant, which can cause different problems, such as not enough incentive to liquidate dust positions.

Vulnerability Detail

In `LibQuote.closeQuote` there is a requirement to have the remaining quote value to not be less than `minAcceptableQuoteValue`:

```
if (LibQuote.quoteOpenAmount(quote) != quote.quantityToClose) {
    require(quote.lockedValues.total() >=
↪   symbolLayout.symbols[quote.symbolId].minAcceptableQuoteValue,
        "LibQuote: Remaining quote value is low");
}
```

Notice the condition when this require happens:

- `LibQuote.quoteOpenAmount(quote)` is remaining open amount
- `quote.quantityToClose` is requested amount to close

This means that this check is ignored if partyA has requested to close amount equal to full remaining quote value, but enforced when it's not (even if closing fully). For example, a quote with opened amount = 100 is requested to be closed in full (amount = 100): this check is ignored. But PartyB can fill the request partially, for example fill 99 out of 100, and the remainder (1) is not checked to confirm to `minAcceptableQuoteValue`.

The following execution paths are possible if PartyA has open position size = 100 and `minAcceptableQuoteValue` = 5:

- `requestToClosePosition(99)` -> revert
- `requestToClosePosition(100)` -> `fillCloseRequest(99)` -> pass (remaining quote = 1)

Impact

There can be multiple reasons why the protocol enforces `minAcceptableQuoteValue`, one of them might be the efficiency of the liquidation mechanism: when quote value is too small (and liquidation value too small too), liquidators will not have enough incentive to liquidate these positions in case they become insolvent. Both partyA and partyB might also not have enough incentive to close or respond to request to close such small positions, possibly resulting in a loss of funds and greater market risk for either user.

Proof of Concept

Add this to any test, for example to `ClosePosition.behavior.ts`.

```
it("Close position with remainder below minAcceptableQuoteValue", async function
↪  () {
  const context: RunContext = this.context;

  this.user_allocated = decimal(1000);
  this.hedger_allocated = decimal(1000);

  this.user = new User(this.context, this.context.signers.user);
  await this.user.setup();
  await this.user.setBalances(this.user_allocated, this.user_allocated,
↪  this.user_allocated);

  this.hedger = new Hedger(this.context, this.context.signers.hedger);
  await this.hedger.setup();
  await this.hedger.setBalances(this.hedger_allocated, this.hedger_allocated);

  await this.user.sendQuote(limitQuoteRequestBuilder()
    .quantity(decimal(100))
    .price(decimal(1))
    .cva(decimal(10)).lf(decimal(5)).mm(decimal(15))
    .build()
  );
  await this.hedger.lockQuote(1, 0, decimal(5, 17));
  await this.hedger.openPosition(1, limitOpenRequestBuilder().filledAmount(decim⌋
↪  al(100)).openPrice(decimal(1)).price(decimal(1)).build());

  // now try to close full position (100)
  await this.user.requestToClosePosition(
    1,
    limitCloseRequestBuilder().quantityToClose(decimal(100)).closePrice(decimal(⌋
↪  1)).build(),
  );
  // now partyA cancels request
  //await this.user.requestToCancelCloseRequest(1);

  // partyB can fill 99
  await this.hedger.fillCloseRequest(
    1,
    limitFillCloseRequestBuilder()
      .filledAmount(decimal(99))
      .closedPrice(decimal(1))
      .build(),
  );

  var q = await context.viewFacet.getQuote(1);
  console.log("quote quantity: " + q.quantity.div(decimal(1)) + " closed: " +
↪   q.closedAmount.div(decimal(1)));

});
```

Console execution result:

```
quote quantity: 100 closed: 99
```

Code Snippet

Notice the condition to perform the `minAcceptableQuoteValue` check: https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L155-L158

Tool used

Manual Review

Recommendation

The condition should be to ignore the `minAcceptableQuoteValue` if request is filled in full (filledAmount == quantityToClose):

```
-       if (LibQuote.quoteOpenAmount(quote) != quote.quantityToClose) {
+       if (filledAmount != quote.quantityToClose) {
            require(quote.lockedValues.total() >=
↪   symbolLayout.symbols[quote.symbolId].minAcceptableQuoteValue,
                "LibQuote: Remaining quote value is low");
        }
```

Discussion

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/31

xiaoming9090

Verified. Fixed in PR 31.

Issue M-4: MultiAccount `depositAndAllocateForAccount` function doesn't scale the allocated amount correctly, failing to allocate enough funds

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/15

Found by

panprog, tvdung94, xiaoming90

This is an issue very similar to issue 222 from previous audit contest, but in a `MultiAccount` smart contract.

`MultiAccount.depositAndAllocateForAccount` uses the same `amount` value both for `depositFor` and for `allocate`. However, deposit amount decimals are from the collateral token while allocate amount decimals = 18. This means that for USDC (decimals = 6), `depositAndAllocateForAccount` will deposit correct amount, but allocate amount which is 1e12 times smaller (dust amount).

Vulnerability Detail

Internal accounting (allocatedBalances) are tracked as fixed numbers with 18 decimals, while collateral tokens can have different amount of decimals. This is correctly accounted for in `AccountFacet.depositAndAllocate`:

```
AccountFacetImpl.deposit(msg.sender, amount);
uint256 amountWith18Decimals = (amount * 1e18) /
    (10 ** IERC20Metadata(GlobalAppStorage.layout().collateral).decimals());
AccountFacetImpl.allocate(amountWith18Decimals);
```

But it is treated incorrectly in `MultiAccount.depositAndAllocateForAccount`:

```
ISymmio(symmioAddress).depositFor(account, amount);
bytes memory _callData = abi.encodeWithSignature(
    "allocate(uint256)",
    amount
);
innerCall(account, _callData);
```

This leads to incorrect allocated amounts.

Impact

Similar to 222 from previous audit contest, the user expects to have full amount deposited and allocated, but ends up with only dust amount allocated, which can lead to unexpected liquidations (for example, user is at the edge of liquidation, calls depositAndAllocate to improve account health, but is liquidated instead). For consistency reasons, since this is almost identical to 222, it should also be high.

Code Snippet

The same amount is used for `depositFor` and `allocate`: https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/multiAccount/MultiAccount.sol#L167-L173

Tool used

Manual Review

Recommendation

Scale amount correctly before allocating it:

```diff
    ISymmio(symmioAddress).depositFor(account, amount);
+   uint256 amountWith18Decimals = (amount * 1e18) /
+       (10 ** IERC20Metadata(collateral).decimals());
    bytes memory _callData = abi.encodeWithSignature(
        "allocate(uint256)",
-       amount
+       amountWith18Decimals
    );
    innerCall(account, _callData);
```

Discussion

sherlock-admin

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

0xyPhilic commented:

> invalid because internally the depositAndAllocateForAccount calls the allocate function which does not use scaled amount but the actual input amount

MoonKnightDev

No funds will be lost. The user simply needs to reallocate their balance. Therefore, the severity is not high since there's no loss of funds.

nevillehuang

Relating to this comment in the previous contest, it has the same root cause and potential same consequence, so could be valid H.

MoonKnightDev

> Relating to this comment in the previous contest, it has the same root cause and potential same consequence, so could be valid H.

The root cause differs. Here, you can easily rectify the allocatedBalance by allocating again. Moreover, no funds are lost

nevillehuang

> > Relating to this comment in the previous contest, it has the same root cause and potential same consequence, so could be valid H.
>
> The root cause differs. Here, you can easily rectify the allocatedBalance by allocating again. Moreover, no funds are lost

Ok can be valid M according to Impact mentioned in the submission.

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/35

xiaoming9090

Verified. Fixed in PR 35.

Issue M-5: PartyBFacetImpl.chargeFundingRate should check whether quoteIds is empty array to prevent partyANonces from being increased, causing some operations of partyA to fail

Source: https://github.com/sherlock-audit/2023-08-symmetrical-judging/issues/41

Found by

nobody2018

[PartyBFacetImpl.chargeFundingRate](https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L310-L315) can increase `partyANonces[partyA]` by 1 with the `quoteIds` empty array. Some functions that partyA can call will use `partyANonces[partyA]` to verify the signature. This opens up the opportunity for partyB to DOS partyA until PartyB makes a profit or reduces its losses.

Vulnerability Detail

```
File: symmio-core\contracts\facets\PartyB\PartyBFacetImpl.sol
310:     function chargeFundingRate(
311:         address partyA,
312:         uint256[] memory quoteIds,
313:         int256[] memory rates,
314:         PairUpnlSig memory upnlSig
315:     ) internal {
316:         LibMuon.verifyPairUpnl(upnlSig, msg.sender, partyA);
317:         require(quoteIds.length == rates.length, "PartyBFacet: Length not
 ↪  match");
318:         int256 partyBAvailableBalance =
 ↪  LibAccount.partyBAvailableBalanceForLiquidation(
319:             upnlSig.upnlPartyB,
320:             msg.sender,
321:             partyA
322:         );
323:         int256 partyAAvailableBalance =
 ↪  LibAccount.partyAAvailableBalanceForLiquidation(
324:             upnlSig.upnlPartyA,
325:             partyA
326:         );
327:         uint256 epochDuration;
328:         uint256 windowTime;
329:         for (uint256 i = 0; i < quoteIds.length; i++) {
......//quoteIds is empty array, so code is never executed.
390:            }
391:            require(partyAAvailableBalance >= 0, "PartyBFacet: PartyA will be
↪   insolvent");
392:            require(partyBAvailableBalance >= 0, "PartyBFacet: PartyB will be
↪   insolvent");
393:            AccountStorage.layout().partyBNonces[msg.sender][partyA] += 1;
394:->          AccountStorage.layout().partyANonces[partyA] += 1;
395:        }
```

As long as `partyBAvailableBalance`(L318) and `partyAAvailableBalance`(L323) are greater than or equal to 0, that is to say, PartyA and PartyB are solvent. Then, partyB can add 1 to `partyANonces[partyA]` at little cost which is the gas of tx.

An example is given to illustrate how to cause losses for partyA. Assume that partyA requests to close a quote via [PartyAFacetImpl.requestToClosePosition](https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L150). PartyB ignored it. partyA can only wait for `maLayout.forceCloseCooldown` seconds, and then call [PartyAFacetImpl.forceClosePosition](https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L239) to forcefully close the quote.

```
File: symmio-core\contracts\facets\PartyA\PartyAFacetImpl.sol
239:        function forceClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory
↪   upnlSig) internal {
240:            AccountStorage.Layout storage accountLayout =
↪   AccountStorage.layout();
241:            MAStorage.Layout storage maLayout = MAStorage.layout();
242:            Quote storage quote = QuoteStorage.layout().quotes[quoteId];
....../assume codes here are executed
273:->          LibMuon.verifyPairUpnlAndPrice(upnlSig, quote.partyB, quote.partyA,
↪   quote.symbolId);
......
283:        }
```

L273, [LibMuon.verifyPairUpnlAndPrice](https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/libraries/LibMuon.sol#L181-L182) is used to verify signatures, where `partyBNonces[partyB][partyA]` and `partyANonces[partyA]` are used internally.

If the current price goes against partyB, then partyB can front-run `forceClosePosition` and call `chargeFundingRate` to increase the nonces of both parties by 1. In this way, partyA's `forceClosePosition` will inevitably revert because the nonces are incorrect.

Similarly, if partyA wants to deallocate funds via [AccountFacetImpl.deallocate](https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/con tracts/facets/Account/AccountFacetImpl.sol#L53), partyB can also prevent this operation via `chargeFundingRate`.

Impact

Due to this issue, partyB can increase nonces of any partyA with little cost, causing some operations of partyA to fail (refer to the Vulnerability Detail section). This opens up the opportunity for partyB to turn the table.

Code Snippet

https://github.com/sherlock-audit/2023-08-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L310-L315

Tool used

Manual Review

Recommendation

```
File: symmio-core\contracts\facets\PartyB\PartyBFacetImpl.sol
310:     function chargeFundingRate(
311:         address partyA,
312:         uint256[] memory quoteIds,
313:         int256[] memory rates,
314:         PairUpnlSig memory upnlSig
315:     ) internal {
316:         LibMuon.verifyPairUpnl(upnlSig, msg.sender, partyA);
317:-        require(quoteIds.length == rates.length, "PartyBFacet: Length not
↪   match");
317:+        require(quoteIds.length > 0 && quoteIds.length == rates.length,
↪   "PartyBFacet: Length is 0 or Length not match");
```

Discussion

sherlock-admin

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

panprog commented:

> valid medium

securitygrid

Escalate This is not a dup of #31. Please review it. This issue describes how PartyB uses `chargeFundingRate` to cause PartyA to suffer losses. **All PartyB can easily take**

advantage of it against PartyA, making themselves always profitable. Consider the following two situations here:

1. If the current price goes for PartyB, then the quote is closed and PartyB makes a profit.
2. If the current price goes against PartyB, then PartyB can use this issue to prevent PartyA from forcibly closing the quote until the price goes for PartyB.

sherlock-admin2

> Escalate This is not a dup of #31. Please review it. This issue describes how PartyB uses `chargeFundingRate` to cause PartyA to suffer losses. **All PartyB can easily take advantage of it against PartyA, making themselves always profitable**. Consider the following two situations here:
>
> 1. If the current price goes for PartyB, then the quote is closed and PartyB makes a profit.
> 2. If the current price goes against PartyB, then PartyB can use this issue to prevent PartyA from forcibly closing the quote until the price goes for PartyB.

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.

nevillehuang

> Escalate This is not a dup of #31. Please review it. This issue describes how PartyB uses `chargeFundingRate` to cause PartyA to suffer losses. **All PartyB can easily take advantage of it against PartyA, making themselves always profitable**. Consider the following two situations here:
>
> 1. If the current price goes for PartyB, then the quote is closed and PartyB makes a profit.
> 2. If the current price goes against PartyB, then PartyB can use this issue to prevent PartyA from forcibly closing the quote until the price goes for PartyB.
>
> So this issue can be H.

Agreed, although both stems from the same root case of abusing chargeFundingRate to increment nonce, it is not a duplicate of #31 given that issue is invalid. This is talking about partyB potentially causing losses to partyA not about preventing its own liquidation process.

@MoonKnightDev want to hear ur thoughts on this, given protocol determines party B is a trusted role.

panprog

Escalate

I believe this is a valid medium. It is similar to #31 in that it uses nonce increase to block certain functionality, however #31 only lists liquidation as impact (which is invalid), while this issue shows valid impact for partyB to be able to avoid `forceClosePosition`. I think function from #31 (`deallocate`) and also `transferAllocation` can also be used for the same (increase partyB nonce to prevent `forceClosePosition`).

So from my view:

- core reason: the same as #31 (which also mentions `chargeFundingRate`, without details though)
- impact: this issue - valid medium. #31 - invalid.

If identifiying core reason is enough to make issue valid, then #31 should also be a (valid) dup of this. But I personally think that lack of correct impact should keep it invalid.

As to why it's medium - it requires partyB to be malicious, and since partyB is semi-trusted role - as established in previous contest, all issues caused by malicious partyB should be treated as medium.

Based on all of this I think this should be a valid medium while #31 should remain invalid.

sherlock-admin2

> Escalate
>
> I believe this is a valid medium. It is similar to #31 in that it uses nonce increase to block certain functionality, however #31 only lists liquidation as impact (which is invalid), while this issue shows valid impact for partyB to be able to avoid `forceClosePosition`. I think function from #31 (`deallocate`) and also `transferAllocation` can also be used for the same (increase partyB nonce to prevent `forceClosePosition`).
>
> So from my view:
>
> - core reason: the same as #31 (which also mentions `chargeFundingRate`, without details though)
> - impact: this issue - valid medium. #31 - invalid.
>
> If identifiying core reason is enough to make issue valid, then #31 should also be a (valid) dup of this. But I personally think that lack of correct impact should keep it invalid.

> As to why it's medium - it requires partyB to be malicious, and since partyB is semi-trusted role - as established in previous contest, all issues caused by malicious partyB should be treated as medium.
>
> Based on all of this I think this should be a valid medium while #31 should remain 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.

nevillehuang

> Escalate
>
> I believe this is a valid medium. It is similar to #31 in that it uses nonce increase to block certain functionality, however #31 only lists liquidation as impact (which is invalid), while this issue shows valid impact for partyB to be able to avoid `forceClosePosition`. I think function from #31 (`deallocate`) and also `transferAllocation` can also be used for the same (increase partyB nonce to prevent `forceClosePosition`).
>
> So from my view:
>
> - core reason: the same as xiaoming90 - PartyB can block liquidation by incrementing the nonce #31 (which also mentions `chargeFundingRate`, without details though)
> - impact: this issue - valid medium. xiaoming90 - PartyB can block liquidation by incrementing the nonce #31 - invalid.
>
> If identifiying core reason is enough to make issue valid, then #31 should also be a (valid) dup of this. But I personally think that lack of correct impact should keep it invalid.
>
> As to why it's medium - it requires partyB to be malicious, and since partyB is semi-trusted role - as established in previous contest, all issues caused by malicious partyB should be treated as medium.
>
> Based on all of this I think this should be a valid medium while #31 should remain invalid.

Agree with this escalation. Unless there is a sufficient penalty mechanism in place for partyB to avoid abusing this vulnerability, this submission should be a valid medium. And given #31 wrongly identifies the attack path from the root cause, it is a valid low based on sherlocks rule here and thus is invalid:

- In addition to this, 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.

securitygrid

Agree that this is M since some issues about malicious partyB were M in previous contest.

hrishibhat

Result: Medium Unique Considering this a valid issue on its own based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- securitygrid: accepted
- panprog: accepted

MoonKnightDev

Fixed Code PR: https://github.com/SYMM-IO/symmio-core/pull/37/commits/80eb930a4c8ba8f4a89f17ad085412f9a41a11cd

xiaoming9090

Verified. Fixed in https://github.com/SYMM-IO/symmio-core/commit/80eb930a4c8ba8f4a89f17ad085412f9a41a11cd.