SHERLOCK

# SHERLOCK SECURITY REVIEW FOR

| | |
|---|---|
| **Contest type:** | Public |
| **Prepared for:** | Symmetrical |
| **Prepared by:** | Sherlock |
| **Lead Security Expert:** | xiaoming90 |
| **Dates Audited:** | June 17 - June 22, 2024 |
| **Prepared on:** | July 5, 2024 |

# Introduction

The novel derivatives Peer2Peer clearing infra, enabling LPs to provide synthetic leveraged exposure to any asset.

## Scope

Repository: SYMM-IO/protocol-core

Branch: develop

Commit: 742388ae6d9acf032ba500bdad7b084384af266f

---

For the detailed scope, see the contest details.

## Findings

Each issue has an assigned severity:

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

## Issues found

| Medium | High |
|---|---|
| 3 | 2 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

xiaoming90 slowfi 0xAadi

# Issue H-1: Wrong precision when adding balance within the `restoreBridgeTransaction` function

Source: https://github.com/sherlock-audit/2024-06-symmetrical-update-2-judging/issues/5

## Found by

0xAadi, slowfi, xiaoming90

## Summary

Wrong precision when adding balance within the `restoreBridgeTransaction` function, leading to loss of assets.

## Vulnerability Detail

In Line 90 below, the `AccountStorage.layout().balances` stores the account's balance in 18 precision, while the `bridgeTransaction.amount` stores the amount of token to be bridged in token native precision (e.g., USDC = 6 decimals).

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Bridge/BridgeFacetImpl.sol#L90

```
File: BridgeFacetImpl.sol
83:     function restoreBridgeTransaction(uint256 transactionId, uint256
↪  validAmount) internal {
84:         BridgeStorage.Layout storage bridgeLayout = BridgeStorage.layout();
85:         BridgeTransaction storage bridgeTransaction =
↪  bridgeLayout.bridgeTransactions[transactionId];
86:
87:         require(bridgeTransaction.status ==
↪  BridgeTransactionStatus.SUSPENDED, "BridgeFacet: Invalid status");
88:         require(bridgeLayout.invalidBridgedAmountsPool != address(0),
↪  "BridgeFacet: Zero address");
89:
90:
↪  AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
↪  (bridgeTransaction.amount - validAmount);
91:         bridgeTransaction.status = BridgeTransactionStatus.RECEIVED;
92:         bridgeTransaction.amount = validAmount;
93:     }
```

Assume that the number of tokens to bridge is 10000 USDC (10000e6). Thus, `bridgeTransaction.amount` will be set to 10000e6. The protocol detects an anomaly

with the bridging transaction and suspends it. After reviewing the transaction, the protocol decides to deduct 50% of the total bridged amount (5000 USDC).

The protocol executes `restoreBridgeTransaction` function with `validAmount` parameter set to 5000 USDC (1e6). The balance of "invalidBridgedAmountsPool" account will be incremented by 5000e6, as shown below. This is incorrect because the account balance in the protocol is denominated in 18 decimal precision. Over here, the code fails to convert the native token precision to the protocol's native precision (18) before assigning it to the account balance.

```
AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
↪  (bridgeTransaction.amount - validAmount);
AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
↪  10000e6 - 5000e6
AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
↪  5000e6
```

When the protocol attempts to withdraw the assets from the "invalidBridgedAmountsPool" account, the `accountLayout.balances[msg.sender]` will be 5000e6, and thus, the maximum value of `amountWith18Decimals` will be 5000e6. If `amountWith18Decimals` is 5000e6, the maximum `amount` that can be withdrawn will be 0.000000000000005 USDC based on the following formula.

The protocol should have received 5000 USDC, but due to a precision error, it could only receive a maximum of 0.000000000000005 USDC, resulting in a loss of assets.

```
amountWith18Decimals = (amount * 1e18) / (10 **
↪  IERC20Metadata(appLayout.collateral).decimals());
5000e6 = (amount * 1e18) / 1e6
5000e6 / 1e6 = amount * 1e18
5000 = amount * 1e18
amount = 5000/1e18
amount = 0.000000000000005
```

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Account/AccountFacetImpl.sol#L33

```
File: AccountFacetImpl.sol
26:     function withdraw(address user, uint256 amount) internal {
27:         AccountStorage.Layout storage accountLayout =
↪  AccountStorage.layout();
28:         GlobalAppStorage.Layout storage appLayout =
↪  GlobalAppStorage.layout();
29:         require(
30:             block.timestamp >= accountLayout.withdrawCooldown[msg.sender] +
↪  MAStorage.layout().deallocateCooldown,
```

```
31:             "AccountFacet: Cooldown hasn't reached"
32:         );
33:         uint256 amountWith18Decimals = (amount * 1e18) / (10 **
 ↪  IERC20Metadata(appLayout.collateral).decimals());
34:         accountLayout.balances[msg.sender] -= amountWith18Decimals;
35:         IERC20(appLayout.collateral).safeTransfer(user, amount);
36:     }
```

## Impact

Loss of assets due to precision error, as shown in the above scenario.

## Code Snippet

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Bridge/BridgeFacetImpl.sol#L90

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Account/AccountFacetImpl.sol#L33

## Tool used

Manual Review

## Recommendation

Scale up to the protocol's native precision of 18 decimals before assigning it to the account balance.

```
- AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
 ↪  (bridgeTransaction.amount - validAmount);
+ AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
 ↪  ((bridgeTransaction.amount - validAmount) * 1e18) / (10 **
 ↪  IERC20Metadata(appLayout.collateral).decimals());
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SYMM-IO/protocol-core/pull/45

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue H-2: Suspended bridge transactions cannot be restored

Source: https://github.com/sherlock-audit/2024-06-symmetrical-update-2-judging/issues/9

## Found by

slowfi, xiaoming90

## Summary

Suspended bridge transactions cannot be restored. As a result, the assets will be stuck, and bridge service providers cannot reclaim the assets they have transferred to the users from the protocol.

## Vulnerability Detail

In Line 90 below, when restoring the bridge transaction, the invalid assets will be deposited into the account of `bridgeLayout.invalidBridgedAmountsPool`. These invalid assets can be withdrawn from this account/pool at a later time.

Per Line 88 below, if the `bridgeLayout.invalidBridgedAmountsPool` is zero, the `restoreBridgeTransaction` transaction will revert.

However, within the codebase, there is no way to update the `bridgeLayout.invalidBridgedAmountsPool` value. Thus, the `bridgeLayout.invalidBridgedAmountsPool` will always be zero. The `restoreBridgeTransaction` transaction will always revert and there is no way to restore a suspended bridge transaction. As a result, the assets will be stuck, and bridge service providers will not be able to reclaim the assets they have transferred to the users from the protocol.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Bridge/BridgeFacetImpl.sol#L88

```
File: BridgeFacetImpl.sol
83:     function restoreBridgeTransaction(uint256 transactionId, uint256
↪  validAmount) internal {
84:         BridgeStorage.Layout storage bridgeLayout = BridgeStorage.layout();
85:         BridgeTransaction storage bridgeTransaction =
↪  bridgeLayout.bridgeTransactions[transactionId];
86:
87:         require(bridgeTransaction.status ==
↪  BridgeTransactionStatus.SUSPENDED, "BridgeFacet: Invalid status");
```

```
88:         require(bridgeLayout.invalidBridgedAmountsPool != address(0),
↪  "BridgeFacet: Zero address");
89:
90:
↪  AccountStorage.layout().balances[bridgeLayout.invalidBridgedAmountsPool] +=
↪  (bridgeTransaction.amount - validAmount);
91:         bridgeTransaction.status = BridgeTransactionStatus.RECEIVED;
92:         bridgeTransaction.amount = validAmount;
93:     }
```

## Impact

Loss of assets. The assets will be stuck, and bridge service providers cannot reclaim the assets they have transferred to the users from the protocol.

## Code Snippet

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Bridge/BridgeFacetImpl.sol#L88

## Tool used

Manual Review

## Recommendation

Implement a setter function for the `invalidBridgedAmountsPool` variable.

```
+ function updateInvalidBridgedAmountsPool(address poolAddress) external
↪  onlyRole(LibAccessibility.DEFAULT_ADMIN_ROLE) {
+  BridgeStorage.layout().invalidBridgedAmountsPool = poolAddress;
+ }
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SYMM-IO/protocol-core/pull/46

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-1: PartyA's allocated balance could increase after `deferredLiquidatePartyA` is executed

Source: https://github.com/sherlock-audit/2024-06-symmetrical-update-2-judging/issues/6

## Found by

xiaoming90

## Summary

PartyA's allocated balance could increase after `deferredLiquidatePartyA` is executed in an edge case, which could result in loss of assets.

## Vulnerability Detail

When the `deferredLiquidatePartyA` function is executed, if there is any excess allocated balance in `accountLayout.allocatedBalances[partyA]`, they will be transferred to the `accountLayout.partyAReimbursement[partyA]` at Line 42 below. Afterwards, the `accountLayout.allocatedBalances[partyA]` should not increase under any circumstance. Otherwise, an accounting error will occur.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/DeferredLiquidationFacetImpl.sol#L22

```
File: DeferredLiquidationFacetImpl.sol
22:     function deferredLiquidatePartyA(address partyA, DeferredLiquidationSig
↪  memory liquidationSig) internal {
23:         MAStorage.Layout storage maLayout = MAStorage.layout();
24:         AccountStorage.Layout storage accountLayout =
↪  AccountStorage.layout();
25:
26:         LibMuon.verifyDeferredLiquidationSig(liquidationSig, partyA);
27:
28:         int256 liquidationAvailableBalance =
↪  LibAccount.partyAAvailableBalanceForLiquidation(
29:             liquidationSig.upnl,
30:             liquidationSig.liquidationAllocatedBalance,
31:             partyA
32:         );
33:         require(liquidationAvailableBalance < 0, "LiquidationFacet: PartyA
↪  is solvent");
34:
```

```
35:         int256 availableBalance =
↪  LibAccount.partyAAvailableBalanceForLiquidation(
36:             liquidationSig.upnl,
37:             accountLayout.allocatedBalances[partyA],
38:             partyA
39:         );
40:         if (availableBalance > 0) {
41:             accountLayout.allocatedBalances[partyA] -=
↪  uint256(availableBalance);
42:             accountLayout.partyAReimbursement[partyA] +=
↪  uint256(availableBalance);
43:         }
```

However, there is an edge case where the `accountLayout.allocatedBalances[partyA]` can increase after the `deferredLiquidatePartyA` function is executed.

Assume Bob is a liquidator and a PartyA at the same time. As PartyA is permissionless, anyone can be a PartyA. There is no validation check in the codebase that prevents a liquidator from also being a PartyA AND no rules stated on the contest page that a liquidator cannot be a PartyA. Thus, it is fair to assume that some liquidators might use the same wallet to trade on the Symm App simultaneously (Users of Symm App = PartyA).

1. At T1, Bob's liquidator account took part in some liquidation process.
2. At T2, Bob's PartyA account gets liquidated by the `deferredLiquidatePartyA` function (Step 1 of liquidation process), and his `accountLayout.allocatedBalances[Bob]` has been set to zero at Line 41 above.
3. At T3. Some liquidations have been settled, and the liquidation fees are transferred to `accountLayout.allocatedBalances[Bob]` as per the code at Line 297-298 below. Assume that 1000 is paid out as a liquidation fee to Bob. Thus, `accountLayout.allocatedBalances[Bob] = 1000` now.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L297

```
File: LiquidationFacetImpl.sol
295:             uint256 lf =
↪  accountLayout.liquidationDetails[partyA].liquidationFee;
296:             if (lf > 0) {
297:
↪  accountLayout.allocatedBalances[accountLayout.liquidators[partyA][0]] += lf
↪  / 2;
298:
↪  accountLayout.allocatedBalances[accountLayout.liquidators[partyA][1]] += lf
↪  / 2;
```

```
299:                 emit
↪  SharedEvents.BalanceChangePartyA(accountLayout.liquidators[partyA][0], lf /
↪  2, SharedEvents.BalanceChangeType.LF_IN);
300:                 emit
↪  SharedEvents.BalanceChangePartyA(accountLayout.liquidators[partyA][1], lf /
↪  2, SharedEvents.BalanceChangeType.LF_IN);
301:             }
```

3. `accountLayout.allocatedBalances[Bob] = 1000` at this point. Next, the `deferredSetSymbolsPrice` function (Step 2 of liquidation process) will called against Bob's PartyA account.
4. The `deferredSetSymbolsPrice` is designed to always expect the `availableBalance` at Line 76 to be zero or negative to work properly. If `availableBalance` is positive, the logic and accounting will be incorrect. Because the `accountLayout.allocatedBalances[Bob] = 1000`, the `availableBalance` will become a positive value. Let's assume that `availableBalance` returned from `LibAccount.partyAAvailableBalanceForLiquidation` function at Line 76 is 800 after factoring the PnL (loss and/or deficit)
5. In Line 80 below, the `remainingLf` will be evaluated as 900. The liquidation fee of PartyA account will unexpectedly increase from 100 to 900, which is incorrect regarding the system's accounting. The liquidators should only receive up to `accountLayout.lockedBalances[partyA].lf` (100) of liquidation and not more than that amount. However, in this case, the liquidators received more than expected (900 instead of 100).

```
remainingLf = accountLayout.lockedBalances[partyA].lf - (- availableBalance);
remainingLf = 100 - (-800);
remainingLf = 900
```

6. As a result, 800 of the liquidation fee, which was supposed to belong to Bob, went to the liquidator's wallet. In this scenario, Bob lost 800.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/DeferredLiquidationFacetImpl.sol#L62

```
File: DeferredLiquidationFacetImpl.sol
62:     function deferredSetSymbolsPrice(address partyA, DeferredLiquidationSig
↪  memory liquidationSig) internal {
63:         MAStorage.Layout storage maLayout = MAStorage.layout();
64:         AccountStorage.Layout storage accountLayout =
↪  AccountStorage.layout();
65:
66:         LibMuon.verifyDeferredLiquidationSig(liquidationSig, partyA);
```

```
67:         require(maLayout.liquidationStatus[partyA], "LiquidationFacet:
↪  PartyA is solvent");
68:
69:         LiquidationDetail storage detail =
↪  accountLayout.liquidationDetails[partyA];
70:         require(keccak256(detail.liquidationId) ==
↪  keccak256(liquidationSig.liquidationId), "LiquidationFacet: Invalid
↪  liquidationId");
71:
72:         for (uint256 index = 0; index < liquidationSig.symbolIds.length;
↪  index++) {
73:
↪  accountLayout.symbolsPrices[partyA][liquidationSig.symbolIds[index]] =
↪  Price(liquidationSig.prices[index], detail.timestamp);
74:         }
75:
76:         int256 availableBalance =
↪  LibAccount.partyAAvailableBalanceForLiquidation(liquidationSig.upnl,
↪  accountLayout.allocatedBalances[partyA], partyA);
77:
78:         if (detail.liquidationType == LiquidationType.NONE) {
79:             if (uint256(- availableBalance) <
↪  accountLayout.lockedBalances[partyA].lf) {
80:                 uint256 remainingLf =
↪  accountLayout.lockedBalances[partyA].lf - uint256(- availableBalance);
81:                 detail.liquidationType = LiquidationType.NORMAL;
82:                 detail.liquidationFee = remainingLf;
83:             } else if (uint256(- availableBalance) <=
↪  accountLayout.lockedBalances[partyA].lf +
↪  accountLayout.lockedBalances[partyA].cva) {
84:                 uint256 deficit = uint256(- availableBalance) -
↪  accountLayout.lockedBalances[partyA].lf;
85:                 detail.liquidationType = LiquidationType.LATE;
86:                 detail.deficit = deficit;
87:             } else {
88:                 uint256 deficit = uint256(- availableBalance) -
↪  accountLayout.lockedBalances[partyA].lf -
↪  accountLayout.lockedBalances[partyA].cva;
89:                 detail.liquidationType = LiquidationType.OVERDUE;
90:                 detail.deficit = deficit;
91:             }
92:             accountLayout.liquidators[partyA].push(msg.sender);
93:         }
94:     }
```

## Impact

Loss of assets as shown in the above scenario.

## Code Snippet

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/DeferredLiquidationFacetImpl.sol#L22

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L297

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/DeferredLiquidationFacetImpl.sol#L62

## Tool used

Manual Review

## Recommendation

Implement the following additional logic to handle any edge case where PartyA's allocated balance could increase after `deferredLiquidatePartyA` (Step 1 of the liquidation process) is executed.

```
    function deferredSetSymbolsPrice(address partyA, DeferredLiquidationSig
↪   memory liquidationSig) internal {
        MAStorage.Layout storage maLayout = MAStorage.layout();
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();

        LibMuon.verifyDeferredLiquidationSig(liquidationSig, partyA);
        require(maLayout.liquidationStatus[partyA], "LiquidationFacet: PartyA is
↪   solvent");

        LiquidationDetail storage detail =
↪   accountLayout.liquidationDetails[partyA];
        require(keccak256(detail.liquidationId) ==
↪   keccak256(liquidationSig.liquidationId), "LiquidationFacet: Invalid
↪   liquidationId");

        for (uint256 index = 0; index < liquidationSig.symbolIds.length;
↪   index++) {
            accountLayout.symbolsPrices[partyA][liquidationSig.symbolIds[index]]
↪   = Price(liquidationSig.prices[index], detail.timestamp);
        }
```

```
        int256 availableBalance =
↪   LibAccount.partyAAvailableBalanceForLiquidation(liquidationSig.upnl,
↪   accountLayout.allocatedBalances[partyA], partyA);

        if (detail.liquidationType == LiquidationType.NONE) {
+               if (availableBalance > accountLayout.lockedBalances[partyA].lf) {
+                uint256 remainingLf = accountLayout.lockedBalances[partyA].lf;
+                detail.liquidationType = LiquidationType.NORMAL;
+                detail.liquidationFee = remainingLf;
+               } else if (uint256(- availableBalance) <
↪   accountLayout.lockedBalances[partyA].lf) {
-           if (uint256(- availableBalance) <
↪   accountLayout.lockedBalances[partyA].lf) {
                uint256 remainingLf = accountLayout.lockedBalances[partyA].lf -
↪   uint256(- availableBalance);
                detail.liquidationType = LiquidationType.NORMAL;
                detail.liquidationFee = remainingLf;
            } else if (uint256(- availableBalance) <=
↪   accountLayout.lockedBalances[partyA].lf +
↪   accountLayout.lockedBalances[partyA].cva) {
                uint256 deficit = uint256(- availableBalance) -
↪   accountLayout.lockedBalances[partyA].lf;
                detail.liquidationType = LiquidationType.LATE;
                detail.deficit = deficit;
            } else {
                uint256 deficit = uint256(- availableBalance) -
↪   accountLayout.lockedBalances[partyA].lf -
↪   accountLayout.lockedBalances[partyA].cva;
                detail.liquidationType = LiquidationType.OVERDUE;
                detail.deficit = deficit;
            }
            accountLayout.liquidators[partyA].push(msg.sender);
        }
    }
```

## Discussion

**sherlock-admin2**

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

**Hash01011122** commented:

> Low/Medium Vuln, Chances of occurrence of mentioned edge case is nearly zero.

**sherlock-admin2**



The protocol team fixed this issue in the following PRs/commits:
https://github.com/SYMM-IO/protocol-core/pull/48

**xiaoming9090**

Escalate

This issue should be at least a Medium Risk due to its significant impact (Loss of assets) if the risk is realized.

Per the protocol's source code, there is nothing to prevent a liquidator from also being a user (PartyA) since the protocol allows anyone to trade on its system. The PartyA role is permissionless and open to anyone. Also, no rules state on the contest page that a liquidator cannot be a PartyA or trade on the system.

Thus, it is entirely a valid scenario where a user is a liquidator and a PartyA simultaneously, leading to the issue mentioned in the report.

Per Sherlock's judging rules (https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue)

> V. How to identify a medium issue: Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

Thus, it meets Sherlock's requirement of a Medium Risk, where it causes a loss of funds but requires certain external conditions or specific states.

**sherlock-admin3**

> Escalate
>
> This issue should be at least a Medium Risk due to its significant impact (Loss of assets) if the risk is realized.
>
> Per the protocol's source code, there is nothing to prevent a liquidator from also being a user (PartyA) since the protocol allows anyone to trade on its system. The PartyA role is permissionless and open to anyone. Also, no rules state on the contest page that a liquidator cannot be a PartyA or trade on the system.
>
> Thus, it is entirely a valid scenario where a user is a liquidator and a PartyA simultaneously, leading to the issue mentioned in the report.
>
> Per Sherlock's judging rules (https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue)
>
> > V. How to identify a medium issue: Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite

> amount of funds, and any amount relevant based on the precision or significance of the loss.

Thus, it meets Sherlock's requirement of a Medium Risk, where it causes a loss of funds but requires certain external conditions or specific states.

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.

**MxAxM**

Possibility of this issue is near to zero so it should be low since it's not viable

**xiaoming9090**

> Possibility of this issue is near to zero so it should be low since it's not viable

Disagree. As mentioned in the report and escalation's comment, it is entirely a valid scenario where a user is a liquidator and a PartyA simultaneously, leading to the issue mentioned in the report.

In Sherlock, edge cases that have an impact that could potentially lead to a loss of assets are always judged as H/M. This view is derived with Sherlock's judging rule (https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue) below:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained.

**WangSecurity**

I think I need clarification on the attack path. I'll show how I see it and please correct me if it's wrong or not:

1. Bob is partyA.
2. Bob is liquidated by Alice and his balance is set at 0.
3. Bob does some liquidations himself and gets a liquidation fee of $1000 in total.
4. Alice calls `deferredSetSymbolsPrice` and after that Alice receives all the liquidation fees that Bob received above +- PnL, correct?
5. Bob loses these funds and Alice gets them (as I understand, doesn't even steal them necessarily).

If we simplify the attack path as much as possible, is it correct?

**xiaoming9090**

I think I need clarification on the attack path. I'll show how I see it and please correct me if it's wrong or not:

1. Bob is partyA.
2. Bob is liquidated by Alice and his balance is set at 0.
3. Bob does some liquidations himself and gets a liquidation fee of $1000 in total.
4. Alice calls `deferredSetSymbolsPrice` and after that Alice receives all the liquidation fees that Bob received above +- PnL, correct?
5. Bob loses these funds and Alice gets them (as I understand, doesn't even steal them necessarily).

If we simplify the attack path as much as possible, is it correct?

@WangSecurity

Each PartyA's account has a liquidation fee (LF) component, which is reserved and locked.

- Refer to explanation of Liquidation Fee (LF) at https://github.com/SYMM-IO/protocol-core?tab=readme-ov-file#sendquote
- Refer to the diagram at https://github.com/SYMM-IO/protocol-core?tab=readme-ov-file#liquidate-party-a where the blue section refer to the locked Liquidation Fee of a PartyA's account.

The PartyA's locked liquidation fee (LF) is the prize that will be paid to the liquidator. In my scenario, the locked LF of Bob's PartyA account is $100. Thus, any liquidator that liquidates Bob's PartyA is entitled to a maximum of $100, as per the protocol's specification. This liquidator fee was agreed upon by both the PartyA (maker) and PartyB (taker) when they opened the position and finalized within the protocol.

The protocol specification already clearly dictates that any liquidator who liquidates Bob's PartyA is entitled to a maximum of $100, which is PartyA's locked liquidation fee (LF). However, in the scenario mentioned in the report, the liquidator (Alice) received $900 instead of $100, which deviates from the protocol specification.

The liquidator (Alice) took more than what was expected from Bob's account. When a liquidator liquidates Bob's account, they are not entitled to all the assets left in the account. They are only entitled to $100 and not anything more than that.

**WangSecurity**

Thank you for that clarification, but there's still one part I might be missing. In your scenario, **after** Alice liquidates Bob (PartyA), and **before** she receives these $100 of LF, Bob also gets $1000. Is it because he liquidated the previous PartyA or other users?

**xiaoming9090**

> Thank you for that clarification, but there's still one part I might be missing. In your scenario, **after** Alice liquidates Bob (PartyA), and **before** she receives these $100 of LF, Bob also gets $1000. Is it because he liquidated the previous PartyA or other users?

@WangSecurity

Firstly, the term "PartyA" is the same as "User/Other Users". Users are normal people who trade on Symm platform and anyone with a blockchain wallet can do so (permissionless).

Secondly, Alice did not receive $100 in my scenario. Instead, she receives $900, which is incorrect and much more than expected. This amount exceeds the maximum allowed liquidation fee of $100 she is entitled. Effectively, $800 is "stolen" from Bob and Bob lost $800.

Following are the timeline of the events for reference:

**WangSecurity**

Thank you, but I'm not sure if it answers my question.

In T3, it says "Some liquidations have settled and Bob receives 1000 as a liquidation fee". It's Bob liquidating other users correct? Just need to clarify where these funds come from.

**xiaoming9090**

> Thank you, but I'm not sure if it answers my question.
>
> In T3, it says "Some liquidations have settled and Bob receives 1000 as a liquidation fee". It's Bob liquidating other users correct? Just need to clarify where these funds come from.

@WangSecurity

- At T1, Bob (Liquidator) kick-off the liquidation process against other users (e.g., James and Jack). James has 500 locked as LF, and Jack also has 500 locked as LF.
- At T3, the liquidations have settled, and the locked LF of James and Jack, which totals 1000, will be transferred to Bob. In short, the funds come from James and Jack, which Bob gains as compensation for the service he has provided in liquidating them.

**WangSecurity**

Thank you very much. I agree that this scenario is viable, even though it's very unlikely to occur, it's still possible with high constraints. Planning to accept the escalation and validate the issue with medium severity.

**WangSecurity**

Result: Medium Unique

**sherlock-admin2**

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-2: Deferred Liquidation can get stuck at step one of the liquidation process if the nonce increment

Source: https://github.com/sherlock-audit/2024-06-symmetrical-update-2-judging/issues/8

## Found by

xiaoming90

## Summary

Deferred Liquidation can get stuck at step one of the liquidation process if the nonce increments, leading to a loss of assets.

## Vulnerability Detail

Assume the following:

1. At T10, PartyA becomes liquidatable due to price movement, resulting in a drop in PnL
2. At T20, PartyA is solvent again. This can be due to various reasons, such as users allocating more funds or the prices recovering. Note that in the new version of the protocol, allocating funds does not lead to an increase in the account's nonce.

   [!NOTE]

   Per the documentation, the purpose of deferred liquidation is to allow the liquidator to liquidate the user who was liquidated at a certain block (it can be the current block or a block in the past). Even if PartyA is solvent in the current block, the account can still be liquidated if it has been eligible for liquidation in the past. The excess collateral in the accounts will returned back to the users.
3. Still at T20. A liquidator (Bob) is aware that PartyA is liquidatable at T10. Thus, he requests the deferred liquidation signature from Muon. At T10, PartyA's nonce is equal to 555. The `liquidationId`, `liquidationTimestamp`, `partyANonces` of the signature will be 100, T10, and 555, respectively. He proceeds to call the `deferredLiquidatePartyA` function with the signature, and the `liquidationId` of 100 is locked in the system. PartyA's liquidation status is set to true, and the account is technically "frozen".
4. At T21, one of the PartyBs that PartyA trades with triggers the `chargeFundingRate` function against the PartyA to charge a funding rate

(non-malicious event), or it can also be executed intentionally with malicious intention. Note that one PartyA can trade with multiple PartyBs. The function will check if PartyA is solvent, and the solvency check will pass because PartyA is solvent at this point (T21). At the end of the function, PartyA's nonce will increment to 556.

5. The liquidator's deferredSetSymbolsPrice transaction gets executed at T22. It is normal to have a slight delay due to many reasons, such as network congestion, low gas, or some preparation needed. Since the PartyA's nonce on the liquidation signature (555) is different from the current PartyA's nonce (556), the liquidation signature (liquidationId=100) will no longer be considered valid, and it cannot be used to proceed with the second step (set symbol price) of the liquidation process. The signature check is performed here.
6. If the liquidator attempts to fetch the liquidation signature for T10 from Muon again, PartyA's nonce of the signature will always remain at 555 because this is PartyA's nonce value at T10.
7. As a result, liquidation for PartyA will be struck.

## Impact

Loss of assets for the counterparty as the transfer of the assets from a liquidatable account to the counterparty cannot be made. The liquidation process cannot be completed as the liquidatable account is stuck.

In addition, since the liquidatable account is stuck, the assets under the accounts are also locked.

## Code Snippet

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/liquidation/DeferredLiquidationFacetImpl.sol#L62

## Tool used

Manual Review

## Recommendation

After step one of the deferred liquidation process, Party A's nonce should not change under any circumstance. Otherwise, the liquidation will be stuck.

Upon review of the codebase, it was found that only the `chargeFundingRate` function can update PartyA's nonce after step one of the deferred liquidation

process. Thus, consider adding the `notLiquidatedPartyA` modifier to the `chargeFundingRate` function to prevent anyone from charging the funding rate against a PartyA account already marked as liquidated to mitigate this issue.

```
    function chargeFundingRate(
        address partyA,
        uint256[] memory quoteIds,
        int256[] memory rates,
        PairUpnlSig memory upnlSig
+   ) external whenNotPartyBActionsPaused notLiquidatedPartyA(partyA) {
-   ) external whenNotPartyBActionsPaused {
        FundingRateFacetImpl.chargeFundingRate(partyA, quoteIds, rates, upnlSig);
        emit ChargeFundingRate(msg.sender, partyA, quoteIds, rates);
    }
```

## Discussion

**sherlock-admin4**

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

**Hash01011122** commented:

> Low/Med, Watson assumes Deferred liquidation can get stuck because of nonce increment via network congestion, low gas or some additional preparation which is unrealistic to occur as most of the liquidation is done in a single block still there is room for this to occur. Probability of this happening is low

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SYMM-IO/protocol-core/pull/50

**xiaoming9090**

Escalate

This issue should be at least a Medium Risk due to its significant impact (Loss of assets) if the risk is realized.

The report has demonstrated how it could lead to asset loss or stuck under normal conditions. The liquidation process is divided into multiple stages/steps as follows:

- Stage 1 - `deferredLiquidatePartyA`
- Stage 2 - `deferredSetSymbolsPrice`
- Stage 3 - `liquidatePendingPositionsPartyA`

- Stage 4 - `liquidatePositionsPartyA`

There is no requirement for the liquidator to complete all the stages in one go within a single block, and this requirement is also not being enforced within the smart contracts. As such, liquidators have the option to execute each of the above stages in a single or multiple blocks as they see fit.

Thus, it is a valid scenario where Stage 2 of the liquidation process is executed on a different block from Stage 1 of the liquidation process, resulting in the issue highlighted in this report to occur.

Per Sherlock's judging rules (https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue)

> V. How to identify a medium issue: Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

Whenever the liquidator does not execute Stage 1 and Stage 2 of the liquidation process within a single block, the issue mentioned in this report can occur, causing the user's asset to be stuck. Thus, it meets Sherlock's requirement of a Medium Risk, where it causes a loss of funds but requires certain external conditions or specific states.

**sherlock-admin3**

> Escalate
>
> This issue should be at least a Medium Risk due to its significant impact (Loss of assets) if the risk is realized.
>
> The report has demonstrated how it could lead to asset loss or stuck under normal conditions. The liquidation process is divided into multiple stages/steps as follows:
>
> - Stage 1 - `deferredLiquidatePartyA`
> - Stage 2 - `deferredSetSymbolsPrice`
> - Stage 3 - `liquidatePendingPositionsPartyA`
> - Stage 4 - `liquidatePositionsPartyA`
>
> There is no requirement for the liquidator to complete all the stages in one go within a single block, and this requirement is also not being enforced within the smart contracts. As such, liquidators have the option to execute each of the above stages in a single or multiple blocks as they see fit.

> Thus, it is a valid scenario where Stage 2 of the liquidation process is executed on a different block from Stage 1 of the liquidation process, resulting in the issue highlighted in this report to occur.
>
> Per Sherlock's judging rules (https://docs.sherlock.xyz/audits/judging/judging#v.-how-to-identify-a-medium-issue)
>
> > V. How to identify a medium issue: Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.
>
> Whenever the liquidator does not execute Stage 1 and Stage 2 of the liquidation process within a single block, the issue mentioned in this report can occur, causing the user's asset to be stuck. Thus, it meets Sherlock's requirement of a Medium Risk, where it causes a loss of funds but requires certain external conditions or specific states.

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.

**WangSecurity**

I need some clarification on how the loss of funds occurs, if the PartyA was liquidatable at T10, but is not liquidatable at T20, what is the loss of funds here exactly? Maybe a numeric example would be perfect here @xiaoming9090

Additionally, are PartyB and PartyA roles or contracts and as I understand they're considered Trusted?

**xiaoming9090**

> I need some clarification on how the loss of funds occurs, if the PartyA was liquidatable at T10, but is not liquidatable at T20, what is the loss of funds here exactly? Maybe a numeric example would be perfect here @xiaoming9090
>
> Additionally, are PartyB and PartyA roles or contracts and as I understand they're considered Trusted?

@WangSecurity

Question: `how the loss of funds occurs` Answer: When this issue is realized, the liquidation process will be stuck, and the account will be frozen. No one (users or liquidators) can retrieve the assets in the "frozen" account. Thus, all assets within

the frozen account are effectively lost. If the "frozen" account has $10000 worth of assets, this amount will be lost.

Question: `if the PartyA was liquidatable at T10, but is not liquidatable at T20` Answer: To simplify, it is a special new feature of Symm IO that allows a liquidator to "go back in time". Assume Bob's account becomes liquidatable for a moment due to volatile market movement (e.g., at T10). The price movement goes down and then up within a short period of time. Thus, when the liquidators detect it at T20, the account might be solvent again since the price has recovered. However, that does not matter for the liquidator because they are allowed to liquidate Bob's account based on old historical data at T10.

Question: `PartyB and PartyA roles or contracts and as I understand they’re considered Trusted?`

- PartyA is permissionless, and anyone can become a PartyA. Thus, it is considered as Untrusted.
- PartyB has to be whitelisted by the protocol. However, in my report's scenario, PartyB does not need to behavior maliciously. Thus, it does not matter whether PartyB is trusted or not. PartyBs will periodically charge funding rates on PartyAs, which is a normal operation and non-malicious event. When the `chargeFundingRate` function is triggered at T21, the issue will be realized.

**WangSecurity**

Thank you very much! And another question on the scenario, if we change it a bit to the following:

3. At T20, PartyB calls `chargeFundingRate`, changing the Nonce to 556.
4. At 21, Bob (liquidator) will request the deferred liquidation signature from Muon. But the signature will return the nonce of 555, correct?

**xiaoming9090**

> Thank you very much! And another question on the scenario, if we change it a bit to the following:
>
> 3. At T20, PartyB calls `chargeFundingRate`, changing the Nonce to 556.
> 4. At 21, Bob (liquidator) will request the deferred liquidation signature from Muon. But the signature will return the nonce of 555, correct?

@WangSecurity

Firstly, a PartyA can interact with many PartyBs within the system. At any point, each PartyB can call the `chargeFundingRate` function.

Secondly, the steps in the POC should not be swapped. The timing and sequence of each step are important for the edge case to be triggered. The report presents

an edge case that will happen when the `chargeFundingRate` function is triggered after the `deferredLiquidatePartyA` function, not the other way around.

On a side note, the sponsor has accepted the technical aspect of the POC, so there is no doubt about its correctness and the possibility of this edge case occurring. Otherwise, a fix would not be required if the edge case cannot occur.

The reason why this has been wrongly judged as Low in the first place is due to the following comments:

> Low/Med, Watson assumes Deferred liquidation can get stuck because of nonce increment via network congestion, low gas or some additional preparation which is unrealistic to occur as most of the liquidation is done in a single block still there is room for this to occur. Probability of this happening is low

However, in Sherlock, issues are not judged based on the probability that an issue can happen.

**WangSecurity**

Thank you, I agree that even though this case might be highly unlikely, it's viable and should be deemed valid. Planning to accept the escalation and validate the report with medium severity, since the constraints for this scenario are quite high.

And @xiaoming9090 just for your info, the comment from a judge you quoted is not from the Lead Judge, but from another Judge from the contest. That doesn't affect anything, just making sure you're aware of that.

**WangSecurity**

Result: Medium Unique

**sherlock-admin4**

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# Issue M-3: Collateral can still be allocated to PartyA when the system is paused by exploiting the new internal transfer function

Source: https://github.com/sherlock-audit/2024-06-symmetrical-update-2-judging/issues/11

## Found by

xiaoming90

## Summary

Collateral can still be allocated to PartyA when the system is paused by exploiting the new internal transfer function.

## Vulnerability Detail

The `allocate` and `depositAndAllocate` functions are guarded by the `whenNotAccountingPaused` modifier to ensure that collateral can only be allocated when the accounting is not paused.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Account/AccountFacet.sol#L48

```
File: AccountFacet.sol
46:     /// @notice Allows Party A to allocate a specified amount of collateral.
 ↪  Allocated amounts are which user can actually trade on.
47:     /// @param amount The precise amount of collateral to be allocated,
 ↪  specified in 18 decimals.
48:     function allocate(uint256 amount) external whenNotAccountingPaused
 ↪  notSuspended(msg.sender) notLiquidatedPartyA(msg.sender) {
49:         AccountFacetImpl.allocate(amount);
..SNIP..
52:     }
53:
54:     /// @notice Allows Party A to deposit a specified amount of collateral
 ↪  and immediately allocate it.
55:     /// @param amount The precise amount of collateral to be deposited and
 ↪  allocated, specified in collateral decimals.
56:     function depositAndAllocate(uint256 amount) external
 ↪  whenNotAccountingPaused notLiquidatedPartyA(msg.sender)
 ↪  notSuspended(msg.sender) {
57:         AccountFacetImpl.deposit(msg.sender, amount);
```

```
58:         uint256 amountWith18Decimals = (amount * 1e18) / (10 **
↪  IERC20Metadata(GlobalAppStorage.layout().collateral).decimals());
59:         AccountFacetImpl.allocate(amountWith18Decimals);
..SNIP..
63:     }
```

However, malicious users can bypass this restriction by exploiting the newly implemented `AccountFacet.internalTransfer` function. When the global pause (`globalPaused`) and accounting pause (`accountingPaused`) are enabled, malicious users can use the `AccountFacet.internalTransfer` function, which is not guarded by the `whenNotAccountingPaused` modifier, to continue allocating collateral to their accounts, effectively bypassing the pause.

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Account/AccountFacet.sol#L79

```
File: AccountFacet.sol
74:     /// @notice Transfers the sender's deposited balance to the user
↪  allocated balance.
75:     /// @dev The sender and the recipient user cannot be partyB.
76:     /// @dev PartyA should not be in the liquidation process.
77:     /// @param user The address of the user to whom the amount will be
↪  allocated.
78:     /// @param amount The amount to transfer and allocate in 18 decimals.
79:     function internalTransfer(address user, uint256 amount) external
↪  whenNotAccountingPaused whenNotInternalTransferPaused notPartyB
↪  userNotPartyB(user) notSuspended(msg.sender) notLiquidatedPartyA(user){
80:         AccountFacetImpl.internalTransfer(user, amount);
..SNIP..
85:     }
```

## Impact

When the global pause (`globalPaused`) and accounting pause (`accountingPaused`) are enabled, this might indicate that:

1) There is an issue, error, or bug in certain areas (e.g., accounting) of the system. Thus, the funds transfer should be halted to prevent further errors from accumulating and to prevent users from suffering further losses due to this issue
2) There is an ongoing attack in which the attack path involves transferring/allocating funds to an account. Thus, the global pause (`globalPaused`) and accounting pause (`accountingPaused`) have been activated to stop the attack. However, it does not work as intended, and the hackers

can continue to exploit the system by leveraging the new internal transfer function to workaround the restriction.

In both scenarios, this could lead to a loss of assets.

## Code Snippet

https://github.com/sherlock-audit/2024-06-symmetrical-update-2/blob/main/protocol-core/contracts/facets/Account/AccountFacet.sol#L79

## Tool used

Manual Review

## Recommendation

Add the `whenNotAccountingPaused` modifier to the `internalTransfer` function.

```
- function internalTransfer(address user, uint256 amount) external
 ↪  whenNotInternalTransferPaused notPartyB userNotPartyB(user)
 ↪  notSuspended(msg.sender) notLiquidatedPartyA(user){
+ function internalTransfer(address user, uint256 amount) external
 ↪  whenNotInternalTransferPaused notPartyB userNotPartyB(user)
 ↪  notSuspended(msg.sender) notLiquidatedPartyA(user){
   AccountFacetImpl.internalTransfer(user, amount);
```

## Discussion

**sherlock-admin2**

The protocol team fixed this issue in the following PRs/commits:
https://github.com/SYMM-IO/protocol-core/pull/47

**sherlock-admin2**

The Lead Senior Watson signed off on the fix.

# 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.